Security Module

Technical Reference

Version 2.1.0

Authors	Christian Bäuerle, Markus Schneider
Status	Released

Document Information

History

Author	Date	Version	Remarks
Christian Bäuerle	2010-10-13	1.0.4	Generalized technical reference
Christian Bäuerle	2012-07-02	1.0.5	Minor corrections
Markus Schneider	2013-11-27	2.0.0	Major rework
Markus Schneider	2014-03-13	2.1.0	Rework of ‘Functional Description’; added additional informations; minor corrections

Reference Documents

No.	Source	Title	Version
[1]	HIS	API IO Library	2.0.3
[2]	HIS	Security Module Specification	1.1
[3]	Vector	HexView Reference Manual	1.6
[4]	RSA	PKCS #1: RSA Cryptography Standard	2.1
[5]	IETF	HMAC: Keyed-Hashing for Message Authentication RFC2104	Feb 1997
[6]	ANSI	Keyed Hash Message Authentication Code X9.71	2000

Scope of the Document

This technical reference describes the general use of the Security Module software.

Contents

Illustrations

Tables

1 Introduction

This document describes the interface and configuration of the security module. The main target of this component is the usage within a flash process to provide the following functionalities depending on your order:

- Authorization
- Authentication, verification and error detection
- Confidentiality

The authorization part of the security component can be used for the Seed/Key procedure to unlock an ECU. The module provides a standard interface for both, seed and key generation.

Usually the functions are available in source code and can be adapted if needed. Parts of the cryptographic algorithms can be delivered in a library.

Authentication, verification and error detection is available in three different levels. They are distinguished by so-called security classes:

- Class DDD
- Class C
- Class CCC

Only one of the three classes can be selected at a time and differs in the strength of the mathematical algorithm.

The security classes C and CCC calculate a signature on one or more data segments. The resulting signature is compared against a signature that is given through the flash download. The checksum calculation with the security class DDD has in principle the same mechanism.

As the security class DDD uses a checksum algorithm, a key file is not needed. The classes C and CCC are using signature algorithm. They require a key parameter for calculation.

The given signatures must be calculated off-line on exactly the same set of data that the security component uses for verification. For security class C, key parameter for the on-line and off-line signature is the same. This is called a symmetric signature calculation.

For security class CCC, an asymmetric signature calculation is used. The off-line signature is calculated with a private key, and the on-line signature calculation (respectively the verification) uses the public key.

The length of the signature varies depending on the used security classes.

Confidentiality can be achieved by data encryption and decryption. The security module offers an interface for symmetrical encryption / decryption.

2 Software Architecture

The Security Module delivery contains the following components:

> The security module itself (with its sub-packages)

> The cryptographic algorithms as a library

> Configuration tool GENy

In addition, the following tool is delivered:

> The software tool HexView. It can be used to change and show the contents of a binary file. Also it can be used for signature and checksum calculation. Besides, the creation of encrypted download files is possible with HexView.

2.1 Security Module Components

The functionalities of the Security Module are separated into several sub-packages. A functional description for the individual modules can be found in chapter 6.

Depending on your order the following modules can be provided with the Security Module:

Name	Description
CRC	Implementation of the HIS security module - CRC calculation. Offers CRC calculation according to CRC-16 CCITT or CRC-32 (IEEE 802.3).
Decryption	Implementation of the HIS security module – Decryption. Offers API for decryption and encryption.
SeedKey	Implementation of the HIS security module - Seed/key authentication. Offers challenge/response authentication interface.
Verification	Implementation of the verification component of the HIS security module - Signature verification. Offers signature/checksum verification interface.
Workspace	Defines the default workspaces used by the library parts for verification, seed/key and decryption.
Sec	Implementation of the HIS security module.

Table 2-1 Security Module Components

3 Integration

This chapter gives necessary information for the integration of the Vector Security Module into an application environment of an ECU.

3.1 Scope of Delivery

The delivery of the Security Module contains the following files:

Module	Filename	Delivery Type		Description
		Source	Lib	
CRC	Sec_Crc.c	■		Source file of the CRC Module.
	Sec_Crc.h	■		Header file of the CRC Module.
Decryption	Sec_Decryption.c	■		Source file of the Decryption Module.
	Sec_Decryption.h	■		Header file of the Decryption Module.
SeedKey	Sec_SeedKey.c	■		Source file of the Seed/Key Module.
	Sec_SeedKey.h	■		Header file of the Seed/Key Module.
Verification	Sec_Verification.c	■		Source file of the Verification Module.
	Sec_Verification.h	■		Header file of the Verification Module.
Workspace	Sec_Workspace.h	■		Header file of the Workspace.
Common	Sec_Inc.h	■		Header file used internally.
	Sec_Types.h	■		Defines types, constantans and configuration switches.
	Sec.c	■		Source file of the Security Module.
	Sec.h	■		Header file of the Security Module.
	SecModLib.a [1]		■	Contains cryptographic primitives and partial code of the Security Module.

Table 3-1 Scope of Delivery

3.2 Include Structure

For usage of the Security Module within an application it is necessary to include the 'Sec.h' header file.

3.3 Compiler Abstraction and Memory Mapping

Depending on the configuration the workspace is mapped to a specific section according to the 'memmap.h' file.

[1] The filename and file extension may differ from your delivery

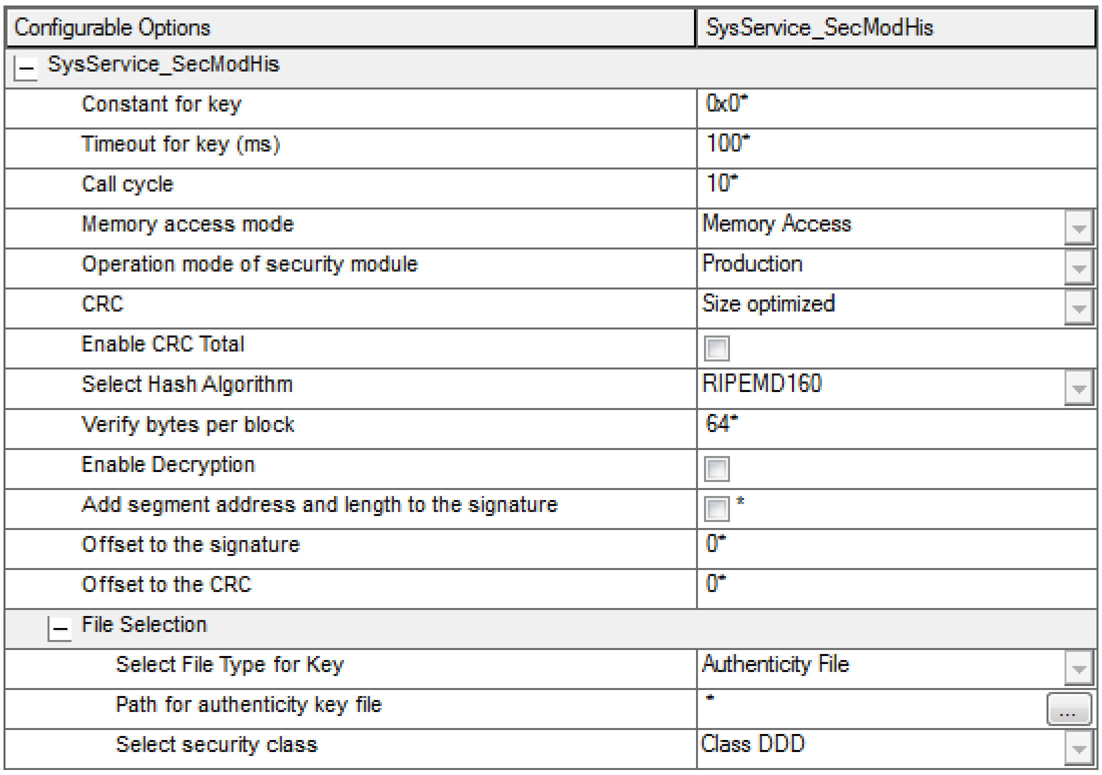
4 Configuration

4.1 GENy

We recommend configuring the Security Module with the configuration tool GENy. Depending on your order the tool can be downloaded from our FTP and installed.

4.2 Security Module Configuration

The features of the security component are configured in the 'SysService_SecModHis' component. The available features or pre-set features are shown on the right side of the window.



Configurable Options	SysService_SecModHis
– SysService_SecModHis	
Constant for key	0x0*
Timeout for key (ms)	100*
Call cycle	10*
Memory access mode	Memory Access
Operation mode of security module	Production
CRC	Size optimized
Enable CRC Total	☐
Select Hash Algorithm	RIPEMD160
Verify bytes per block	64*
Enable Decryption	☐
Add segment address and length to the signature	☐ *
Offset to the signature	0*
Offset to the CRC	0*
– File Selection	
Select File Type for Key	Authenticity File
Path for authenticity key file	*
Select security class	Class DDD

Figure 4-1 Security Component configuration

4.3 Configuration Parameters

Due to the OEM pre-configuration the following parameter may appear in the GENy configuration. Note that the default values may differ depending on the OEM.

Attribute Name	Default Values	Description
Constant for key	0x0	This is the key constant to adjust the authentication algorithm for a specific type of ECU.
Timeout for key	100 ms	This parameter is currently not used and is only provided for compatibility reasons to the HIS specification.

Attribute Name	Default Values	Description
		The timeout handling of the authorization is managed within the diagnostic layer, not in the security component according to OEM specification and cannot be adjusted.
Call cycle	10	Specifies the call cycle to the function SecTask(). Commonly SecTask() is not used. In most cases this parameter is only for HIS-conformance.
CRC	Size optimized	The CRC-module can run in two different modes: SIZE optimized and SPEED optimized. When ‘SPEED optimized’ is selected, a RAM table will be generated. If the checksum calculation shall also been used within the application, we strongly recommend to use “SIZE optimized”.
Enable CRC Total	True	The security component always calculates a CRCtotal for a logical block when enabled. Thus, the application is able to verify the values within the application.
Memory access mode	Memory access	The access mode of the security component is fixed to the memory access function. The setting is displayed for documentation purposes only.
Operation mode	Production	The mode of the security component is set to Production. The setting is displayed for documentation purposes only.
Enable CRC Total	True	The security component always calculates a CRCtotal for a logical block when enabled. Thus, the application is able to verify the values within the application.
Select Hash algorithm	SHA1	Select the hash function for the security classes C and CCC.
Verify bytes per block	64	Number of bytes that are reserved by the security module for verification. This value determines the number of bytes that are verified in one block operation.
Enable Decryption	False	Enables or disables the AES decryption functionality of the security module.
Enable Encryption	False	Enables or disables the AES encryption functionality of the security module.
Add segment address and length to the signature	True	For the security classes C and CCC, the address and length information of every segment is part of the signature calculation. This setting is usually given by the OEM.
Offset to the signature	0	Specifies the offset to the signature / CRC within the verification input data stream. If both
Offset to the CRC	0	

Attribute Name	Default Values	Description
		offsets are default (zero), the Security Module treats the first part of the verification input data stream as signature and the following data as checksum.
Select file type for key	Authenticity File	The key for the security classes C and CCC can be read and interpreted in two different formats: Authenticity key file format: A file that simply contains the data in the HIS/ASN.1-format INI-key file: An INI-file format that contains the key and other additional information The dialog below will change when the file type is changed.
Authenticity file mode		
Path for Authenticity key file	-	Select the key-file that contains the key information for the corresponding security class. The data is simply a one-line text file with a hex byte stream in the HIS ASN.1 file format.
Security class	DDD	Select the required security class for the Bootloader. Note that the selected key must contain the keys in the appropriate format. If not, GENy will generate an error during generation.
INI-Key file mode		
Path for ini key file	-	Specify the path to an INI-file that contains one or more key information for the security classes C, CCC or AAA. See chapter 4.4.2.
Read section from INI-file	-	This button must be pressed to let GENy read in the sections of the selected INI-file. Afterwards, you need to change to another component and then back to the SysService_SecModHis, to update the GUI.

Table 4-1 GENy configuration parameter

4.4 Key Files

The key files can be selected in two different modes:

- Authenticity file mode
- INI-Key file mode

The appearance of the dialog will differ depending on the selection. This affects only the key-file selection for the security classes C and CCC. The default configuration in the upper part of the configuration window is not affected and appears the same in both configuration options.

4.4.1 Authenticity file mode

If the authenticity file mode is selected, GENy expects the data in a simply one-line text file with a hex byte stream in a special format (pseudo-ASN.1) which has been specified in the HIS security module specification [2]. GENy as well as HexView require this format for data processing.

Security Class C:

Tag	Length description	Length (byte)	Value
'FF 59'	'XX' resp. '81 XX'	max. 34	Object with key components of a private key
'D3'	'XX' resp. '81' XX typically '14'	xx resp. 20	HMAC key

Table 4-2 Format of HMAC key

Example
FF5916D3145F1CBE397C4AF8956E26DC4DAED95DB25A14B429

Tag	Description
'FF 59'	Tag: Private key
'16'	Total length information
'D3'	Tag: HMAC key
'14'	Length of key data
'5F…29'	Key data

Table 4-3 Key structure of the example key

Security Class CCC:

Object with key components of a public key:

Tag	Length description	Length (byte)	Value
'7F 49'	'81 XX' '82 XX XX'	-	Object with key components of a public key
'81'	'81 80' resp. '82 01 00'	128 resp. 256	Modulus 'n'
'82'	XX	typically 2…3	Public (encryption) exponent 'e'

Table 4-4 Format of public key without additional information

Object with key components of a public and a private key:

Tag	Length description	Length (byte)	Value
‘FF 49’	’81 XX’ resp. ’82 XX XX’	-	Object with key components of a public and private key
‘81’	’81 80’ resp. ’82 01 00’	128 resp. 256	Modulus ‘n’
‘82’	XX	typically 2…3	Public (encryption) exponent ‘e’
‘91’	’81 80’ resp. ’82 01 00’	128 resp. 256	Private (decryption) exponent ‘d’

Table 4-5 Format of public key and private key

Example

FF49 82 010B 81 81 80 D7D9F92AF1E4DD67DABC9169E76F880D49CE9707BA95
CFDBE3C6F1D8AAEE2887F04F3F1447DBDF66F70AE282D8701F74D65281203413
F86E0FD3014FF3D80099C01FBEE2E4316EEB64A6412D285AC52010762BD79994
4A80333A74B4B59164773291F26A162680E61A764764DC04FC86C133CCF008A09
AF601434C4025C15507 82 03 010001 91 81 80 4AD2D69ADA6B598CACB87D37D
6F0449B46A0849E3B41BF621C1D54ACA4E415CCB9B2DECCEAF3FF07344FE3A1
33E1FA25883DE561026C81CBB55491B9E36F571551DA6B2D1FF42A03BD5F5DDF
2FCA2F3BC3EF9F409A6B9B5E91C80811AB80B294A0A3ED9CA122255EF5C0CEC
DEBD2652A613B2DB6AFA588D71B42005528FAB231

Tag	Description
‘FF 49’	Tag: Private key
‘82’	2 byte length information (length >= 0x80)
’01 0B’	Total length
‘81’	Tag: Modulus ‘n’
‘81’	1 byte length information (length >= 0x80)
‘80’	Length of modulus
‘D7 … 07’	Modulus ‘n’
‘82’	Tag: Public exponent ‘e’
‘03’	Length of exponent (length < 0x80; no additional length information)
’01 00 01’	Public exponent ‘e’
‘91’	Tag: Private exponent ‘d’
‘81’	1 byte length information (length >= 0x80)
‘80’	Length of exponent
‘4A … 31’	Private exponent ‘d’

Table 4-6 Key structure of the example key

4.4.2 INI-Key file mode

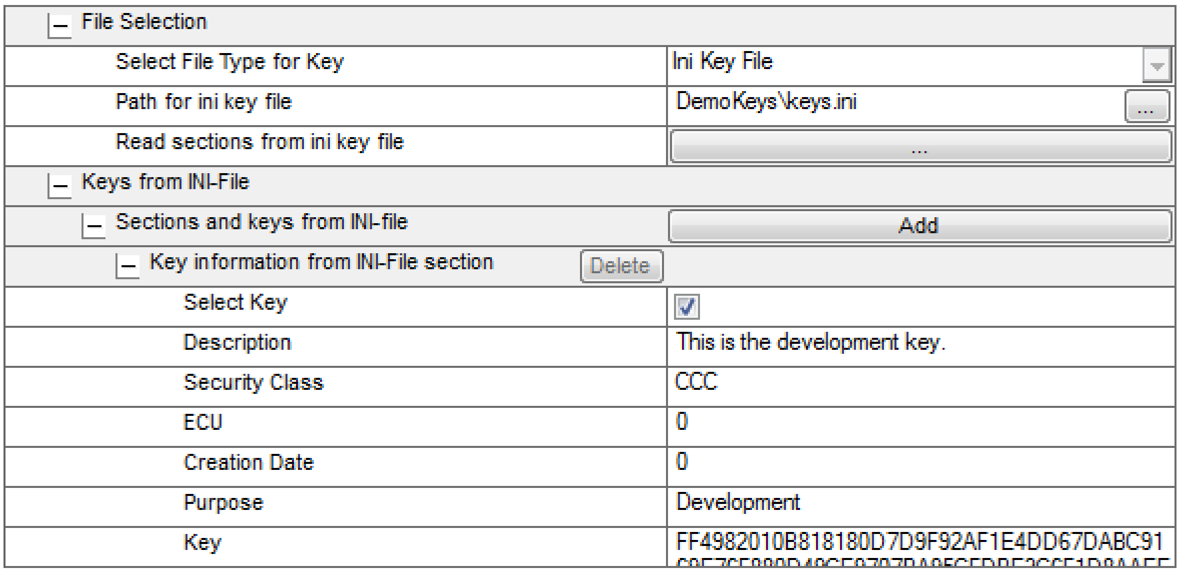
The INI-Key mode is a proper way to manage the keys. The key is also based on the ASN.1 format (see 4.4.1) but this file can be used to provide additional informations and can include multiple keys.

Parameter	Description
DESC	Description text which is showed in GENy.
VALUE (C) or KEY-PRIV (CCC)	The key in ASN.1 format.
ECU	ECU identifier.
CREATION-DATE	Timestamp of the creation of the key
PURPOSE	Describes the purpose of the key being used. E.g. production, development …
SECURITY-CLASS	Information about the security class.

Table 4-7 INI-Key file parameter

An example can be seen below:

```
[RSAKey01]

DESC = 'This is the development key.'

KEY-PRIV =

'FF4982010B818180D7D9F92AF1E4DD67DABC9169E76F880D49CE9707BA95CFDBE3C6F1D8AAEE288
7F04F3F1447DBDF66F70AE282D8701F74D65281203413F86E0FD3014FF3D80099C01FBEE2E4316EE
B64A6412D285AC52010762BD799944A80333A74B4B59164773291F26A162680E61A764764DC04FC8
6C133CCF008A09AF601434C4025C15507820301000191818 04AD2D69ADA6B598CACB87D37D6F0449
B46A0849E3B41BF621C1D54ACA4E415CCB9B2DECCEAF3FF07344FE3A133E1FA25883DE561026C81C
BB55491B9E36F571551DA6B2D1FF42A03BD5F5DDF2FCA2F3BC3EF9F409A6B9B5E91C80811AB80B29
4A0A3ED9CA122255EF5C0CECDEBD2652A613B2DB6AFA588D71B42005528FAB231'

ECU = My ECU

CREATION-DATE = 2006-04-13T12:02:09+02:00

PURPOSE = Development

SECURITY-CLASS = CCC
```



File Selection	
Select File Type for Key	Ini Key File
Path for ini key file	DemoKeys\keys.ini
Read sections from ini key file	...
Keys from INI-File	
Sections and keys from INI-file	Add
Key information from INI-File section	Delete
Select Key	✓
Description	This is the development key.
Security Class	CCC
ECU	0
Creation Date	0
Purpose	Development
Key	FF4982010B818180D7D9F92AF1E4DD67DABC91

Figure 4-2 Key file selection in the INI-file mode

4.5 Running the Generator

When you have finished your configuration selections, you need to run the generator to produce the files needed to compile the FBL. To generate the files, click on the lightning-bolt button on the toolbar, or select “Generate System” from the Configuration menu. The following table describes the contents of the generated files:

File Name	Description
Secm_cfg.h	The configuration file of the security module. For the most part the file defines switches in the form of SEC_ENABLE_<feature> or SEC_DISABLE_<feature> according to the settings in the dialog.
Secmpar.c, Secmpar.h	This file contains the secret key if security class C or the public key (on security class CCC) has been selected. The file is generated from the contents of the key given in the authenticity key file definition. You must compile and link these files with your security component.
v_cfg.h	Contains macro definitions to your hardware.
v_inc.h	Includes the generated headers, so that they may be obtained from a single source.
v_par.c, v_par.h	Contains information about your license. The files are not necessary for a successful compile/link procedure with the security component.

Table 4-8 Generated files

5 Preparation of the Application

Before a download can be made preparing the download file is a necessary step. For bootloaders with logical block tables, the address space used by one single file has to match information of a single block stored in the logical block table. It is possible to download multi region files, but the region count is limited by the Bootloader.

Each download file requires a signature or resp. a checksum file, which is created out of the file contents. A signature file is a byte stream in hexadecimal format with comma separation (e.g. 0x12, 0x34, 0x45). The number of bytes within a signature file and its creation depend on the security class that is used for the download.

The following three different security classes are distinguished (see chapter 1):

> Security Class DDD

> Security Class C

> Security Class CCC

For Security class DDD, the checksum file contains 4 bytes, which reflects the CRC-32 checksum from the download file or 2 bytes in case of CRC-16.

For Security Class C, the signature file contains 20 byte for SHA-1 or RIPEMD-160. The signature of SHA-256 has a length of 32 bytes. The security class C requires a secret key[2]. The key information for security class C and CCC are required by a tool (HexView) to create this signature from the download file.

For class C the same secret key information is needed by the Bootloader for the verification process after a successful flash download. Thus, the contents of the secret key must be stored in the generation tool. This is done by referencing the key within GENy tool (see chapter 4.4). The tool will then create the file ‘Secmpar.c’. This file must be compiled and linked with the Bootloader.

Store the secret key carefully away and do not provide this information to others, as this is the secret for further updates of your ECU software.

The signature file for the Security Class CCC is depending on the key length of the chosen RSA (e.g.: 128 bytes for RSA-1024). It uses a public and a private key, similar to PGP. The private key is necessary to create the signature files, whereas the public key is used for the verification process within the Bootloader. It is necessary to use the private key to create the signature files.

The key files for Security Class CCC are also stored in the ASN.1 format. The creation of the verification data through GENy and the signature files through HexView are similar to the Security Class C.

[2] The keys for series production are usually provided by the OEM. However, to support the development process, some sample keys are part of the Bootloader delivery (see folder `.\Demo\DemoKeys` in the delivery). They are NOT INTENDED for series production, but only for development purposes.

To support you in the development process, the Bootloader product comes along with some tools to support you to create these checksum and signature files.

5.1 HexView

HexView is a software tool from Vector Informatik. It can be used to change and show the contents of a binary file, an Intel Hex file or a Motorola S-Record file. HexView has an extensive list of features. These features are important for the application:

> HexView can be used to fill the gaps of your application hex file and align the data to the memory segment size.

> HexView can be used to calculate the checksum or signature of your application.

>

Note
For detailed information about HexView please refer to the 'HexView Reference Manual' [3].

5.1.1 Fill and align the hex file with HexView

These are the command line arguments to fill and align a hex file:

> `hexview.exe demo.hex -S -e:error.txt -ad:0x08 -fa -xi -o demo_out.hex`
The hex file regions are aligned to 8 bytes addresses, the gaps are filled and the output is written into the file `demo_out.hex`

5.1.2 Create a checksum / signature file with HexView

These are the command line arguments for the signature and checksum calculation.

> Security Class DDD:
`hexview.exe demo.hex -S -e:error.txt -cs9:demo.crc`
The 4-byte CRC-32 value is directly written into the file demo.crc

> Security Class C:
`hexview.exe demo.hex -S -e:error.txt -dp5:..\key.txt`
by default, HexView writes the signature into the file SignDAL_sha1.txt. The file is located where the hex-file has been loaded from.

> Security Class CCC:
`hexview.exe demo.hex -S -e:error.txt -dp10:..\key.txt`
by default, HexView writes the signature into the file SignDAL_sha1.rsa. The file is located where the hex-file has been loaded from.

5.1.3 Create an AES128 Encrypted file with HexView

The AES128 encryption standard operates on blocks of 16 bytes. As the module operates in streaming mode, the file to be encrypted has to be padded so that all contiguous address ranges consist of an integer multiple of 16 bytes. The padding is performed according to PKCS#5.

Additionally, AES128 with CBC mode requires an initialization vector. It can either be used explicit by prefixing on each segment of the download file or as an implicit initialization vector.

5.2 Signature Generation

This sub-chapter covers the off-board signature generation without using HexView. The signature generation is split into two main parts:

> Hash calculation

> Signing operation

5.2.1 Hash calculation

The hash is used to verify the integrity of the signed message. If the hash calculated on the received data matches the one from the signature one can be confident the message contents are unaltered.

Varying hash algorithms can be used to generate a signature, with SHA-1 and RIPEMD-160 being the most commonly used for the Bootloader use-case. The exact choice is defined by the OEM specification.

The hash value is calculated as following:

- Initialize hash (algorithm dependent)

  Depending on the utilized security class the initialization includes some additional steps. Those are explained in the respective subchapter of the signing operation.

- Update the hash value with the following information.

  In case the input data is split into multiple segments the next steps have to be performed for each segment one after another

  - Address information, padded to 4 byte (big-endian order)
    Depending on the OEM requirements this could be a physical or logical address (e.g. block index).
  - Length information, padded to 4 byte (big-endian order)
    Usually this represents the length of the original data as it is written to memory, especially before being compressed and encrypted.
  - Actual segment data, usually processed in smaller chunks (e.g. 64 byte)
    Depending on the OEM requirements this is either the original (uncompressed and unencrypted) data as written to memory or the processed (compressed and/or encrypted) data as transferred over the bus.

- Finalize hash (algorithm dependent)

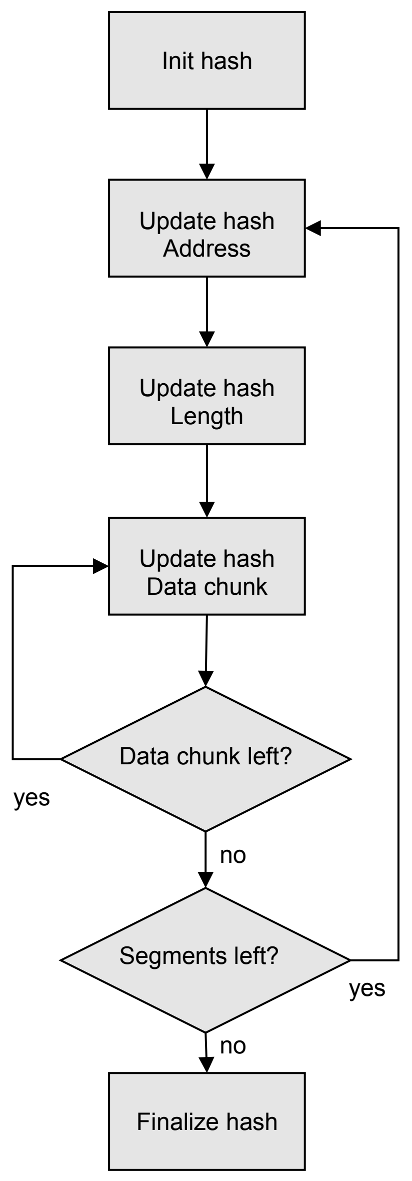


Figure 5-1 Hash calculation

5.2.2 Signing operation

The actual signature is generated by signing the hash, calculated in the previous step, using either the secret key (symmetrical algorithm) or the private key of the signer (asymmetrical algorithm). This assures the authenticity of the signed message as only an authority with knowledge of the respective key is able to create a signature matching the data.

5.2.2.1 Security class C

Class C implements a HMAC (Hash-based Message Authentication Code) according to RFC 2104 [5] (HMAC, Keyed-Hashing for Message Authentication) and ANSI X9.71 [6] (Key Hash Message Authentication Code).

It uses a secret key, typically of a length between the hash length (typically 20 byte) and the hash chunk length (typically 64 byte). But larger keys can be used by first hashing the key itself.

To generate a class C signature the following operations are performed:

- Prepare the secret key
  - If the input key is longer than the hash chunk length (typically 64 byte) hash the key value and use this value
  - Otherwise use the key value directly
- Pad key value with zeroes to fill it up to a length of the hash chunk size (typically 64 byte)
- XOR each byte of the padded key with 0x36 (“ipad”)
- Initialize hash algorithm
- Update hash with “ipad” value
- Perform hash updates on input data
- Finalize hash calculation (“inner hash”)
- XOR each byte of the padded key with 0x5C (“opad”)
- Initialize hash algorithm
- Update hash with “opad” value
- Update hash with “inner hash”
- Finalize hash calculation (“outer hash”)

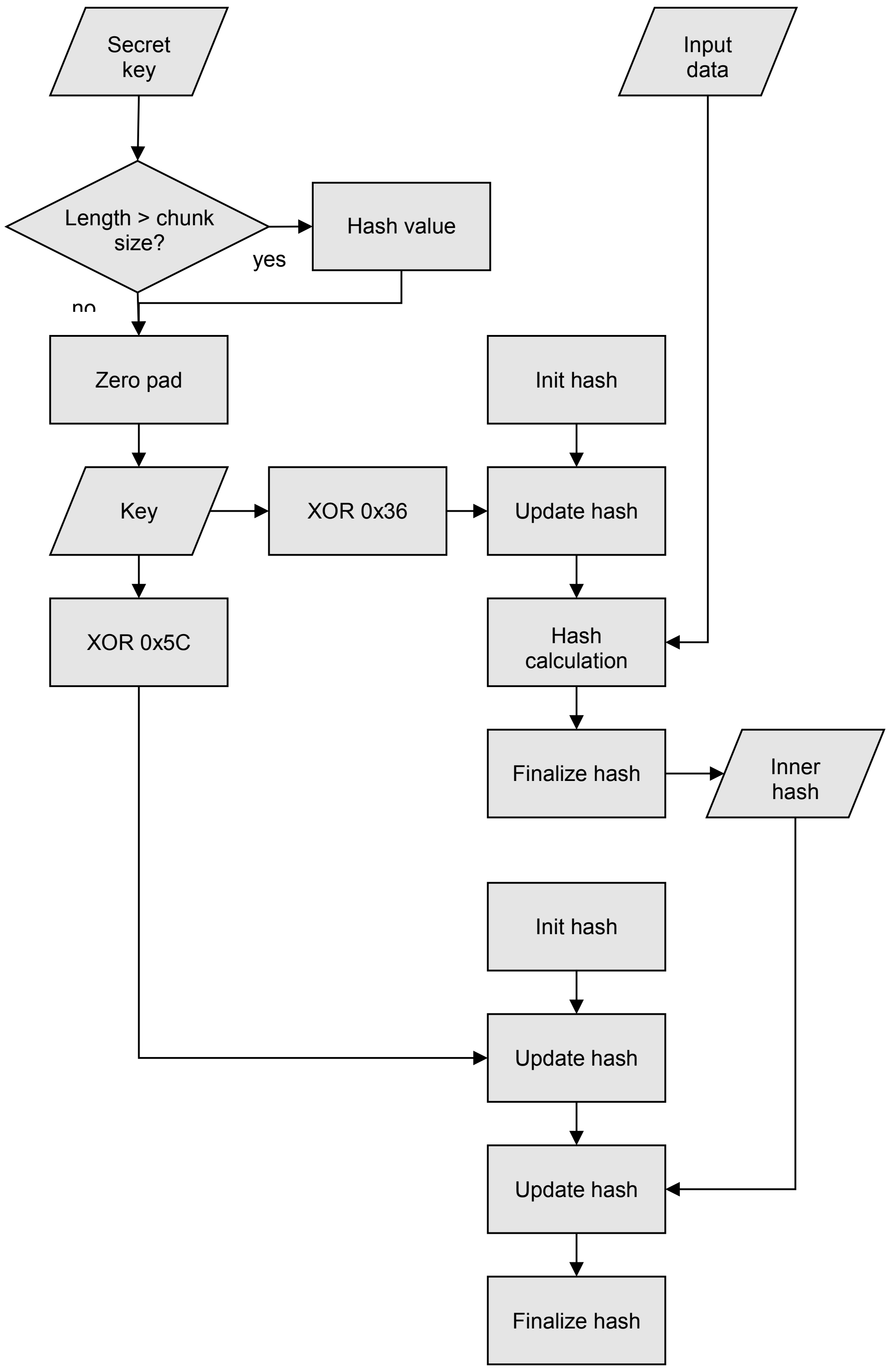


Figure 5-2 Security class C signature

5.2.2.2 Security class CCC

Class CCC is implements an RSA based signature according to PKCS #1 ([4] RSASSA-PKCS1-v1_5, Signature Schemes with Appendix). It uses a private RSA key pair to sign the hash value of the input data and the corresponding public RSA key pair to verify the signature afterwards.

The calculation of an RSA signature requires the following steps:

> Calculate the hash value of the input data (without any further modifications)
> Prepend the following header information to the hash value, forming the encoded message
  > 0x00 0x01
  > Fill up with 0xFF to match RSA encryption length (e.g. total of 128 byte with 1024 bit RSA):
    Number of fill bytes = signature length – digest length – hash length - 3
  > 0x00
  > Algorithm information digest (see the information below)
    0x00 0x01 0xFF … 0xFF 0x00 <digest> <hash>
> Encrypt encoded message using private RSA key pair

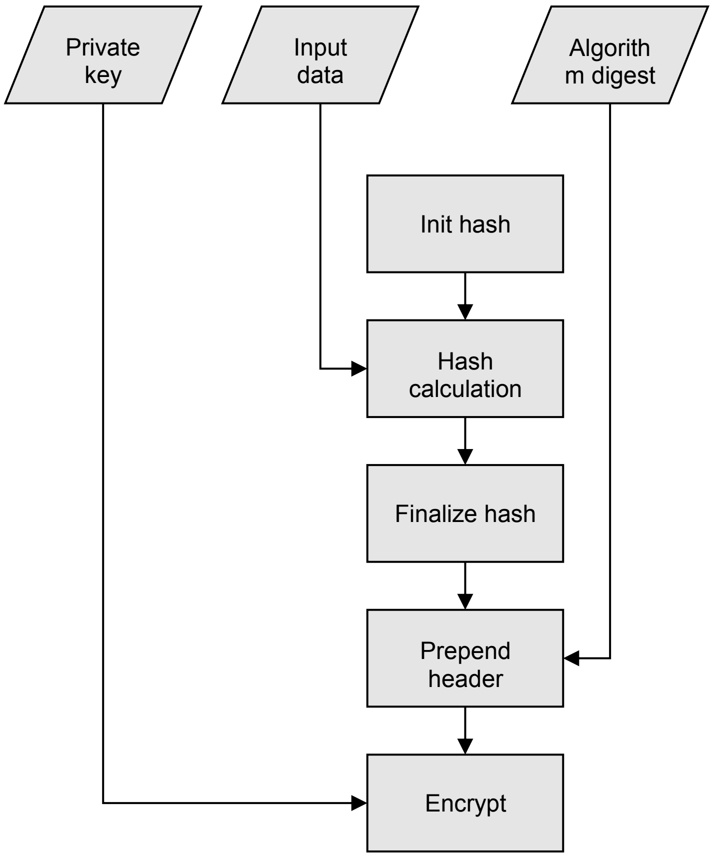


Figure 5-3 Security class CCC signature

Algorithm information digest

The utilized hash algorithm is identified through the algorithm information digest. Table 5-1 lists some commonly used digest values.

Hash algorithm	Digest
SHA-1	(0x)30 21 30 09 06 05 2b 0e 03 02 1a 05 00 04 14
SHA-256	(0x)30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20
RIPEMD-160	(0x)30 21 30 09 06 05 2b 24 03 02 01 05 00 04 14

Table 5-1 Algorithm information digest

5.3 MkPsKeys

MkPsKeys stands for MaKePSeudoKEYS and can be used to create further keys for your validation. The tool is a command line tool to create the secret key for security class and the private key for the security class CCC. The key values are pseudo-random and especially the RSA-keys are not fully approved since there is only a simple verification.

Besides the key information, created in the ASN-like (a further description can be found in 4.4), the tool can create the output into a C file. This file contains the secret key for the security class C resp. public key information for Security Class CCC.

MkPsKey provides the following options:

-hmac	Create an HMAC key for security class. If this option is omitted, an RSA key for security class CCC will be generated.
-d	Add a timestamp to the key.
-o filename	Overwrites the default output name rsakeys.txt.
-v file.c	Overwrites the default output name SecMpar`.c` for the C file.
-n keyname	Specifies a keyname. This name will be inserted into the byte stream.
-?	Show help.

Example: Creating an HMAC key for Security Class C

Mkpskeys –hmac –o hmackey.txt

This will generate the file SecMpar.c and the HMAC key for security class C into the file hmackey.txt. This file can be compiled and linked with the Security Module. However we recommend using GENy to generate the SecMpar.c.

Key selection
In the sub-chapter 4.4.1 you can obtain supplemental information about the key selection in GENy.

6 Functional Description

This chapter describes the security modules, their variants and provides implementation samples for some common use-cases.

6.1 Verification Module

In the Bootloader case the security component is used to verify one or more sections of data. Depending on the security class, it consists of more or less complex mathematical routines that create a set of data with fixed length out of these sections of data. This fixed length data is called the signature or checksum.

It is specific to the mathematical routines that the signature changes drastically even though just a small portion or even a bit in the data stream has changed. It is also specific, that a key is necessary to create this signature.

6.1.1 Variants

> RSA and SHA-1 signature verification

> Internal/external workspace

6.1.2 Description of the verification process

As mentioned above, the signature is created on a set of data. This is usually a data stream that shall be downloaded and programmed into the memory of an ECU.

For this purpose, the flash memory is partitioned into logical blocks. Each logical block can store a program or other code that consists of one or more sections.

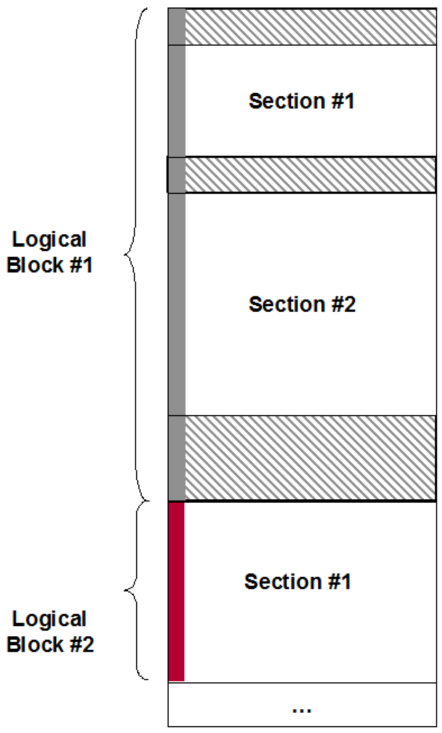


Figure 6-1 Logical Blocks

The signature is created over the segments of a logical block. Thus, for the module of logical block #1, there are two segments available.

Usually the signature is not only generated on the data itself. The address and length information of each segment is also added to the creation of the signature.

The segments reside in physical address regions. However, it can be necessary, that the address information for the download does not correspond to the physical addresses, because of necessary address translations during the download, e.g. for banked systems or re-locatable modules. Here, logical or virtual addresses are used instead of physical addresses. Note that the signature will be based only on the logical or virtual address information, because only this information is available when creating the signature. However, when the data shall be read from the memory, the physical address information is required.

As a result, the SecM_Verification process requires both the address information that was given by the download and the address information where the data resides in memory. The logical or virtual address information is stored in the transferredAddress element of the segment definition.

6.1.3 Usage

The following sample demonstrates how to use the verification module in monolithic use for verification on two sections and a logical block.

```
/* Function to read the data from memory */
SecM_SizeType ReadMemoryFct(SecM_AddrType address, vuint8 *buffer, SecM_SizeType length)
{
   memcpy(buffer, (void *)address, length);
}

/* Function is called frequently during the verification. */
/* At least in a shorter time distance than 1 ms.         */
void WatchdogFct( void )
{
   return 0u;
}

/* Verification parameters */
SecM_VerifyParamType verifyParam;

/* Signature to be verified */
vuint8 signature[128u];

/* List of memory segments */
FL_SegmentListType segmentList;

/* Setup section information of section #1 */
segmentList.segmentInfo[0].targetAddress      = FLASH_SECTION1_MEMORY_ADDRESS;
segmentList.segmentInfo[0].transferredAddress = FLASH_SECTION1_DOWNLOAD_ADDRESS;
segmentList.segmentInfo[0].length             = FLASH_SECTION1_LENGTH;

/* Setup section information of section #2 */
segmentList.segmentInfo[1].targetAddress      = FLASH_SECTION2_MEMORY_ADDRESS;
segmentList.segmentInfo[1].transferredAddress = FLASH_SECTION2_DOWNLOAD_ADDRESS;
segmentList.segmentInfo[1].length             = FLASH_SECTION2_LENGTH;

/* Number of memory segments */
segmentList.nrOfSegments                      = 2u;

/* Initialize verification structure */
/* Start address of full block */
verifyParam.blockStartAddress = FLASHREGION_LBT0_ADDRESS;
/* Length of full block */
verifyParam.blockLength       = FLASHREGION_LBT0_LENGTH;
/* Pass segment list */
verifyParam.segmentList       = &segmentList;
/* Function to read memory at given address */
verifyParam.readMemory        = ReadMemoryFct;
/* Cyclically called function */
verifyParam.wdTriggerFct      = WatchdogFct;
/* Pass signature */
verifyParam.verificationData  = (SecM_VerifyDataType)signature;
/* Use internal workspace */
verifyParam.workspace.size    = SEC_DEFAULT_WORKSPACE_SIZE;

/* Perform signature verification */
if ( (SECM_OK == SecM_InitVerification(V_NULL))
  && (SECM_VER_OK == SecM_Verification(&verifyParam))
  && (SECM_OK == SecM_DeinitVerification(V_NULL)) )
{
   /* Verification successful */
}
```

The example below shows the verification with the streaming use-case.

```
/* Verification parameters */
SecM_SignatureParamType sigParam;
/* Buffer for data to be verified */
vuint8 inputData[0x100u];
/* Control variable */
```

```
SecM_LengthType inputLength;

/* Read and hash input data */
SecM_AddrType inputAddress   = 0x10000u;
SecM_SizeType inputRemainder = 0x1000u;

/* Cyclically called function */
sigParam.wdTriggerFct = WatchdogFct;
/* Pass input buffer (not relevant for initialization) */
sigParam.sigSourceBuffer = inputData;

/* Initialize signature verification */
sigParam.sigState = SEC_HASH_INIT;
sigParam.sigByteCount = 0u;

if (SECM_VER_OK != SecM_VerifySignature(&sigParam))
{
   /* Initialization failed */
}

/* Proceed computation */
while (inputRemainder > 0u)
{
   /* Truncate input length */
   inputLength = sizeof(inputData);
   if (inputLength > inputRemainder)
   {
      inputLength = (SecM_LengthType)inputRemainder;
   }

   /* Read data into temporary buffer */
   sigParam.sigByteCount = (SecM_LengthType)ReadMemoryFct(inputAddress, inputData,
inputLength);

   if (SECM_VER_OK != SecM_VerifySignature(&sigParam))
   {
      /* Computation has failed */
      break;
   }

   /* Update control variables */
   inputRemainder -= inputLength;
   inputAddress   += inputLength;
}

/* Finalize hash calculation */
sigParam.sigState        = SEC_HASH_FINALIZE;
sigParam.sigByteCount    = 0u;

if (SECM_VER_OK == SecM_VerifySignature(&sigParam))
{
   /* Verify signature */
   sigParam.sigState           = SEC_SIG_VERIFY;
   /* Pass signature */
   sigParam.sigSourceBuffer    = signature;
   sigParam.sigByteCount       = sizeof(signature);

   if (SECM_VER_OK == SecM_VerifySignature(&sigParam))
   {
      /* Verification successful */
   }
}
```

6.2 CRC Module

The CRC is an error-detecting code which can be used to build a checksum over a defined segment. If this module was ordered a 16 or 32 Bit CRC implementation is included in the delivery. The SecM_ComputeCRC function can be called multiple times. The so called streaming approach can be used for the verification of a data stream. It can be necessary to verify the integrity of a logical block.

6.2.1 Variants

> CRC-16 (CCITT) or CRC-32 (IEEE 802.3)

> CRCTotal

> Speed/Size optimized

6.2.2 CRCTotal

To avoid storing the whole segment information with the resulting signature, the security module calculates a CRCTotal. This CRCTotal is generated internally by the security component during the verification process by the function SecM_Verification(). It is only necessary to pass the blockStartAddress and the blockLength of the logical block. The results will be placed into the CRCtotal field by the security module. If the CRCTotal option is activated in GENy it is always calculated, regardless of the security class.

Note
The CRCWritten is the CRC-value as the signature value calculated over the segments for the security class DDD.

The CRCwritten is not available for the Security Class C or CCC.

6.2.3 Speed / size optimization

In the GENy configuration the CRC module can be switched between two different modes: Speed or size optimized. When ‘SPEED optimized’ is chosen, a 1024 byte look-up table will be generated in case of CRC-32 or 256 byte for CRC-16.

Caution
If the checksum calculation shall also be used within the application, we strongly recommend to use ‘SIZE optimized’

6.2.4 Operation mode flags

These operation mode flags are valid and can be set by using the ‘crcState’.

> SEC_CRC_INIT

Sets the initial CRC value and initialize the look-up table.

> SEC_CRC_COMPUTE

Update the CRC calculation using the provided data.

> SEC_CRC_FINALIZE

Optional: Complete the CRC value

6.2.5 Usage

The following sample shows the usage of the CRC calculation. The usage of the module is similar to the stream verification.

The initialization sets up the environment:

```
/* CRC parameters */
SecM_CRCParamType crcParam;
/* Buffer for data to be verified */
vuint8            inputData[0x100u];

/* Cyclically called function */
crcParam.wdTriggerFct      = WatchdogFct;
/* Pass input buffer (not relevant for initialization) */
crcParam.crcSourceBuffer   = inputData;

/* Initialize signature verification */
crcParam.crcState          = SEC_CRC_INIT;
crcParam.crcByteCount      = 0u;


if (SECM_OK == SecM_ComputeCRC(&crcParam))
{
   ...
}
```

In the computation phase the processing will be done. Addresses and length information are typically not included in the CRC. The CRC Module doesn’t have access to the memory in the function. Therefore a function to read from the memory is needed (in the example represented as ‘ReadMemoryFct’).

```
/* Control variable */
SecM_AddrType      inputAddress    = 0x10000u;
SecM_SizeType      inputRemainder = 0x1000u;
SecM_LengthType    inputLength;

/* Read input data and update checksum */
crcParam.crcState = SEC_HASH_COMPUTE;
while (inputRemainder > 0u)
{
   /* Truncate input length */
   inputLength = sizeof(inputData);
   if (inputLength > inputRemainder)
   {
      inputLength = (SecM_LengthType)inputRemainder;
   }

   /* Read data into temporary buffer */
   crcParam.crcByteCount = (SecM_LengthType)ReadMemoryFct(
      inputAddress, inputData, inputLength);

   if (SECM_VER_OK != SecM_ComputeCRC(&crcParam))
   {
      break;
   }

   /* Update control variables */
   inputRemainder -= inputLength;
   inputAddress   += inputLength;
}
```

During the finalization no additional data are required. The checksum is provided in the member currentCRC.

```
/* Checksum to be verified */
```

```
SecM_CRCType       checksum;

/* Finalize CRC calculation */
crcParam.crcState       = SECM_CRC_FINALIZE;
crcParam.crcByteCount   = 0u;

if (SECM_VER_OK == SecM_ComputeCRC(&crcParam))
{
   /* Compare checksum */
   if (crcParam.currentCRC == checksum)
   {
      /* Checksum comparison successful */
   }
}
```

6.3 Encryption / Decryption Module

The Encryption / Decryption Module provide encryption and decryption of data using AES. The current implementation supports AES encryption / decryption with CBC and PKCS#5 padding mode. The key length depends on your order and can be 128, 192 or 256 bit.

6.3.1 Variants

> AES 128/192/256 bit encryption / decryption in CBC mode

> With or without initialization vector

> Internal/external workspace usage

6.3.2 Using an explicit initialization vector

The first 16 bytes (block size) of data will be interpreted as the initialization vector, when the operation mode flag SEC_DECRYPTION_MODE_AES_<xxx>_PKCS_CBC_IV is set.

By using the encryption API ‘SecM_Encryption’ the IV will be used for the AES CBC encryption and will be copied to the output buffer.

Figure 6-2 describes the usage of the explicit initialization vector with the encryption use case. The input buffer consists of the IV (one block size) and the plaintext data.

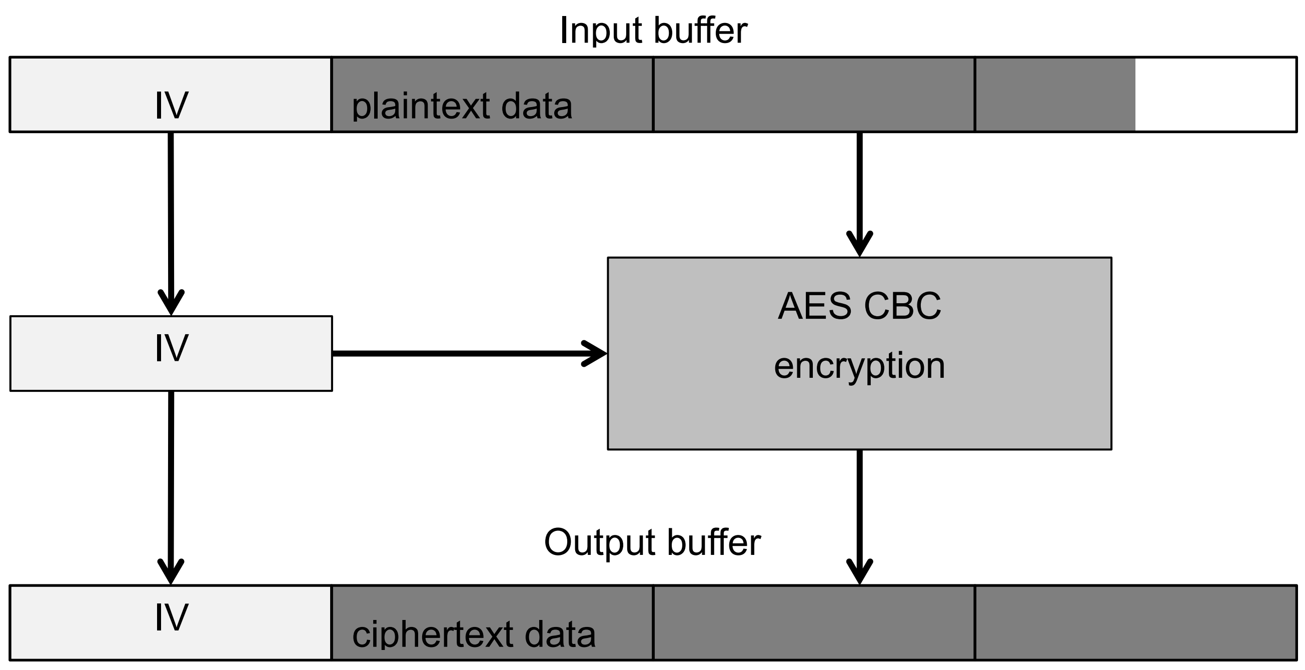


Figure 6-2 Explicit usage of an initialization vector with the encryption API

By the usage of an explicit initialization vector, the first block will be used as the initialization vector and will be copied to the output buffer. Due to the padding mode PKCS#5 the output data will be padded to the next multiple of the block size. If the plaintext is aligned to the 16 byte block size an additional padding block will be added.

Using an explicit initialization vector with the decryption API is shown in the Figure 6-3.

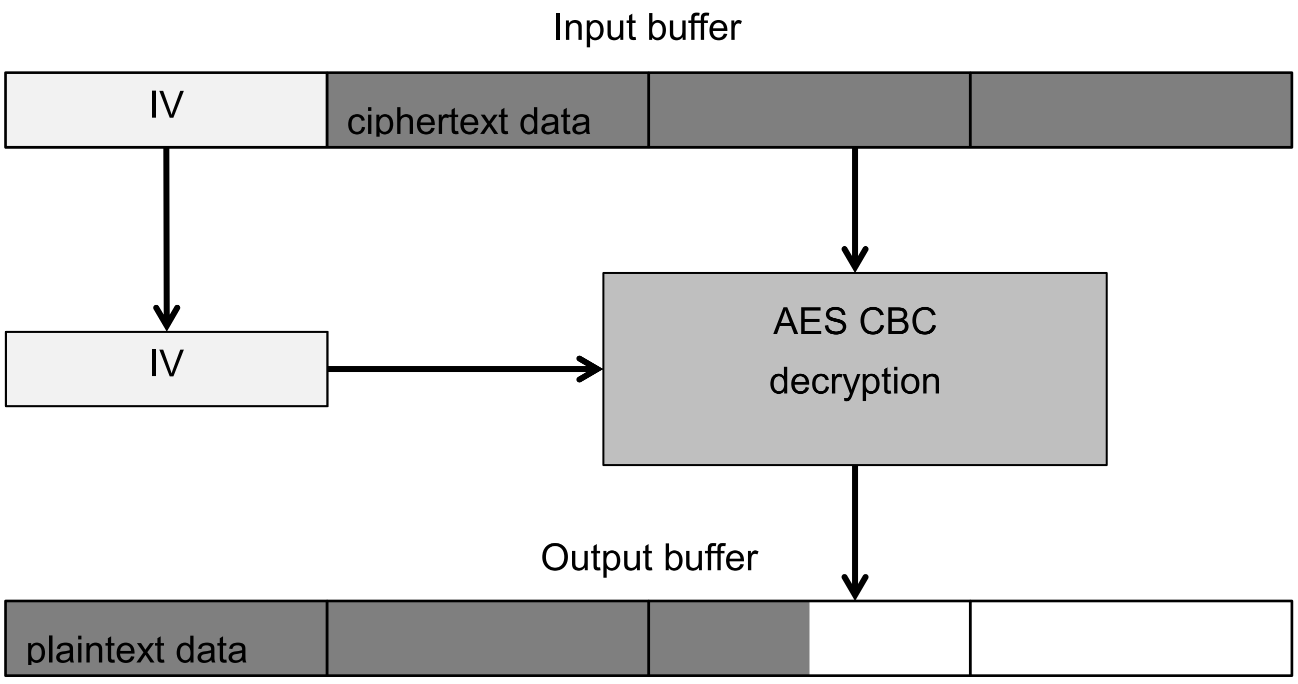


Figure 6-3 Explicit usage of an initialization vector with the decryption API

Similar to the encryption API the decryption mode will treat the first 16 bytes as the IV. This has the effect that no result will be written to the output buffer when the first round provides an input buffer size of less than 32 bytes. Hence the output buffer data will be smaller as the input buffer data due to the removed padding from the result and the obsolete IV data.

6.3.3 Internal / external Key

An external key can be passed to the decryption / encryption in case enabled in the configuration. When the provided key.size is equal to zero and the usage of internal key is enabled, the internal key is used for encryption / decryption.

The internal key symbol is SecM_AES<xxx>key (e.g.: SecM_AES128key) and will be located in configuration file Secmpar.c.

6.3.4 Operation mode flags

The operation mode is set by the member 'mode' of the function parameter 'pEncParam' respectively 'pDecParam'.

On these operation modes the CBC initialization vector will be filled with zero. No additional data have to be passed with the input data buffer:

> SEC_DECRYPTION_MODE_AES_128_PKCS_CBC

> SEC_DECRYPTION_MODE_AES_192_PKCS_CBC

> SEC_DECRYPTION_MODE_AES_256_PKCS_CBC

Using an explicit initialization vector requires one of these flags:

> SEC_DECRYPTION_MODE_AES_128_PKCS_CBC_IV

- SEC_DECRYPTION_MODE_AES_192_PKCS_CBC_IV
- SEC_DECRYPTION_MODE_AES_256_PKCS_CBC_IV

6.3.5 Operation mode modifiers

The operation mode modifier triggers the initialization or the finalization of the AES operation. These modifiers can be combined with the operation mode flags by an OR-operation.

- SEC_DECRYPTION_MODE_INIT

This flag will initialize the workspace, set up the key and copy the IV if necessary.

- SEC_DECRYPTION_MODE_FINALIZE

Setting the flag will process the last input data as well as applying the padding.

6.3.6 Usage

This sample shows how to encrypt a cipher text and decrypt in streaming mode. The usage of the decryption is similar.

```
   /* Variables for the encryption / decryption loop */
   vuint8 retVal = 0;
   vuint8 blocksize;
   vsint8 remainingInput, remainingOutput;
   vuint8 inputOffset, outputOffset;

   /* Variables for encryption / decryption */
   SecM_DecInputParamType pInBlock;
   SecM_DecOutputParamType pOutBlock;
   SecM_DecParamType pEncParam;

   SecM_DecInputParamType pDecInBlock;
   SecM_DecOutputParamType pDecOutBlock;
   SecM_DecParamType pDecParam;

   SecM_SymKeyType key;

   /* Plaintext buffer */
   vuint8 plaintext[34] = {0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99,
0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff, 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88,
0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff, 0xaa, 0xbb};

   /* Ciphertext buffer */
   vuint8 ciphertext[48] = {0};

   /* Plaintext output buffer (decryption) */
   vuint8 plaintext_out[48] = {0};
   /* Key data */
   vuint8 keydata[16] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

   /* Set the encryption buffer */
   pInBlock.DataBuffer = plaintext;
   pOutBlock.DataBuffer = ciphertext;

   /* Set encryption parameter */
   pEncParam.key.data = keydata;
   pEncParam.key.size = 16;
   pEncParam.wdTriggerFct = (FL_WDTriggerFctType)V_NULL;

   /* Set the decryption buffer */
   pDecInBlock.DataBuffer = ciphertext;
   pDecOutBlock.DataBuffer = plaintext_out;

   /* Set decryption parameter */
   pDecParam.key = pEncParam.key;
   pDecParam.wdTriggerFct = pEncParam.wdTriggerFct;


   /* Encryption sample */
   blocksize = 16;                       /* Bytes to process per call */
   remainingInput  = sizeof(plaintext);  /* Remaining input byte counter */
   remainingOutput = sizeof(ciphertext); /* Remaining output byte counter */

   /* Pre-set init mode */
   pEncParam.mode = SEC_DECRYPTION_MODE_AES_128_PKCS_CBC | SEC_DECRYPTION_MODE_INIT;

   while ( ( remainingInput >= 0 ) && ( retVal == 0 ) )
   {
      if ( remainingInput >= blocksize )
      {
         /* Normal round */
         pInBlock.Length = blocksize;
      }
      else if ( remainingInput == 0 )
      {
         /* Last round */
         pInBlock.Length = 0;
```

```
        pEncParam.mode |= SEC_DECRYPTION_MODE_FINALIZE;
        remainingInput--; /* leave the loop after this round */
     }
     else
     {
        /* Before last round */
        pInBlock.Length = remainingInput;
     }
     pOutBlock.Length = remainingOutput;

     /* Set input data */
     pInBlock.DataBuffer  = &plaintext[sizeof(plaintext)-remainingInput];
     /* Set output data */
     pOutBlock.DataBuffer = &ciphertext[sizeof(ciphertext)-remainingOutput];

     /* Perform operation */
     retVal |= SecM_Encryption( &pInBlock, &pOutBlock , &pEncParam );

     pEncParam.mode  = SEC_DECRYPTION_MODE_AES_128_PKCS_CBC; /* Set normal mode */
     remainingInput  -= pInBlock.Length;        /* Update remaining input bytes */
     remainingOutput -= pOutBlock.Length;       /* Update remaining output bytes */
  }

  /* Decryption sample */
  remainingInput  = sizeof(ciphertext);    /* Remaining input byte counter */
  remainingOutput = sizeof(plaintext_out); /* Remaining output byte counter */

  /* Pre-set init mode */
  pDecParam.mode = SEC_DECRYPTION_MODE_AES_128_PKCS_CBC | SEC_DECRYPTION_MODE_INIT;

  while ( ( remainingInput >= 0 ) && ( retVal == 0 ) )
  {
     if ( remainingInput >= blocksize )
     {
        /* Normal round */
        pDecInBlock.Length = blocksize;
     }
     else if ( remainingInput == 0 )
     {
        /* Last round */
        pDecInBlock.Length = 0;
        pDecParam.mode |= SEC_DECRYPTION_MODE_FINALIZE;
```

```
         remainingInput--; /* leave the loop after this round */
      }
      else
      {
         /* Before last round */
         pDecInBlock.Length = remainingInput;
      }
      pDecOutBlock.Length = remainingOutput;

      /* Set input data */
      pDecInBlock.DataBuffer  = &ciphertext[sizeof(ciphertext)-remainingInput];
      /* Set output data */
      pDecOutBlock.DataBuffer = &plaintext_out[sizeof(plaintext_out)-remainingOutput];

      /* Perform operation */
      retVal |= SecM_Decryption( &pDecInBlock, &pDecOutBlock , &pDecParam );

      pDecParam.mode   = SEC_DECRYPTION_MODE_AES_128_PKCS_CBC; /* Set normal mode */
      remainingInput  -= pDecInBlock.Length;      /* Update remaining input bytes */
      remainingOutput -= pDecOutBlock.Length;     /* Update remaining output bytes */
   }
```

On the first call of SecM_Encryption / SecM_Decryption the mode flag SEC_DECRYPTION_MODE_INIT has to be set as well as the operation mode flag (in the example above: SEC_DECRYPTION_MODE_AES_128_PKCS_CBC). After processing the output length contains the length of the produced data and the input length contains the length of the consumed data.

While there is input data left, only the operation mode flag has to be set before calling the function. Before the last round the remaining input bytes have to be read.

On the last round the latest data input stored in the workspace will be processed and the padding will be appended. Therefore the input length has to be zero and the mode flag SEC_DECRYPTION_MODE_FINALIZE has to be set in addition to the operation mode flag.

6.4 Seed / Key

The Seed/Key interface provides the generation of a pseudo random seed, the generation of a key and a comparison function.

6.4.1 Variants

> Default and extended API

> RSA, HMAC SHA1 or OEM specific Seed/Key generation

> Internal/External workspace usage

6.4.2 Default and extended API

Depending on the OEM either the default API or the extended API is being used. The extended API additionally provides an external key and workspace.

6.4.3 Usage

This demonstration generates a seed and computes a key. The usage of the module can vary between the OEMs. If the Security Module comes along with a Bootloader delivery the OEM related demo implementation can be found in the 'fbl_ap.c' file.

Seed generation:

```
SecM_WordType EntropySource( void )
{
   return <random value>;
}

/* Generate seed value */
SecM_SeedType seed;

/* Provide initial seed for pseudo random number generator */
seed.seedX = EntropySource();
seed.seedY = EntropySource();

/* Generate seed */
if (SECM_OK == SecM_GenerateSeed(&seed))
{
   /* Seed successfully generated in seed.seedX */
}
```

The provided key can be send to the tester. The tester has to generate a key and send it back to the ECU. The ECU has to generate with the seed value an own key and compare the received key from the tester with the generated one:

```
/* Check key from tester */
#if ( SECM_HIS_SECURITY_MODULE_VERSION < 0x010002u )
if (SECM_OK == SecM_CompareKey(key))
#else
if (SECM_OK == SecM_CompareKey(key, seed)
#endif
{
   /* Security access granted */
}
```

Note
The API varies depending on the version of the HIS specification. In case of HIS specification 1.0.1 or earlier the last seed is stored in an internal structure. Later versions do use an additional parameter for the seed. However the Security Module supports both variants.

6.5 Workspace

The workspace is needed for the cryptographic algorithms. The workspaces for the specific services are included in the Security Module Library. There are three variants of workspace usage as described in the following sub-chapter.

6.5.1 Variants

> Only internal workspace can be used
> Only external workspace can be used
> Internal and external workspace can be used

6.5.2 Usage

The following table describes which parameter of the API provides the workspace information for external workspace use-case.

Module	Usage
Encryption / Decryption	Pointer and length information of the workspace buffer are provided by the 'SecM_Decryption' and 'SecM_Encryption' parameter: SecM_DecParamType * pDecParam 'SecM_DecParamType' contains an element workspace of type 'SecM_WorkspaceType'
SeedKey	Only available with the extended API: 'SecM_CompareKey' provides the parameter key. Its type definition 'SecM_KeyType' declares the element 'context' as a void pointer type. The 'SecM_WorkspaceType' can be casted for this element.
Verification	SecM_Verification: 'SecM_Verification' provides the parameter pVerifyParam of type SecM_VerifyParamType. The type definition of this parameter provides the element workspace of type SecM_WorkspaceType. SecM_VerifySignature: 'SecM_Verification' provides the parameter pVerifyParam of type SecM_SignatureParamType. The type definition of this parameter provides the element workspace of type SecM_WorkspaceType.

Table 6-1 External workspace usage

The variant 'Internal/External' needs to distinguish whether the API is using the internal or external workspace. Explicit usage of the internal buffer in this variant can be achieved by setting the workspace length information to SEC_DEFAULT_WORKSPACE_SIZE.

A demo of using the external workspace can be found in chapter 6.6.1.

6.6 Accessing the Security Module from within the Application

This chapter is only relevant if the security component comes along with the Bootloader.

There might be the necessity to use the features of the security module, located in the Bootloader, also within the application. This can be useful if functions of the Security Module shall be used in a consistent way, or ROM memory shall be saved. For this, a user-interface is available to call some routines of the security module from within the application. The file “fbl_def.h” contains the necessary definitions for the security module interface.

The usage of the definitions is equivalent to the API (chapter 7).

Definition	Description
ApplSecComputeKey	Definition for the key computation function.
ApplSecComputeCRC	Definition for the CRC computation function.
ApplSecVerifySignatureFct	Definition for the signature verification function.
ApplSecInitDecryption	Definition for the decryption initialization.
ApplSecDecryption	Definition for the decryption function.
ApplSecDeinitDecryption	Definition for the decryption de-initialization.
ApplSecGenerateSeed	Definition for the seed generation function.

Table 6-2 Function access macros for the FBL header structure

6.6.1 Usage

This demo shows how to use the Security Module within an application and the usage of an external workspace. It is adapted from the sample in 6.1.3.

```
/* Verification parameters */
SecM_VerifyParamType verifyParam;

/* Signature to be verified */
vuint8 signature[128u];

/* Workspace */
vuint8 workspaceData[WORKSPACE_SIZE];

/* List of memory segments */
FL_SegmentListType segmentList;

/* Setup section information of section #1 */
segmentList.segmentInfo[0].targetAddress      = FLASH_SECTION1_MEMORY_ADDRESS;
segmentList.segmentInfo[0].transferredAddress = FLASH_SECTION1_DOWNLOAD_ADDRESS;
segmentList.segmentInfo[0].length             = FLASH_SECTION1_LENGTH;
```

```
/* Setup section information of section #2 */
segmentList.segmentInfo[1].targetAddress      = FLASH_SECTION2_MEMORY_ADDRESS;
segmentList.segmentInfo[1].transferredAddress = FLASH_SECTION2_DOWNLOAD_ADDRESS;
segmentList.segmentInfo[1].length             = FLASH_SECTION2_LENGTH;

/* Number of memory segments */
segmentList.nrOfSegments                      = 2u;

/* Initialize verification structure */
/* Start address of full block */
verifyParam.blockStartAddress = FLASHREGION_LBT0_ADDRESS;
/* Length of full block */
verifyParam.blockLength       = FLASHREGION_LBT0_LENGTH;
/* Pass segment list */
verifyParam.segmentList       = &segmentList;
/* Function to read memory at given address */
verifyParam.readMemory        = ReadMemoryFct;
/* Cyclically called function */
verifyParam.wdTriggerFct      = WatchdogFct;
/* Pass signature */
verifyParam.verificationData  = (SecM_VerifyDataType)signature;
/* Use external workspace */
verifyParam.workspace.data = (SecM_WorkspacePtrType)workspaceData;
verifyParam.workspace.size = sizeof(workspaceData);

/* Perform signature verification */
if ( (SECM_OK == ApplSecInitVerificationFct(V_NULL))
  && (SECM_VER_OK == ApplSecVerificationFct(&verifyParam))
  && (SECM_OK == ApplSecDeinitVerificationFct(V_NULL)) )
{
   /* Verification successful */
}
```

7 API Description

7.1 Type Definitions

This chapter describes the types used by the interface functions of the security module.

7.1.1 General Types

Typedef	Elements	Basetype	Description
SecM_StatusType	Not particularly defined. Typically, returns SECM_OK or SECM_NOT_OK. See function description for details.	vuint8	Return value from any function of the security module
SecM_WordType	-	vuint32	Typically used by the seed/key functions for seed and key values.
SecM_KeyType	-	SecM_Word Type	The length of the key for seed/key calculation.
SecM_CRCType	-	SecM_Word Type	Used to store the CRC
SecM_LengthType	-	vuint32	Used to store length information for CRC calculation and decryption
SecM_AddrType	-	vuint32	Used to store address information for segments
SecM_SizeType	-	vuint32	Used to store length information of segments
SecM_VerifyInitType	-	void *	-
SecM_VerifyDeinitType	-	void *	-
SecM_VerifyDataType	-	vuint8 *	-
FL_WDTriggerFctType	typedef void (*)(void);		Pointer to a function that is frequently called from the security module on long-lasting mathematical functions.
FL_ReadMemoryFctType	typedef SecM_SizeType (*)(SecM_AddrType, vuint8 *, SecM_SizeType);		Function is used by SecM_Verification() to read data from memory.

Table 7-1 Type definitions

7.1.2 Structures used by Verification

Structure	Element	Type	Description
SecM_Verify ParamType	segmentList	FL_SegmentListType / FL_SegmentListType* [3]	Address and length information of download segments
	blockStartAddress	SecM_AddrType	Start address of logical block
	blockLength	SecM_SizeType	Length of logical block
	verificationData	SecM_VerifyDataType	Pointer to verification data (e.g. checksum and/or signature)
	crcTotal	SecM_CRCType	CRC total calculated over full logical block
	wdTriggerFct	FL_WDTriggerFctType	Pointer to watchdog trigger function
	workspace	SecM_WorkspaceType	Reference to workspace (extension of the HIS specification)
	key	SecM_VerifyKeyType	Pointer to verification key (extension of the HIS specification)

Table 7-2 SecM_VerifyParamType

Structure	Element	Type	Description
SecM_Signature ParamType	currentHash	SecM_SignatureType	Reference to current hash value (also used to pass workspace)
	currentDataLength	SecM_SizeType *	Pointer to the current length of the hashed data. Is only set when SEC_ENABLE_VERIFICATION_DATA_LENGTH is activated.
	sigState	SecM_StatusType	Signature state / operation to be executed
	sigSourceBuffer	SecM_VerifyDataType	Pointer to input or verification data
	sigByteCount	SecM_LengthType	Size of input or verification data
	wdTriggerFct	FL_WDTriggerFctType	Pointer to watchdog trigger function
	key	SecM_VerifyKeyType	Pointer to verification key (extension of the HIS

[3] As an extension to the HIS specification the segment list can be used as a pointer. This depends on the OEM.

Structure	Element	Type	Description
			specification)

Table 7-3 SecM_SignatureParamType

7.1.3 Structures used by CRC

Structure	Element	Type	Description
SecM_CRCParam Type	currentCRC	SecM_CRCType	Current CRC-value
	crcState	SecM_ByteType	Operation state to be carried out
	crcSourceBuffer	SecM_ConstRamDataT ype	Pointer to source data
	crcByteCount	SecM_LengthType	Number of bytes in source buffer
	wdTriggerFct	FL_WDTriggerFctType	Watchdog trigger function

Table 7-4 SecM_CRCParamType

7.1.4 Structures used by Encryption / Decryption

Structure	Element	Type	Description
SecM_DecInput ParamType	DataBuffer	SecM_ConstRamDataT ype	Pointer to input data
	Length	SecM_LengthType	Length of input data

Table 7-5 SecM_DecInputParamType

Structure	Element	Type	Description
SecM_DecOutput ParamType	DataBuffer	SecM_RamDataType	Pointer to output data
	Length	SecM_LengthType	Length of output buffer / data

Table 7-6 SecM_DecOutputParamType

Structure	Element	Type	Description
SecM_DecParam Type	segmentAddress	SecM_AddrType	Start address of the segment
	segmentLength	SecM_SizeType	Length of segment in bytes
	mode	SecM_ByteType	Data format ID of encryption and compression
	wdTriggerFct	FL_WDTriggerFctType	Pointer to watchdog trigger function
	workspace	SecM_WorkspaceType	Reference to workspace (extension of the HIS specification)
	key	SecM_SymKeyType	Reference to decryption key

Structure	Element	Type	Description
			(extension of the HIS specification)

Table 7-7 SecM_DecParamType

7.1.5 Structures used by Seed / Key

Structure	Element	Type	Description
SecM_KeyType	data	SecM_KeyBaseType *	Pointer to actual key data
	context	SecM_VoidPtrType	Additional context information, e.g. workspace

Table 7-8 SecM_KeyType

7.2 Common

7.2.1 SecM_InitPowerOn

Prototype	
`SecM_StatusType` **`SecM_InitPowerOn`** `(SecM_InitType initParam)`	
Parameter	
SecM_InitType initParam	Initialization parameters (unused - reserved for future use)
Return Code	
SecM_StatusType	SECM_OK if initialization successful SECM_NOT_OK otherwise
Functional Description	
Initialize security module	
Particularities and Limitations	
none	
Pre-Conditions	
Call Context	
none	

Table 7-9 SecM_InitPowerOn

7.2.2 SecM_Task

Prototype	
`void` **`SecM_Task`** `(void)`	
Parameter	
void	none
Return Code	
void	none

Functional Description
Cyclic task of security module
Particularities and Limitations
none
Pre-Conditions
Call Context
none

Table 7-10 SecM_Task

7.3 CRC

7.3.1 SecM_ComputeCRC

Prototype	
`SecM_StatusType` **`SecM_ComputeCRC`** `(V_MEMRAM1 SecM_CRCParamType V_MEMRAM2 V_MEMRAM3 *crcParam)`	
Parameter	
SecM_CRCParamType crcParam	Pointer to parameter structure
Return Code	
SecM_StatusType	SECM_OK if operation was successful SECM_NOT_OK otherwise
Functional Description	
Function that manages the state of CRC computation	
Particularities and Limitations	
none	
Pre-Conditions	
Call Context	
none	

Table 7-11 SecM_ComputeCRC

7.4 Encryption / Decryption

7.4.1 SecM_InitDecryption

Prototype	
`SecM_StatusType` **`SecM_InitDecryption`** `(SecM_DecInitType init)`	
Parameter	
SecM_DecInitType init	Dummy pointer (currently not used)

Return Code	
SecM_StatusType	SECM_OK if initialization successful SECM_NOT_OK if error occurred during initialization
Functional Description	
Initialize decryption	
Particularities and Limitations	
none	
Pre-Conditions	
Call Context	
none	

Table 7-12 SecM_InitDecryption

7.4.2 SecM_DeinitDecryption

Prototype	
`SecM_StatusType` **`SecM_DeinitDecryption`** `(SecM_DecDeinitType deinit)`	
Parameter	
SecM_DecDeinitType deinit	Dummy pointer (currently not used)
Return Code	
SecM_StatusType	SECM_OK if deinitialization successful SECM_NOT_OK if error occurred during deinitialization
Functional Description	
Deinitialize decryption	
Particularities and Limitations	
none	
Pre-Conditions	
Call Context	
none	

Table 7-13 SecM_DeinitDecryption

7.4.3 SecM_Decryption

Prototype	
`SecM_StatusType` **`SecM_Decryption`** `(const V_MEMRAM1 SecM_DecInputParamType V_MEMRAM2 V_MEMRAM3 *pInBlock, V_MEMRAM1 SecM_DecOutputParamType V_MEMRAM2 V_MEMRAM3 *pOutBlock, V_MEMRAM1 SecM_DecParamType V_MEMRAM2 V_MEMRAM3 *pDecParam)`	
Parameter	
SecM_DecInputParamType pInBlock	Reference to input data

SecM_DecOutputParamType pOutBlock	Reference to output data. Length contains available buffer size as input and length of decrypted data as output
SecM_DecParamType pDecParam	Decryption parameters (segmentAddress and segmentLength not used yet)
Return Code	
SecM_StatusType	SECM_OK if decryption successful SECM_NOT_OK if error occurred during decryption
Functional Description	
Decrypts all encrypted data referenced in input block into buffer provided in output block. Operation reports an error in case the output buffer isn't large enough.	
Particularities and Limitations	
none	
Pre-Conditions	
Call Context	
none	

Table 7-14 SecM_Decryption

7.4.4 SecM_InitEncryption

Prototype	
`SecM_StatusType` **`SecM_InitEncryption`** `(SecM_EncInitType init)`	
Parameter	
SecM_EncInitType init	Dummy pointer (currently not used)
Return Code	
SecM_StatusType	SECM_OK if initialization successful SECM_NOT_OK if error occurred during initialization
Functional Description	
Initialize encryption	
Particularities and Limitations	
none	
Pre-Conditions	
Call Context	
none	

Table 7-15 SecM_InitEncryption

7.4.5 SecM_DeinitEncryption

Prototype
`SecM_StatusType` **`SecM_DeinitEncryption`** `(SecM_EncDeinitType deinit)`

Parameter	
SecM_EncDeinitType deinit	Dummy pointer (currently not used)
Return Code	
SecM_StatusType	SECM_OK if deinitialization successful SECM_NOT_OK if error occurred during deinitialization
Functional Description	
Deinitialize encryption	
Particularities and Limitations	
none	
Pre-Conditions	
Call Context	
none	

Table 7-16 SecM_DeinitEncryption

7.4.6 SecM_Encryption

Prototype	
`SecM_StatusType` **`SecM_Encryption`** `(const V_MEMRAM1 SecM_EncInputParamType V_MEMRAM2 V_MEMRAM3 *pInBlock, V_MEMRAM1 SecM_EncOutputParamType V_MEMRAM2 V_MEMRAM3 *pOutBlock, V_MEMRAM1 SecM_EncParamType V_MEMRAM2 V_MEMRAM3 *pEncParam)`	
Parameter	
SecM_EncInputParamType pInBlock	Reference to input data
SecM_EncOutputParamType pOutBlock	Reference to output data. Length contains available buffer size as input and length of encrypted data as output
SecM_EncParamType pEncParam	Encryption parameters (segmentAddress and segmentLength not used yet)
Return Code	
SecM_StatusType	SECM_OK if decryption successful SECM_NOT_OK if error occurred during decryption
Functional Description	
Encrypts all data referenced in input block into buffer provided in output block. Operation reports an error in case the output buffer isn't large enough.	
Particularities and Limitations	
none	
Pre-Conditions	
Call Context	
none	

Table 7-17 SecM_Encryption

7.5 Seed / Key Common

7.5.1 SecM_GenerateSeed

Prototype	
`SecM_StatusType` **`SecM_GenerateSeed`** `(V_MEMRAM1 SecM_SeedType V_MEMRAM2 V_MEMRAM3 *seed)`	
Parameter	
SecM_SeedType seed	Pointer to seed, where the random values shall be stored
Return Code	
SecM_StatusType	SECM_OK
Functional Description	
Uses a pseudo random number generator and an initial seed to generate a seed value. In case of HIS specification 1.0.1 or earlier the last seed is stored in an internal structure.	
Particularities and Limitations	
none	
Pre-Conditions	
The values of seedX and seedY (default API) or seed[0] (extended API) should have a random start value.	
Call Context	
none	

Table 7-18 SecM_GenerateSeed

7.5.2 SecM_CompareKey (HIS Specification < 1.0.1[4])

Prototype	
`SecM_StatusType` **`SecM_CompareKey`** `(SecM_KeyType key)`	
Parameter	
SecM_KeyType key	Key for the authorisation
Return Code	
SecM_StatusType	SECM_OK if key calculation and comparison was successful SECM_FAILED if key calculation failed or key mismatch
Functional Description	
Runs the key calculation and compares the calculated key against the received key	
Particularities and Limitations	
none	
Pre-Conditions	
Function SecM_GenerateSeed must have been called before.	
Call Context	

[4] Depending on the OEM

none

Table 7-19 SecM_CompareKey

7.5.3 SecM_CompareKey (HIS Specification >= 1.0.1)

Prototype	
`SecM_StatusType` **`SecM_CompareKey`** `(SecM_KeyType key, SecM_SeedType lastSeed)`	
Parameter	
SecM_KeyType key	Key for the authorization contains additional parameters (workspace, secret key) in case of extended API
SecM_SeedType lastSeed	Start value (seed) for the authorisation
Return Code	
SecM_StatusType	none
Functional Description	
Runs the key calculation and compares the calculated key against the received key	
Particularities and Limitations	
none	
Pre-Conditions	
Function SecM_GenerateSeed must have been called before.	
Call Context	
none	

Table 7-20 SecM_CompareKey

7.6 Seed / Key default API

7.6.1 SecM_ComputeKey

Prototype	
`SecM_StatusType` **`SecM_ComputeKey`** `(SecM_SeedType inputSeed, SecM_ConstType constant, V_MEMRAM1 SecM_KeyStorageType V_MEMRAM2 V_MEMRAM3 *computedKey)`	
Parameter	
SecM_SeedType inputSeed	The random seed the key calculation shall be based on
SecM_ConstType constant	A fixed constant used by the key calculation algorithm contains additional parameters (workspace, secret key) in case of extended API
SecM_KeyStorageType computedKey	Pointer to resulting key data as a formula of key = f(seed,k)
Return Code	
SecM_StatusType	SECM_OK if calculation was successful SECM_FAILED if calculation has failed (e.g. wrong parameters)

Functional Description
Calculates a key value based on the provided seed and constant value
Particularities and Limitations
none
Pre-Conditions
Function SecM_GenerateSeed must have been called at least once.
Call Context
none

Table 7-21 SecM_ComputeKey

7.7 Seed / Key extended API

7.7.1 SecM_ComputeKey

Prototype	
`SecM_StatusType` **`SecM_ComputeKey`** `(SecM_KeyType inputKey, SecM_ConstType constant, V_MEMRAM1 SecM_SeedType V_MEMRAM2 V_MEMRAM3 *computedSeed)`	
Parameter	
SecM_KeyType inputKey	The received key on which calculation shall be based
SecM_ConstType constant	A fixed constant used by the key calculation algorithm contains additional parameters (workspace, secret key) in case of extended API
SecM_SeedType computedSeed	Pointer to resulting seed as a formula of seed = f(key,k)
Return Code	
SecM_StatusType	SECM_OK if calculation was successful SECM_FAILED if calculation has failed (e.g. wrong parameters)
Functional Description	
API is a deviation from the HIS specification. Modified to support asymmetric key calculation where only the tester is able to generate the key from the original seed value, e.g. using a private key.	
Particularities and Limitations	
none	
Pre-Conditions	
Function SecM_GenerateSeed must have been called at least once.	
Call Context	
none	

Table 7-22 SecM_ComputeKey

7.8 Verification

7.8.1 SecM_InitVerification

Prototype	
`SecM_StatusType` **`SecM_InitVerification`** `(SecM_VerifyInitType init)`	
Parameter	
SecM_VerifyInitType init	Dummy pointer (currently not used)
Return Code	
SecM_StatusType	SECM_OK if initalization successful SECM_NOT_OK if error occured during initalization
Functional Description	
Initializes the verification	
Particularities and Limitations	
none	
Pre-Conditions	
Call Context	
none	

Table 7-23 SecM_InitVerification

7.8.2 SecM_DeinitVerification

Prototype	
`SecM_StatusType` **`SecM_DeinitVerification`** `(SecM_VerifyDeinitType deinit)`	
Parameter	
SecM_VerifyDeinitType deinit	Dummy pointer (currently not used)
Return Code	
SecM_StatusType	SECM_OK if deinitalization successful SECM_NOT_OK if error occured during deinitalization
Functional Description	
Deinitializes the verification	
Particularities and Limitations	
none	
Pre-Conditions	
Call Context	
none	

Table 7-24 SecM_DeinitVerification

7.8.3 SecM_Verification

Prototype	
`SecM_StatusType` **`SecM_Verification`** `(V_MEMRAM1 SecM_VerifyParamType V_MEMRAM2 V_MEMRAM3 *pVerifyParam)`	
Parameter	
SecM_VerifyParamType pVerifyParam	Pointer to parameter structure for signature verification crcTotal.
Return Code	
SecM_StatusType	SECM_VER_OK if signature verification successful SECM_VER_ERROR if error occured during verification SECM_VER_CRC if CRC verification failed SECM_VER_SIG if HMAC/Signature verification failed
Functional Description	
Calculates the checksum or signature of all segments present in the verification parameter function passed in readMemory is used to access memory. Afterwards the calculated value is used to verify the checksum/signature provided in verificationData. If configured a CRC over the complete block, defined by blockStartAddress and blockLength, including all readable gaps between the actual segments is calculated too and returned in parameter	
Particularities and Limitations	
none	
Pre-Conditions	
Call Context	
none	

Table 7-25 SecM_Verification

7.8.4 SecM_VerifySignature

Prototype	
`SecM_StatusType` **`SecM_VerifySignature`** `(V_MEMRAM1 SecM_SignatureParamType V_MEMRAM2 V_MEMRAM3 *pVerifyParam)`	
Parameter	
pVerifyParam	Pointer to parameter structure for signature verification member currentHash must (SEC_DISABLE_DEFAULT_WORKSPACE) or may (SEC_ENABLE_WORKSPACE_INTERNAL) contain reference to buffer used as workspace.
Return Code	
SecM_StatusType	SECM_VER_OK if verification operation successful SECM_VER_ERROR if error occurred during verification SECM_VER_CRC if CRC verification failed SECM_VER_SIG if HMAC/Signature verification failed
Functional Description	
Calculate and verify checksum or signature using primitive of configured security class	

Particularities and Limitations
none
Pre-Conditions
Call Context
none

Table 7-26 SecM_VerifySignature

8 Contact

Visit our website for more information on

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

www.vector.com