MOTIX™ TLE989x/TLE988x

Firmware User Manual

About this document

Scope and purpose

This document provides technical information on the BootROM firmware features of the TLE989x/TLE988x and technical guidance on how to interact with the device using embedded mechanisms and the user API functions. The subsequent sections provide the necessary information for device configuration and BootROM firmware API handling.

The BootROM firmware for the TLE989x/TLE988x family provides the following features:

- Startup procedure
- Support for connecting debuggers
- Default Bootstrap Loader (BSL) for NVM programming and diagnostics
- Support for proprietary user BSL
- NVM operations handling like programming, erasing, and verifying the NVM
- Cryptographic library

Intended audience

The intended audience are software developers, application system integrators, and debugging tool vendors.

Table of contents

1 Firmware architecture

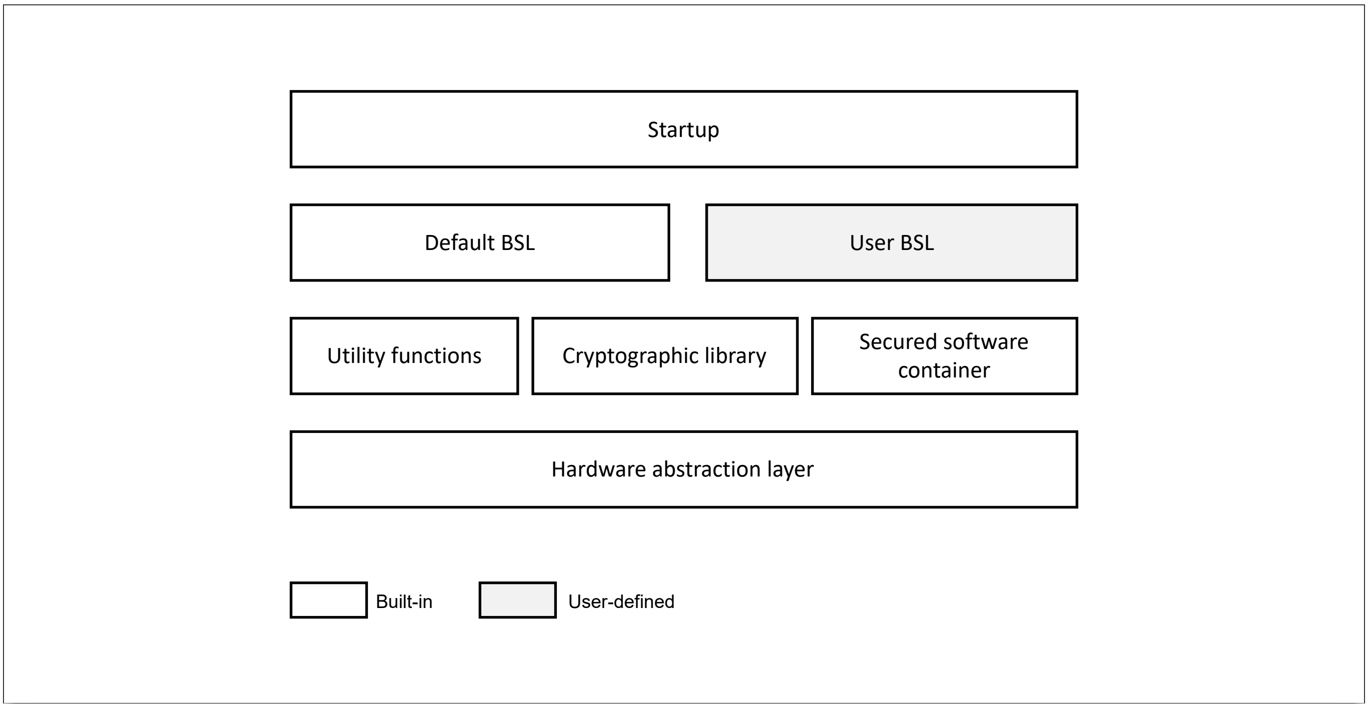

Figure 1 Firmware architecture

1.1 Startup

The Startup module includes these features:

- It executes the first software-controlled operation in the BootROM that is automatically executed after every reset.
- It performs different device initialization steps and enters the operation mode determined by the provided configuration.
- Executed with the highest privilege level, it cannot be called from a debugger, and cannot be re-entered after the sequence completion.
- It uses the various routines of the lower abstraction levels.

The Startup module expects the startup configuration in the first page of the User BSL segment (UBSL). A detailed description of the startup page and its parameters can be found in the MCU chapter of the user manual.

1.2 Default Bootstrap Loader (BSL)

The Default Bootstrap Loader (BSL) module supports uploading user application code into the NVM using a message-based command request-and-response communication. The communication interface is UART over CAN.

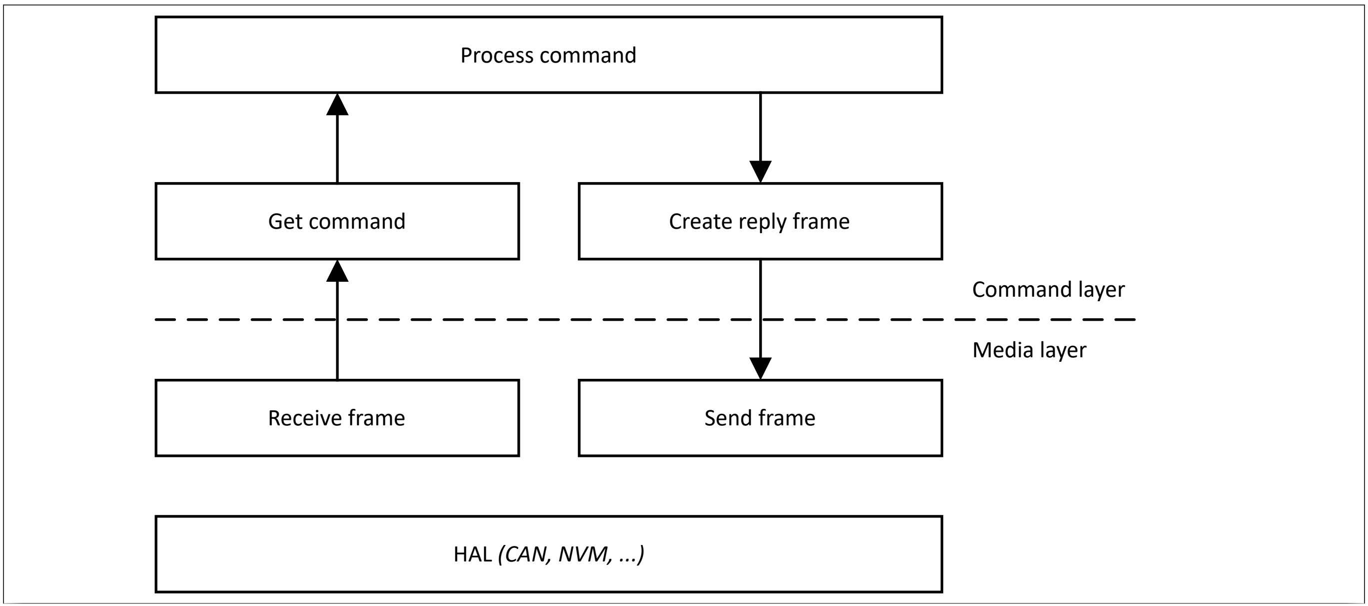

Figure 2 BSL architecture

Command layer

This layer is responsible for parsing and processing the command. If the command is valid, it will perform the requested operation and prepare the reply frame.

Media layer

This layer is responsible for receiving data over the communication interface and assembling it into a complete frame. Intraframe timeout measurement is used to keep track of frame reception. Only correct frames received within the intraframe time window will be further transported to the command layer.

A media frame is used to transmit data to the device or to receive a response from the device.

1.3 User Bootstrap Loader (UBSL)

The UBSL module supports uploading user application programs into the NVM using proprietary user-defined protocols and flows. Firmware functions available via User Firmware API are available for execution of the required low level operation for NVM programming, use of the Cryptographic library and other essential routines. The UBSL can optionally be executed before branching to the user code. The UBSL is not a part of the BootROM code.

1.4 Utility functions

The BootROM exposes some library functions to the user mode software. These library functions allow configuration of the device boot parameters and access the NVM.

The main features of the utility functions are the following:

- Reading and writing the various 100TP pages inside the NVM
- Writing and erasing the NVM pages and sectors
- Configuring the BSL parameters (for example, timeout configuration, NAD address)
- Retrieving the customer identification number
- Performing a RAM MBIST test
- Checking for single and double ECC errors in the NVM

1.5 Cryptographic library

The Cryptographic library provides support for cryptographic operations using security keys embedded in the protected key storage. It supports these algorithms:

- AES256
- CMAC

The selected key is accessed in the background according to the argument referencing the key ID in the key storage. The secure context code and temporary data remains inaccessible for user-context routines.

Note: Only one cryptographic operation can be executed at a time. If a cryptographic operation was started by the corresponding API start function, another cryptographic operation can only be started if the API finish function of the previously started operation was called.

1.6 Secured software container

The Secured Software Container can be used to protect specific code from being read by third parties. It can be installed by using a dedicated BootROM firmware API.

2 Boot modes

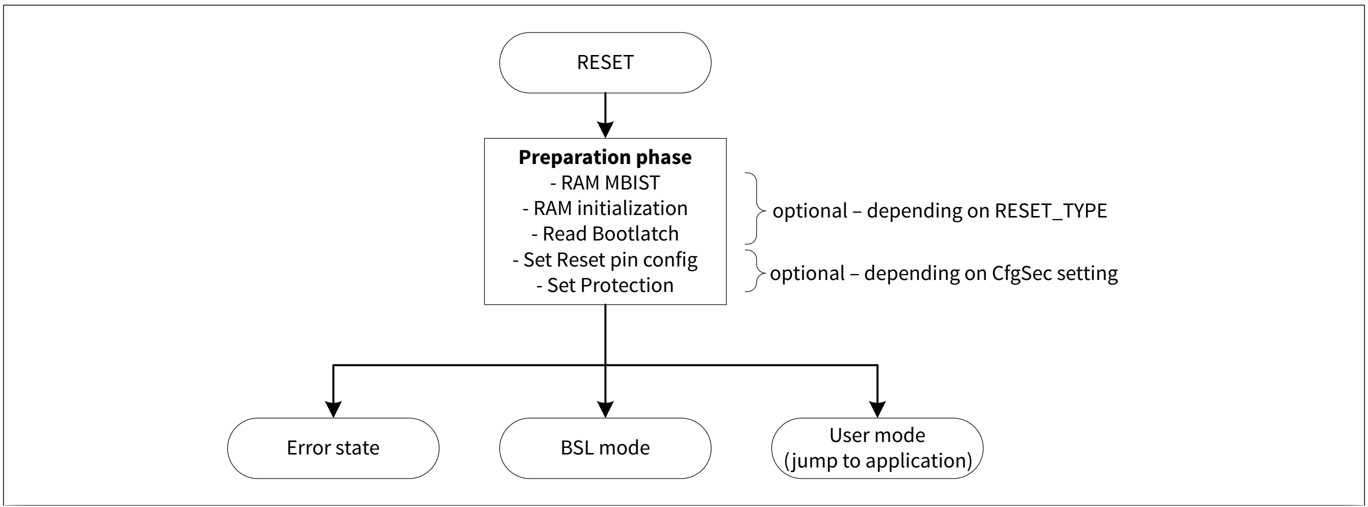

Figure 3 BootROM firmware startup

2.1 BSL mode

The BSL mode follows a serial communication protocol between a host and the device. The protocol is UART-based (half-duplex), the interface is over the built-in CAN transceiver. The BSL mode is entered when the host sends the correct passphrase (see Table 1) within the configured no-activity-counter time (NAC). The NAC value is stored in the startup page.

In BSL mode the FS_WDT is disabled.

2.1.1 Host synchronization

The host synchronization consists of a single BSL frame with the format:

- [length] + [NAD] + [PASSPHRASE] + [chk]

The [PASSPHRASE] is composed by the ASCII values of the word "PASSPHRASE" as shown in the table below. The passphrase frame in the Default BSL protocol is extended by a checksum byte field. Details about frame encapsulation are given in Table 1. Upon successful reception of a valid passphrase frame, the device sends back single acknowledge byte 55_H and is ready for receiving BSL commands.

Table 1 Passphrase frame format

0	1	2	3	4	5	6	7	8	9	10	11	12
Length	NAD	0x50 "P"	0x41 "A"	0x53 "S"	0x53 "S"	0x50 "P"	0x48 "H"	0x52 "R"	0x41 "A"	0x53 "S"	0x45 "E"	Chk

The NAD address is stored in the Startup page when the device is programmed. A detailed description of the Startup page can be found within the MCU chapter in the User's Manual.

The host synchronization is completed when the full passphrase has been received before the NAC timer expires.

Note: If no valid startup configuration is installed, the device enters the Default BSL and infinitely waits for a valid passphrase.

2.1.1.1 NAD address

The NAD field specifies the address of the active responder node (only responder nodes have NAD addresses). Table 2 lists NAD address ranges supported by the BootROM firmware.

Table 2 NAD address range

NAD value	Description
00_H to FE_H	This is the valid address range for addressing an individual responder node.
FF_H	This is the broadcast address for addressing responders concurrently. It is the default address if no NAD value is programmed.

2.1.1.2 NAC time

The No-Activity Counter (NAC) value defines a time window with a granularity of 5 ms. After reset release, the firmware is able to receive a BSL passphrase within the specified time frame. If no BSL passphrase is received before the NAC expires, the firmware code proceeds to execute the user code. In case of an invalid NAC value in the NAC location, a "wait forever" (NAC set to FF_H) is given to the Default BSL module. A changed NAC value takes effect only after the next reset.

The maximum NAC timeout is 140 ms (NAC set to $1C_H$). The BootROM firmware reads the NAC from the Startup page and sets the NAC time window accordingly. The translation from NAC value to NAC time window is explained in Table 3.

Table 3 NAC time window

NAC value	Timeout behavior
< 02_H	The Default BSL window is closed, no BSL connection is possible and execution jumps to user code immediately.
02_H to $1C_H$	There is a timeout delay of NAC*5 ms before jumping to user code.
> $1C_H$	No timeout is used. The BootROM firmware switches off the FS_WDT and waits indefinitely for a Default BSL connection attempt.

Note: The time quantum of 5 ms refers to a nominal HP_CLK frequency not considering actual frequency deviations imposed by the HP_CLK accuracy.

2.1.2 Media frame format

The media frame uses the general [length] [message] [chk] format, regardless of the communication interface.

Media frames are used to send data to the device or to receive a response from the device.

- [Length] denotes the number of successive bytes in the frame.
- [Message] contains the data block sent to or received from the device. The size is in the range of 1 to 133 bytes.
- [Chk] is the media frame checksum. It is calculated over the length byte and the message bytes.

Table 4 Media frame format

1 byte	1 up to 133 bytes	1 byte
Length	Message	Chk

The message contained in the media frame has the format [message type] [arguments]. Depending on the message type, the message is referred to as "command frame" or "response frame".

- [Message type] is a CMD_ID (BSL command number) or a RESP_ID (BSL response number).
- [Arguments] are optional with a length of 0 to 132 bytes. It contains the arguments required for a specific BSL command or BSL response.

Table 5 Message contained in a media frame

1 byte	0 up to 132 bytes
Message type	Arguments

2.1.3 Media frame timing

The host has to add a delay after each sent Default BSL command header before sending the next one. The BootROM firmware also requires an additional waiting time to process the complete received Default BSL command. During this period of time, no response message can be provided by the BootROM firmware and hence the host cannot send new commands. The host must wait this length of time before sending a new command.

To give the BootROM firmware time to process each byte in a CMD or EOT frame, the byte and frame timing must comply with the values shown in Table 6.

Table 6 Default BSL byte and frame timing limits

Delay type	Minimum interval duration [µs]
Between bytes	3.7
Host waiting time after reception of a response before a new frame can be sent	20

Certain Default BSL commands involve NVM write or erase operations and hence need longer processing times. The host waiting time is longer before a command response can be requested or before a result is sent back.

As an example, changing a value in an already programmed NVM page (which happens if a setting is being changed) requires the following steps:

- Read the full page into the hardware assembly buffer
- Update the hardware buffer with new data
- Program the page from the hardware assembly buffer
- Erase the old page

This complete procedure takes approximately 8 ms (nominal value). The processing time must always be taken into account.

2.1.4 Media frame timeout

To keep track of Default BSL frame transmission violations, a frame transmission timeout is used between the different media frames. Default BSL frame transmission timeouts depend on whether a host synchronization has been done or not:

- Before host synchronization: The NAC timeout value is used as the frame timeout. If a timeout is reached, this means that the NAC timer has expired.
- After host synchronization: The BootROM firmware starts polling the incoming bytes. If a valid frame is received before the frame timeout, the firmware continues parsing and handling the BSL command.

  When a frame timeout occurs, the firmware clears the receive buffer and re-starts the timeout to receive a new media frame. The frame timeout is set to 280 ms (nominal value).

2.2 User mode

In the user mode, the BootROM firmware hands over the control to the user application code. To enter the User mode, the TMS pin needs to be kept low during startup.

The entry address for the user code after the startup sequence is determined by the Arm® Cortex®[1)]-M3 compliant vector table as described within the MCU chapter in the User Manual. To transition into the user application code, the BootROM firmware loads the initial stack pointer into the stack pointer register and jumps to the user reset vector address. Both, the initial stack pointer and the user reset vector are specified within the Startup page as described in the MCU chapter in the User Manual. They must be configured during building and linking of the application software.

2.2.1 Debug mode

The Debug mode is an option of the User mode and allows the user to debug the user code via SWD interface. To enter the Debug mode, the TMS and P0.0 pins need to be kept high during startup. More information can be found within the MCU chapter of the User Manual.

In Debug mode the FS_WDT is disabled.

Note: The SWD connection is only available if no permanent memory protection is set. Otherwise, the SWD connection is disconnected from the CPU.

2.2.2 Secure boot

The Secure Boot feature prevents the UBSL code from being executed if the data integrity is no longer given. For this purpose, the boot key has to be stored along with the UBSL code size and the Secure Boot signature within the Startup Page. If all conditions are met, the BootROM firmware proceeds with the Secure Boot.

A detailed description of the Secure Boot along with its configuration via the Startup page can be found within the MCU chapter in the User Manual.

While Secure Boot execution the FS_WDT is enabled.

2.2.3 Reset pin configuration

The BootROM firmware further supports the configuration option for the P0.10 I/O function. By default, this pin is configured as GPIO but can be reconfigured as a dedicated reset pin.

The configuration can be changed by using the BootROM firmware API to change the 100TP entry PMU_START_CONFIG. The PMU_START_CONFIG 100TP entry is then read during startup to eventually program the behavior of pin P0.10. If the startup sequence fails, for example, due to a corrupted 100TP section, a default configuration of the P0.10 behavior is installed. See User Manual for more details.

2.3 Error state

To ensure that the device is properly booted, error checking and error handling are added to the startup procedure. If a startup error occurs, the BootROM firmware enters an endless loop.

In Error state the FS_WDT is enabled.

1 Arm and Cortex are registered trademarks of Arm Limited, UK

3 Programming model

3.1 Memory protection and handling

3.1.1 Read-while-write (RWW)

The diagram below illustrates the read-while write (RWW) functionality which allows for example code execution from FLASH1 while data is written to FLASH0.

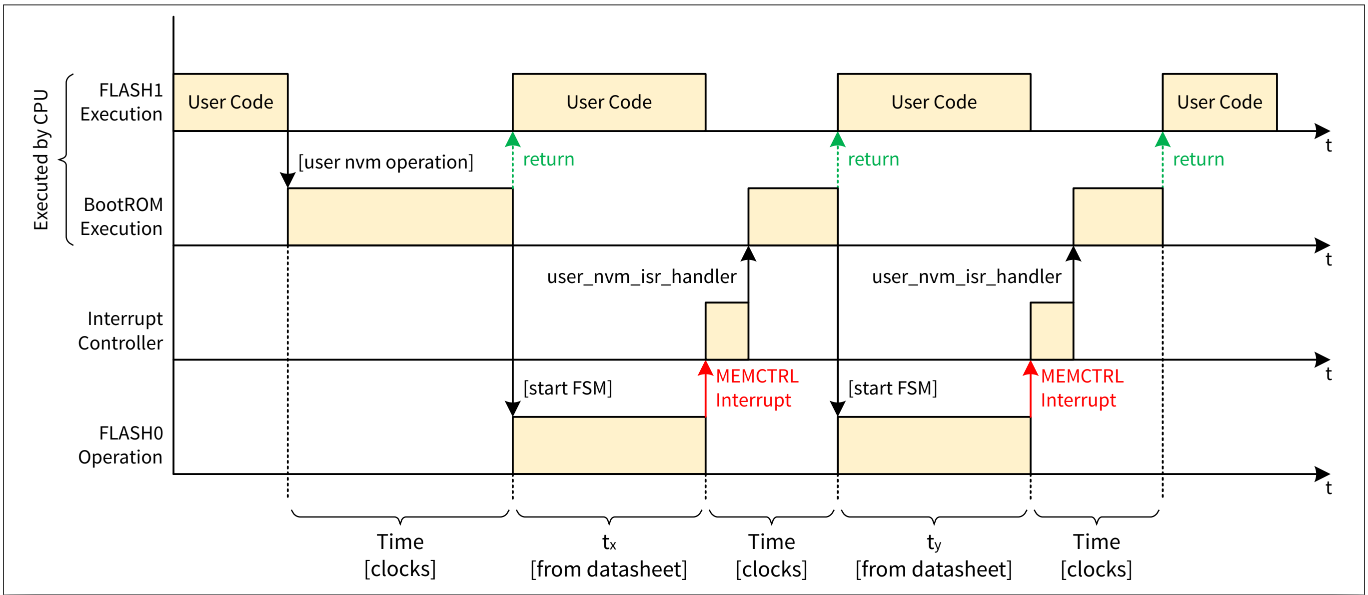

Figure 4 **RWW sequence**

The entire operation consists of one or two steps (two in the example above). Each step handles one NVM operation like for example a write or erase operation. When the user calls the [user nvm operation] API from FLASH1, targeting FLASH0, the API validates the arguments and starts the FSM of the NVM to perform the first of the two operations. Directly after starting the FSM, the BootROM firmware returns to the user application code. As soon as the NVM operation finishes, an interrupt is raised, calling the user_nvm_isr_handler. In case only one operation needs to performed by the NVM, the ISR handler of the BootROM firmware returns to the user application code and the RWW sequence is finished. In case a second operation needs to be performed by the NVM, the user_nvm_isr_handler starts the FSM a second time and returns to the user application code until the FSM finishes the second time, raising the ISR a second time. The ISR handler updates the result in the result register and concludes the entire operation.

During the NVM operation FLASH0 is in busy state and read accesses to it would be pended. However, code execution and reading on FLASH1 in the meantime is possible. The RWW feature is available for user_nvm_page_write, user_nvm_page_erase and user_nvm_sector_erase and is activated by default if the operation source and target are not in the same memory. This could be for example code from FLASH0 writes to FLASH1 or code from PSRAM writes to FLASH0/1. The user can explicitly deactivate the RWW feature, causing the API to only return to the caller after the entire NVM operation has been completed.

If the user data part of FLASH0 is being written or erased, the user must not access the FLASH0 while the BootROM firmware is executing. Otherwise, the access might result in a device reset due to interference with the BootROM firmware execution. For the case the RWW is disabled this can, for example, occur if the BootROM firmware is interrupted by an ISR. In case the RWW is enabled it can, for example, occur if the user_nvm_isr_handler is interrupted by another ISR. In both cases, the corresponding ISR must not be located within FLASH0 (including the vector table) or interrupts should be disabled as long as the API call did not ultimately finish.

Note: The RWW interrupt handler can be interrupted by higher priority interrupts.

3.1.2 NVM read protection

The read protection can be set individually for each segment (User Code, User Data, User BSL). It prevents the content of a protected segment from being read by other segments, with permission determined by the access-privileged level. If a read protection is set for any segment, the Serial Wire Debug (SWD) port is disconnected from the MCU and a debugger connection cannot be established anymore. More information on SWD and debugging can be found within the MCU chapter of the User Manual.

Note: The flash read protection is controlled by the permanent protection as described in Permanent protection.

3.1.3 Permanent protection

If the permanent protection is set for the target NVM segment, it blocks any device-internal BSL command which can be used to download code to the device. The permanent protection can be configured in the default BSL mode and secured with a passphrase. The permanent protection is activated during the boot process by the BootROM firmware before the default BSL mode is entered and before a debugger can establish an SWD connection.

The permanent protection cannot be modified with code executed from within the MCU. It can only be modified via the Default BSL commands Cmd 0x89 NVM permanent protection set and Cmd 0x98 NVM permanent protection clear. More information on the permanent protection can be found within the MCU chapter in the User Manual.

3.1.4 Service algorithm

The service algorithm (SA) is executed during startup as part of the map RAM initialization function in case failures occurred during the map RAM initialization. The SA scans the entire data flash sector and tries to repair faulty pages if possible. The SA further scans for pages which point to the same map RAM entry (double mapping). Up to one double mapping can be resolved by deleting one of the two pages. If more than one double mapping exist, the SA cannot repair them. The SA can also be started by the user via the user API of the BootROM firmware.

The SA is enabled by default but is skipped if the User Code segment (UCODE) is write protected. However, this can be changed via the NVM_SA_WITH_PROT entry of 100TP to run the SA regardless of an active UCODE write protection.

3.2 Cryptographic operations and security

The BootROM firmware API provides a variety of cryptographic operations to encrypt and decrypt data. Furthermore, the API can be used to download and store software in a secured context.

3.2.1 AES operation

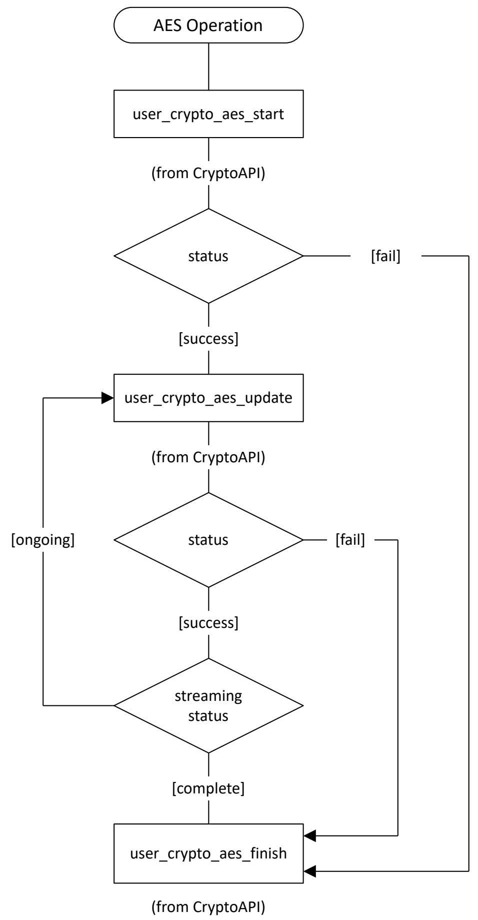

Figure 5 AES operation

First, the user_crypto_aes_start function is called along with the arguments for the AES mode, the key ID, and a pointer to the initial CBC vector. This function initializes the necessary internal variables and internal buffers. After the successful execution of the start function, the user_crypto_aes_update function has to be called at least once to perform the actual cryptographic operation. The update function requires a data structure containing a pointer to the input/output buffer including the pointer to the buffer and the buffer length for each. After successful execution of the update function, the output buffer length variable contains the number of bytes written to the output buffer.

The update function can be called multiple times if the user wants to process the data in multiple steps. For example, if the total input data length is 512 bytes, the update function can be called four times, with each call processing 128 bytes of data. It can also process 512 bytes data in a single call. Each call of the update function updates the result. At the end, the user_crypto_aes_finish function concludes the cryptographic process and resets the internal states and variables. The input data is optional. If no additional input data has to be passed, the input length can be set to 0.

Note: If one of the steps fails, the finish function must be called as well to reset the internal states and variables.

The AES decryption is done in the same way like encryption but does not require an initial vector as it is already part of the ciphertext.

3.2.2 CMAC operation

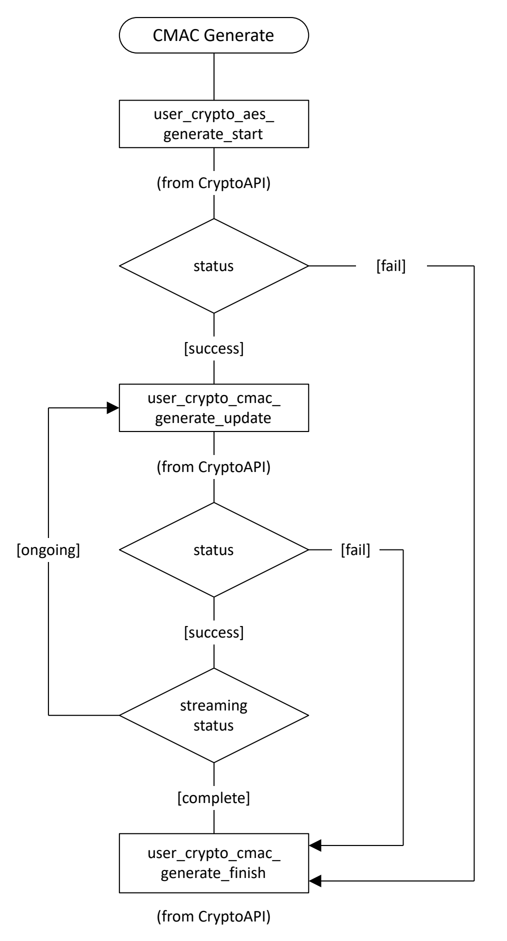

Figure 6 CMAC operation

First, the user_crypto_aes_cmac_generate_start function is called, with a key ID as argument. This function initializes the necessary internal variables and internal buffers. After the successful execution of the start function, the user_crypto_aes_cmac_generate_update function has to be called at least once to perform the actual cryptographic operation. The update function requires a data structure containing a pointer to the input buffer and the input data length. It will add the passed input data to the MAC calculation and update the intermediate result. The update function can be called multiple times if the user wants to process the data in multiple steps.

At the end, the user_crypto_aes_cmac_generate_finish function completes the MAC generation and resets the internal states and variables. The input data is optional. If no additional input data has to be passed, the input length can be set to 0. The output buffer length variable indicates the length of MAC that was written to the output buffer.

Note: If one of the steps fails, the finish function must be called as well to reset the internal states and variables.

The verification functions for start (user_crypto_aes_cmac_verify_start) and update (user_crypto_aes_cmac_verify_update) are used like their generation counterparts. The user_crypto_aes_cmac_verify_finish function compares the MAC given as argument to the MAC being generated by the verify functions. If they match, it returns ERR_LOG_SUCCESS, otherwise ERR_LOG_CODE_CMAC_VERIFY_FAIL.

3.2.3 Key write operation

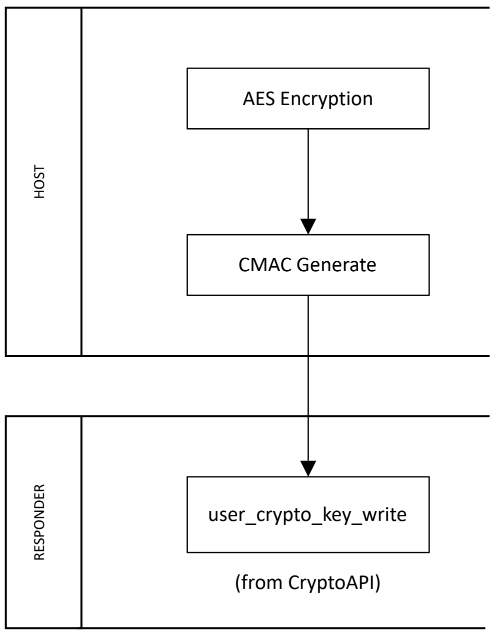

Figure 7 Key write operation

Writing a new key is done in three consecutive steps. As described below, these are the AES encryption, CMAC generation, and eventually the key write operation itself.

1. AES Encryption

   Initialize the new key variable new_key of type user_key_t and encrypt it with AES by calling the appropriate API functions (see AES operation) with the CBC mode and encrypt_key_id as arguments. The 36 bytes of input data are then transformed into the output data, consisting of 64 bytes.
2. CMAC Generate

   After the AES encryption, the output of the encryption, the target_key_id, and encrypt_key_id are assembled together as user_key_write_params_t. The user_key_write_params_t are then used for the CMAC generation (see CMAC operation) to calculate the 16-byte MAC.
3. Writing the new crypto key

As the final step, the user_key_write_params_t and the generated MAC signature are passed as user_key_write_t to the user_crypto_key_write function. It validates the passed MAC signature against the calculated MAC from user_key_write_params_t. If they match, it continues with decrypting the ciphertext of new_key to get the plain new_key. In the end, the plain new_key is written into the target key slot and its redundant slot.

3.2.4 Secured software container

The Secured Software Container is used to store software which shall not be accessible besides its execution. This can for example be used to mitigate the risk of leaks to keep the software closed source. Code stored within the Secured Software Container can be executed by the CPU but it cannot be read from, even though it shares the same privilege level as the UCODE segment. Reading the Secured Software Container will result in a bus fault.

Using the Secured Software Container will block the SWD connection due to the permanent protection installed by the BootROM firmware. Hence, it is important to prepare the UBSL to be able to flash code into the Secured Software Container and further to perform an update at a later point in time. To flash code into the Secured Software Container, a dedicated BootROM firmware API has to be used by the UBSL as described in Secure software download.

For more information see the MCU chapter within the User Manual.

3.2.5 Secure software download

The secure download flow allows downloading encrypted data into a dedicated Secured Software Container. During the operation, the encrypted data will be decrypted and then written into the Secured Software Container. The data to be downloaded needs to be encrypted beforehand with AES CBC and the correct key.

Downloading procedure

To perform a secure download, the user_secure_download_start has to be called along with the key_id, the number of sectors to be written and a pointer to the input buffer of the data. The start function then initializes the secure download by erasing the secure container, erasing the PSRAM, and the initialization of the cryptographic context for the AES CBC decryption. The start function expects the first two cipher blocks of input data (32 bytes), allowing the following secure download procedure to output the decrypted data page aligned.

After the secure download was started successfully, the user_secure_download_update function is used to continue the secure download. It has to be called at least once to generate the decrypted data and expects the page index as well as a pointer to the input buffer. Upon a successful execution of the update function, the 128 bytes of decrypted data are written into the requested Secured Software Container. The update function can be called multiple times if more data needs to be written to the Secured Software Container. The target address is determined by

[Secured Software Container start address] + [page_index] * 128

At the end of the secure software download, the user_secure_download_finish function needs to be called once to reset the internal states, variables, and the cryptographic context. The download can be verified afterwards with the CMAC verification APIs. In case the verification fails, the user is advised to repeat the secure software download.

Example

The complete procedure for the secure software download and its preparation is summarized in the following example where 240 bytes of code are downloaded into the Secured Software Container:

<u>Preparation</u>

1. Add padding to make the data page aligned (being 256 bytes)
2. Encrypt the 256 bytes with AES CBC and perform a CMAC generation on the data to get 288 bytes of encrypted data along with a 16 bytes signature

<u>Download</u>

1. Start the secure software download by calling user_secure_download_start with the key_id, size = 1, and the first 32 bytes of encrypted data
2. Call the user_secure_download_update function with page_index = 0 along with the next 128 bytes of encrypted data
3. Call the user_secure_download_update function with page_index = 1 along with the last 128 bytes of encrypted data
4. Finish the secure software download with user_secure_download_finish
5. Start the verification by calling the user_crypto_aes_cmac_verify_start function with the same key_id
6. Call the user_crypto_aes_cmac_verify_update function with the first 128 bytes of decrypted data
7. Call the user_crypto_aes_cmac_verify_update function with the second 128 bytes of decrypted data
8. Finish the verification by calling user_crypto_aes_cmac_verify_finish with the CMAC signature from the preparation step 2

3.3 Debug interface

After the BootROM firmware has initialized the device, the debug support mode is available for the Serial Wire Debug (SWD) interface. It supports the following features:

- Regular Arm® Cortex®-M3 debug features
- Firmware API calls supported by the utility functions and the cryptographic library

The BootROM firmware then hands over the execution to the user application which can then wait for a debugger to establish a proper SWD connection. This so-called wait-for-debug is delivered with the SDK and is used to establish a SWD connection before the user code starts executing. Otherwise, the connection will be established while the user application is already running which could prevent debugging of the start of the user application.

All debugger features are restricted to user code and user data. Attempts to access protected regions of the address space are ignored and have no effect. Restricted address spaces are for example the key storage, the Secured Software Container, and the cryptographic library.

4 API documentation

4.1 BSL commands

All BSL commands available are listed below, sorted into their respective NVM protection group.

The NVM protection group is checked before a BSL command is executed. An error is returned upon when an access violation occurs.

NVM protection group definitions:

- Group 1: For these commands, any protection is ignored.
- Group 2: These commands are blocked when NVM protection is active on the target segment.
- Group 3: These commands are blocked when NVM protection is active on any segment.

Note: "NVM protection" can be read or write protection.

100TP pages are considered part of a code segment. Write or read access to 100TP pages via BSL requires write or read protection, respectively validation on the code segment.

Table 7 NVM protection check for BSL commands

NVM protection group	BSL command
Group 1 Protection ignored	Cmd 0x86 Memory execute Cmd 0x98 NVM permanent protection clear Cmd 0x0C NVM verify Cmd 0x93 BSL baud rate set Cmd 0x92 Device reset
Group 2 Protection on target segment	Cmd 0x89 NVM permanent protection set Cmd 0x05 Memory write (NVM, PSRAM) Cmd 0x87 Memory read (NVM, PSRAM) Cmd 0x88 NVM erase (page, sector) Cmd 0x0D NVM 100TP write Cmd 0x8E NVM 100TP read Cmd 0x97 NVM 100TP erase
Group 3 Protection on any segment	Cmd 0x88 NVM erase (mass) Cmd 0x99 UBSL size set Cmd 0x9C UBSL privilege set

4.1.1 Cmd 0x86 Memory execute

The device provides a BSL command to execute code from PSRAM in user mode.

Table 8 Cmd 0x86 command frame

0	1	2	3	4
Message type	Address byte #0 (MSB)	Address byte #1	Address byte #2 (LSB)	Target

Table 9 "Cmd 0x86 - Memory execute" command frame parameter definition

Field	Description
Message type	Memory execute command. Always set to 0x86.
Address byte #0 (MSB)	24-bit memory address offset, pointing to the vector table address. The offset is based on the respective memory base address.
Address byte #1	
Address byte #2 (LSB)	
Target	0x10 (SRAM)

SRAM base address: 0x18000000

This BSL command rejects the operation if the provided vector address if out of range.

When targeting SRAM, the valid range is within PSRAM.

This BSL command performs clean-up before executing:

- Deinitializing the media configuration
- Clearing the timer
- Clearing the interrupt source, interrupt status, and NMI status
- Remapping the user vector table

Note: The command does not switch the system clock. Therefore, the target code will be executed with HPCLK.

Note: The command does not re-enable FS_WDT. This means that the device stays in the fail-safe state and the motor cannot run.

This BSL command executes the reset handler in the vector table. It rejects the operation and reports an error if the vector address is not 32-word aligned.

4.1.2 Cmd 0x98 NVM permanent protection clear

The device provides a BSL command to clear the protection of individual NVM segments..

Table 10 Cmd 0x98 command frame

0	1	2	3	4
Message type	Passbyte #3 (MSB)	Passbyte #2	Passbyte #1	Passbyte #0 (LSB)

Table 11 "Cmd 0x98 - NVM permanent protection clear" command frame parameter definition

Field	Description
Message type	NVM permanent protection clear command. Always set to 0x98.
Passbyte #3 (MSB) Passbyte #2 Passbyte #1 Passbyte #0 (LSB)	Passphrase (passbyte[3:0]) options: • UBSL segment passphrase: 0xBBXX5555 • Code segment passphrase: 0xCCXX5555 • Data segment passphrase: 0xDDXX5555 "XX" is ignored. Whether the command erases the segment does not depend on the value supplied here but on the preinstalled passphrase.

If the provided passphrase is invalid, the command rejects the operation and reports an error.

The command evaluates the preinstalled passphrase.

If the preinstalled passphrase does not contain an erase flag, the command clears the permanent protection of the target segment and its lower-privileged segments.

If the preinstalled passphrase does contain an erase flag, the command clears the permanent protection of the target segment and its lower-privileged segments and erases them. In this case, if the target segment is an UBSL segment, the BSL privilege setting is reset as well. Furthermore, if the target segment is an UBSL or UCODE segment, the security keys (except default key) are erased as well.

When the permanent protection is cleared both read- and write-protection are immediately removed from the segments.

4.1.3 Cmd 0x0C NVM verify

The device provides a BSL command to check the integrity of the flash memory.

Table 12 Cmd 0x0C command frame

0	1	2	3	4	5	6	7	8
Message type	Address byte #0 (MSB)	Address byte #1	Address byte #2 (LSB)	Target	Number of pages byte #0 (MSB)	Number of pages byte #1 (LSB)	Checksum byte #0 (MSB)	Checksum byte #1 (LSB)

Table 13 "Cmd 0x0C - NVM verify" command frame parameter definition

Field	Description
Message type	NVM verify command. Always set to 0x0C.
Address byte #0 (MSB)	24-bit memory address offset from where to start NVM data verification. The offset starts counting from the respective memory start address. NVM data get verified against incrementing memory addresses.
Address byte #1	
Address byte #2 (LSB)	
Target	FLASH0: 0x00, FLASH1: 0x01.
Number of pages, byte #0 (MSB)	16-bit number indicating the number of pages to be verified. The number must not exceed the number of NVM pages available in linear regions.
Number of pages, byte #1 (LSB)	
Checksum byte #0 (MSB)	User-provided 16-bit reference checksum.
Checksum byte #0 (MSB)	

FLASH0 base address: 0x11000000.

FLASH1 base address: 0x12002000.

The command is executed regardless of the NVM protection status.

The command will not be executed if targeted NVM range exceeds the overall linear NVM size.

4.1.4 Cmd 0x93 BSL baud rate set

The device provides a BSL command to change the BSL baud rate (for the CAN interface) in the current BSL session.

Table 14 Cmd 0x93 command frame

0	1	2
Message type	Baud rate option	Reserved

Table 15 "Cmd 0x93 - BSL baud rate set" command frame parameter definition

Field	Description
Message Type	Get chip ID command. Always set to 0x93.
Baud rate options	Baud rate options: 0x00: 500 kBd 0x01: 1 MBd 0x02: 1.25 MBd

The new baud rate takes effect immediately after the response has been sent back to the host. The response is still using the old baud rate.

4.1.5 Cmd 0x92 Device reset

This device provides a BSL command to reset the device. Executing this command makes the device exit any Default BSL communications and reboot. No response message is sent.

When the command executes successfully, it does not send back a SUCCESS response. Instead, it performs a cold reset.

The reset is triggered by re-enabling FS_WDT. When a WDT self-test is performed, the microcontroller is kept in reset during the test.

If the user wants to execute user application code after reset, the user needs to set a proper NAC value (for example, NAC=0x0) to avoid re-entering into BSL communications.

Table 16 Cmd 0x92 command frame

1
Message type

Table 17 "Cmd 0x92 - Device reset" command frame parameter definition

Field	Description
Message type	Device reset command. Always set to 0x92.

4.1.6 Cmd 0x89 NVM permanent protection set

The device provides a BSL command to set permanent protection on individual NVM segments.

Table 18 Cmd 0x89 command frame

0	1	2	3	4
Message type	Passbyte #3 (MSB)	Passbyte #2	Passbyte #1	Passbyte #0 (LSB)

Table 19 "Cmd 0x89 - NVM permanent protection set" command frame parameter definition

Field	Description
Message type	NVM permanent protection set command. Always set to 0x89.
Passbyte #3 (MSB) Passbyte #2 Passbyte #1 Passbyte #0 (LSB)	Passphrase (passbyte[3:0]) options: • UBSL segment passphrase without erase flag: 0xBB005555 • UBSL segment passphrase with erase flag: 0xBBFF5555 • Code segment passphrase without erase flag: 0xCC005555 • Code segment passphrase with erase flag: 0xCCFF5555 • Data segment passphrase without erase flag: 0xDD005555 • Data segment passphrase with erase flag: 0xDDFF5555 Note: for erase flag usage see also Cmd 0x98, no erase action here.

The BSL command to set permanent NVM protection is rejected if the protection of the segment with higher privileges is not set.

If the specified NVM protection passphrase is valid, the command installs the passphrase into the device to set permanent protection on the target segment.

When the permanent protection is set, the segment is immediately both read- and write-protected.

Important note:

Whether the protection passphrase includes the erase flag influences the scope of the FAR analysis.

If the protection passphrase with the erase flag is installed, clearing the permanent protection erases the affected segments. FAR analyses of user code flash failures (for example, ECC error, mapping error) is not possible.

If a protection passphrase without the erase flag is installed, clearing the permanent protection leaves the user code in place. FAR analyses of user code flash failures are possible.

4.1.7 Cmd 0x05 Memory write

The device provides a Default BSL command to write to NVM and RAM.

Table 20 Cmd 0x05 command frame

0	1	2	3	4	5 .. 132
Message type	Address byte #0 (MSB)	Address byte #1	Address byte #2 (LSB)	Target	Data

Table 21 "Cmd 0x05 - Memory write" command frame parameter definition

Field	Description
Message type	Memory write command. Always set to 0x05.
Address byte #0(MSB)	24-bit memory address offset where to start storing the download data. The offset starts counting from the respective memory base address. Data gets written at incrementing memory addresses.
Address byte #1	
Address byte #2(LSB)	
Target	FLASH0: 0x00, FLASH1: 0x01, SRAM: 0x10.
Data	8-bit data bytes to be written, minimum size 1 byte, maximum size 128 bytes.
Checksum	Checksum from length byte of the media frame to the end of data, excluding the checksum byte.

The BSL command to write to an NVM segment is blocked if a protection is set on the target NVM segment.

The BSL command to write to PSRAM is blocked if a protection is set on a code segment.

Writing to DSRAM is possible regardless of the protection settings.

This command does not support writing across page boundaries. It rejects the page write operation if the offset is out of range or the offset plus length of the data is out of range. It returns an error code in the response message.

Memory write supports partial non-page-aligned writing, preserving the page data not passed as an input.

Memory write supports writing a minimum of 1 byte and a maximum of 128 bytes.

The command rejects the operation and reports an error if asked to access secure RAM.

4.1.8 Cmd 0x87 Memory read

The device provides a BSL command to read NVM and RAM.

Table 22 Cmd 0x87 command frame

0	1	2	3	4	5	6
Message type	Address byte #0 (MSB)	Address byte #1	Address byte #2 (LSB)	Target	Count	Reserved

Table 23 "Cmd 0x87 - Memory read" command frame parameter definition

Field	Description
Message type	Memory read command. Always set to 0x87.
Address byte #0(MSB)	24-bit memory address offset where to start reading data. The offset starts counting from the respective memory start address. Data gets read at incrementing memory addresses.
Address byte #1	
Address byte #2(LSB)	
Target	FLASH0: 0x00, FLASH1: 0x01, SRAM: 0x10, CFS0: 0x20.
Count	Number of 8-bit data bytes to be read, minimum size 1 byte, maximum size 128 bytes.

FLASH0 base address: 0x11000000.
FLASH1 base address: 0x12002000.
SRAM base address: 0x18000000.

This BSL command supports reading of up to 128 bytes of data.
This BSL command rejects the operation if the target address if out of range.
The BSL command to read memory does not support reading across page boundaries.
The command rejects the operation if it attempts to read a secure RAM location.
Reading of PSRAM is rejected if the NVM code segment is protected.
Reading of DSRAM is allowed regardless of any segment protection.
Reading of flash is rejected if the target NVM segment is protected.
The command supports read access to CFS0 in test mode. It rejects the operation and reports an error if any segment is read-protected.

4.1.9 Cmd 0x88 NVM erase

The device provides a BSL command to erase an NVM page, NVM sector, or NVM module.

Table 24 Cmd 0x88 command frame

0	1	2	3	4	5	6
Message type	Address byte #0 (MSB)	Address byte #1	Address byte #2 (LSB)	Target	Erase type	Reserved

Table 25 "Cmd 0x88 - NVM erase" Command frame parameter definition

Field	Description
Message type	NVM erase command. Always set to 0x88.
Address byte #0 (MSB)	24-bit NVM address offset for page, sector, or module to erase. The offset is based on the NVM start address.
Address byte #1	
Address byte #2 (LSB)	
Target	FLASH0: 0x00, FLASH1: 0x01
Erase type	Supported erase type field values: 0 - NVM page erase 1 - NVM sector erase 2 - NVM module mass erase

FLASH0 base address: 0x11000000.
FLASH1 base address: 0x12002000.
When erasing a page or sector, the BSL command rejects the operation if the target address is out of range.
When erasing a page or sector, the BSL command rejects the operation if the target NVM segment is protected.
When erasing a module, the BSL command rejects the operation if any segment in the module is protected.

4.1.10 Cmd 0x0D NVM 100TP write

The device provide a BSL command to write 100TP pages.

Table 26 Cmd 0x0D command frame

0	1	2	3	4	5 .. 128
Message type	Page index	Page offset	Reserved	Counter value	Data

Table 27 "Cmd 0x0D - NVM 100TP write" command frame parameter definition

Field	Description
Message type	NVM 100TP write command. Always set to 0x0D.
Page index	100TP page selector, valid range 0..7.
Page offset	Offset within selected page.
Counter value	New 100TP counter value.
Data	8-bit data bytes to be written, minimum size 1 byte, maximum size 124 bytes.

The command supports setting the page write counter value. If the specified counter value is larger than the current counter value, and is smaller than or equal to 100, the specified counter value is written. If the specified counter value is larger than 100, it is truncated to 100. If the specified counter value is smaller than or equal to the current value, the current counter value is incremented by one.

The command supports partial page writing operations, with the selected byte offset and number of bytes.

The command does not support writing across page boundaries.

The command rejects the operation if the page write counter value already has reached 100. (The initial counter value is 0xFF. After the first write, the counter value is incremented to 1, after the second write to 2, ..., after the 100th write, the counter value is incremented to 100.)

4.1.11 Cmd 0x8E NVM 100TP read

The device provides a BSL command to read 100TP pages.

Table 28 Cmd 0x8E command frame

0	1	2	3	4
Message type	Page index	Page offset	Reserved	Count

Table 29 "Cmd 0x8E – NVM 100TP read" command frame parameter definition

Field	Description
Message type	NVM 100TP read command. Always set to 0x8E.
Page index	100TP page selector, valid range 0..7.
Page offset	Offset within selected page, valid range 0..127.
Count	Number of 8-bit data bytes to be read, minimum size 1 byte, maximum size 128 bytes.

The command does not support reading across page boundaries.

The command performs an ECC2 check on the target page and reports an error when that check fails.

4.1.12 Cmd 0x97 NVM 100TP erase

The device provides a BSL command to erase 100TP pages.

Table 30 Cmd 0x97 command frame

0	1	2
Message type	Page index	Reserved

Table 31 "Cmd 0x97 – NVM 100TP erase" command frame parameter definition

Field	Description
Message type	NVM 100TP erase command. Always set to 0x97.
Page index	100TP page selector, valid range 0..7.

The command preserves the 100TP counter value.

The command sets the counter value to 95 if the page contains an ECC2DATA error.

The command invalidates the 100TP page by writing an invalid checksum to ensure that the erased page is not used. Writing to the erased 100TP page will make it valid again.

4.1.13 Cmd 0x99 UBSL size set

The device provides a BSL command to write the size of the USBL into the configuration sector.

Table 32 Cmd 0x99 command frame

0	1	2
Message type	UBSL size	Reserved

Table 33 "Cmd 0x99 - UBSL size set" command frame parameter definition

Field	Description
Message type	UBSL size set command. Always set to 0x99.
UBSL size	Possible UBSL size options: 0x00: 4 kB 0x01: 8 kB 0x02: 12 kB 0x03: 16 kB 0x04: 20 kB 0x05: 24 kB 0x06: 28 kB (default) 0x07: 32 kB

The command can be executed only once. It rejects the operation and reports an error if it has been executed before or if the UBSL location contains an ECC2 error.

The command rejects the operation if any NVM segment is protected.

The command rejects the operation if the UBSL size parameter is invalid.

Note: If this command is called, the user must also adapt the size of UBSL area settings in the used tool chain, for example in the Linker file.

4.1.14 Cmd 0x9C UBSL privilege set

The device provides a BSL command to change the UBSL privilege settings in the configuration sector. Calling this command sets the default privilege level (4) of the UBSL to be equal to that of UCODE (3). The privilege level of the UBSL cannot be changed back later.

Table 34 Cmd 0x9C command frame

0
Message type

Table 35 "Cmd 0x9C - UBSL privilege set" command frame parameter definition

Field	Description
Message type	UBSL privilege set command. Always set to 0x9C.

The command can be executed only once. It rejects the operation if it has been executed before or if the UBSL location contains an ECC2 error.

The command rejects the operation if any NVM segment is protected.

4.1.15 Resp 0x80 Data response

Some BSL commands request data from the device. These messages expect a data response message.

Table 36 Cmd 0x80 response message frame

0	1 .. 128
Message type	Data

Table 37 "Resp 0x80 - Data response" response frame parameter definition

Field	Description
Message type	Data response, always set to 0x80.
Data	The requested data, minimum size 1 byte, maximum size 128 bytes.

4.1.16 Resp 0x81 Acknowledge response

The device sends back an acknowledge response message if the command does not request any data or if the BSL command fails.

Table 38 Cmd 0x81 response message frame

0	1	2
Message type	Response code byte #0 (MSB)	Response code byte #1 (LSB)

Table 39 "Resp 0x81 - Acknowledge response" response frame parameter definition

Field	Description
Message type	Acknowledge response. Always set to 0x81.
Response code byte #0 (MSB)	Signed 16-bit command response code. The value is set to zero if the requested command was executed successfully. Otherwise, the response code is an error code.
Response code byte #1 (LSB)	

4.2 User API routines

These routines are exposed by the BootROM to the customer user mode software.

User API routines support flash access and protection configuration. Security variants also support cryptographic operations.

Table 40 User API routines function overview

Name	Description
user_nvm_service_algorithm	This user API function runs the service algorithm on a mapped sector, attempting to repair faulty pages or double mappings.
user_nvm_mapram_recover	This user API function attempts to reconstruct map RAM by extracting mapping information from good pages.
user_nvm_mapram_init	This user API function triggers the map RAM initialization on the target mapped sector.
user_cid_get	This user API function gets the customer identification number.
user_nvm_ecc_check	This user API function checks for single and double ECC errors on the target flash.
user_nvm_ecc_addr_get	This user API function returns the address of a double ECC event that has occurred in the target flash.
user_nvm_100tp_read	This user API function reads data from a specified 100TP page.
user_nvm_100tp_write	This user API function writes data to a specified 100TP page.
user_nvm_100tp_erase	This user API function erases a data field of the specified 100TP page.
user_nvm_config_get	This user API function returns the size of each NVM segment.
user_nvm_temp_protect_get	This user API function gets the current protection status of a specified NVM segment.
user_nvm_udata_temp_protect_set	This user API function temporarily sets the write protection of the UDATA segment.
user_nvm_ucode_temp_protect_set	This user API function temporarily sets the write protection of the UCODE segment.
user_nvm_ubsl_temp_protect_set	This user API function temporarily sets the write protection of the UBSL segment.
user_nvm_udata_temp_protect_clear	This user API function temporarily clears the write protection of the UDATA segment.
user_nvm_ucode_temp_protect_clear	This user API function temporarily clears the write protection of the UCODE segment.
user_nvm_ubsl_temp_protect_clear	This user API function temporarily clears the write protection of the UBSL segment.
user_nvm_page_erase	This user API function erases a specified flash page.
user_nvm_sector_erase	This user API function erases a specified flash sector.
user_nvm_page_write	This user API function writes a number of bytes from the source to the specified flash address.

(table continues...)

Table 40 (continued) User API routines function overview

Name	Description
user_ram_mbist	This user API function performs a MBIST on the specified SRAM range.
user_crypto_aes_cmac_generate_start	This user API function initializes a CMAC generation.
user_crypto_aes_cmac_generate_update	This user API function updates the ongoing CMAC generation.
user_crypto_aes_cmac_generate_finish	This user API function finalizes the ongoing CMAC generation.
user_crypto_aes_cmac_verify_start	This user API function initializes a CMAC verification operation.
user_crypto_aes_cmac_verify_update	This user API function updates the ongoing CMAC verification.
user_crypto_aes_cmac_verify_finish	This user API function finalizes the ongoing CMAC verification.
user_crypto_aes_start	This user API function initializes an AES operation.
user_crypto_aes_update	This user API function updates the ongoing AES operation.
user_crypto_aes_finish	This user API function finalizes the ongoing AES operation.
user_crypto_key_write	This user API function writes a cryptographic key to the target key slot.
user_crypto_key_erase	This user API function erases a cryptographic key.
user_crypto_key_verify	This user API function verifies and optionally repairs an existing cryptographic key.
user_nvm_isr_handler	The NVM read-while-write interrupt handler.
user_secure_download_start	This user API function initializes the secure container and starts the secure download process.
user_secure_download_update	This user API function continues the secure download process.
user_secure_download_finish	This user API function finalizes the secure download process.
user_cache_operation	This user API function provides an alternative to writing to cache registers in addition to direct register access.
user_secure_dualboot	This user API function configures and enables the secondary UBSL image.
user_ubsl_size_restore	This user API function is used to restore the UBSL size in case of a Stop mode reset.
user_nvm_perm_protect_set	This user API function sets permanent protection on NVM segments.

4.2.1 user_nvm_service_algorithm

Description

This user API function runs the service algorithm on a mapped sector, attempting to repair faulty pages or double mappings.

Prototype

```
int32_t user_nvm_service_algorithm (
  uint32_t sector_address
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	sector_address	Address of the sector on which to run the service algorithm (SA).	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_SEGMENT_PROTECTED #ERR_LOG_CODE_MEM_ADDR_RANGE_INVALID #ERR_LOG_CODE_SA_UNRECOVERABLE

Stack Usage

The execution of this API function has a maximum stack usage of 16 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.2 user_nvm_mapram_recover

Description

This user API function attempts to reconstruct map RAM by extracting mapping information from good pages.

It can be called by the user if the NVM service algorithm (SA) fails to repair a corrupted data map sector. Requests to initialize the map RAM for an unavailable sector or for a linearly mapped sector are ignored. Pages that are mapped two or more times are counted as faulty pages.

Prototype

```
int32_t user_nvm_mapram_recover (
  uint32_t sector_address
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	sector_address	Address of the sector from which to recover mapping information.	-

Return values

Data type	Description
int32_t	Function execution status. Non-negative after successful execution, indicating the amount of good mapped pages that were found. #ERR_LOG_CODE_MEM_ADDR_RANGE_INVALID #ERR_LOG_CODE_NVM_ECC2_MAPRAM_ERROR #ERR_LOG_CODE_SEMAPHORE_RESERVED

Stack Usage

The execution of this API function has a maximum stack usage of 216 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.3 user_nvm_mapram_init

Description

This user API function triggers the map RAM initialization on the target mapped sector.

Prototype

```
int32_t user_nvm_mapram_init (
  uint32_t sector_address
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	sector_address	Sector address to perform operation.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_MEM_ADDR_RANGE_INVALID #ERR_LOG_CODE_MAPRAM_INIT_PAGE_FAIL #ERR_LOG_CODE_MAPRAM_INIT_DM_PAGE_FAIL

Stack Usage

The execution of this API function has a maximum stack usage of 216 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.4 user_cid_get

Description

This user API function gets the customer identification number. It contains information about the variant and design step.

Prototype

```
int32_t user_cid_get (
  uint32_t * customer_id
)
```

Parameters

Data type	Name	Description	Dir
uint32_t *	customer_id	Pointer where to store the customer identification number (CID) read from the device configuration sector. The address indicated by the pointer must be located in RAM.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_POINTER_RAM_RANGE_INVALID

Customer ID

The customer ID consists of four bytes with a specific meaning (see Table 41).

Table 41 Customer ID encoding

Byte 0 Grade		Byte 1 Design Step		Byte 2 Package, Variant		Byte 3 Family	
Grade 0	20_H	AA-Step	AA_H	48-pin	$X7_H$	TLE988x	06_H
Grade 1	00_H	AB-Step	AB_H	64-pin	XB_H	TLE989x	07_H
		AK-Step	BA_H	TLE98x1	$1X_H$		
				TLE98x3	$3X_H$		

Note: An 'X' within a hexadecimal value represents a "dont'care" position.

Stack Usage

The execution of this API function has a maximum stack usage of 12 bytes.

4.2.5 user_nvm_ecc_check

Description

This user API function checks for single and double ECC errors on the target flash.

All ECC error flags are cleared before the check starts to prevent the reading of previously set error flags. Upon exit, the function clears the current ECC status.

Prototype

```
int32_t user_nvm_ecc_check (
  uint32_t flash
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	flash	Target flash (see Constant reference).	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS: No single or double ECC events have occurred. #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_ECC1READ_ERROR #ERR_LOG_CODE_ECC2READ_ERROR #ERR_LOG_CODE_PARAM_INVALID

Stack Usage

The execution of this API function has a maximum stack usage of 96 bytes.

Remarks

This routine does not provide the addresses of the ECC errors. In case an ECC error is detected, call the user_nvm_ecc_addr_get routine to retrieve the failure address.

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.6 user_nvm_ecc_addr_get

Description

This user API function returns the address of a double ECC event that has occurred in the target flash.
The value of pNVM_Addr can be one of below patterns:

- 0x11XXXXXX ECC2 failure in FLASH0 area, it indicates the absolute memory address.
- 0x12XXXXXX ECC2 failure in FLASH1 area, it indicates the absolute memory address.
- 0x100000XY ECC2 in 100TP pages, where X = 100TP page number, Y = block offset inside the page (block granularity: 8 bytes).
- 0x01000000 ECC2 in internal NVM CS area, not recoverable.

Prototype

```
int32_t user_nvm_ecc_addr_get (
  uint32_t flash
  uint32_t * pNVM_Addr
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	flash	Target flash (see Constant reference).	-
uint32_t *	pNVM_Addr	Pointing to ECC2 failure address.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS: No ECC2 event or event address has been obtained successfully. #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_USER_POINTER_RAM_RANGE_INVALID #ERR_LOG_CODE_PARAM_INVALID

Stack Usage

The execution of this API function has a maximum stack usage of 52 bytes.

Remarks

When the function exits, it clears the current ECC2 flag.
Any other NVM operations also clear the ECC2 flag.
In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.7 user_nvm_100tp_read

Description

This user API function reads data from a specified 100TP page.
This function can read a maximum of 128 bytes (including the counter field and checksum field).

Prototype

```
int32_t user_nvm_100tp_read (
  uint32_t npage
  user_100tp_read_t * params
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	npage	The index of the page from which to read. Valid range: 0 to 7.	-
user_100tp_read_t *	params	100TP read parameters.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_POINTER_RAM_RANGE_INVALID #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_PARAM_INVALID #ERR_LOG_CODE_100TP_PAGE_INVALID #ERR_LOG_CODE_ECC2READ_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 48 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.8 user_nvm_100tp_write

Description

This user API function writes data to a specified 100TP page.

The function can write up to 124 bytes in a data field each time. The function supports maximum 100 times write operation. The function performs an implicit update of the page checksum.

Prototype

```
int32_t user_nvm_100tp_write (
  uint32_t npage
  user_100tp_write_t * params
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	npage	The index of the page to which to write. Valid range: 0 to 7.	-
user_100tp_write_t *	params	100TP write parameters.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_POINTER_RAM_RANGE_INVALID #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_PARAM_INVALID #ERR_LOG_CODE_100TP_PAGE_INVALID #ERR_LOG_CODE_SEGMENT_PROTECTED #ERR_LOG_CODE_ECC2READ_ERROR #ERR_LOG_CODE_100TP_WRITE_COUNT_EXCEEDED #ERR_LOG_CODE_ACCESS_AB_MODE_ERROR #ERR_LOG_CODE_NVM_ECC2_DATA_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 88 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.9 user_nvm_100tp_erase

Description

This user API function erases the specified 100TP page. The write counter field is preserved.

The function should be called if the 100TP page is corrupted. Upon successful execution, the page is initialized with an invalid checksum.

Prototype

```
int32_t user_nvm_100tp_erase (
  uint32_t npage
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	npage	The index of the 100TP page to erase. Valid range: 0 to 7.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_100TP_PAGE_INVALID #ERR_LOG_CODE_SEGMENT_PROTECTED #ERR_LOG_CODE_ACCESS_AB_MODE_ERROR #ERR_LOG_CODE_NVM_VER_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 72 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.10 user_nvm_config_get

Description

This user API function returns the size of the UBSL, UCODE, and UDATA NVM segments.

Prototype

```
int32_t user_nvm_config_get (
  uint32_t * ubsl_nvm_size
  uint32_t * code_nvm_size
  uint32_t * data_nvm_size
)
```

Parameters

Data type	Name	Description	Dir
uint32_t *	ubsl_nvm_size	Pointer to where to store the retrieved NVM UBSL segment size.	-
uint32_t *	code_nvm_size	Pointer to where to store the retrieved NVM UCODE segment size.	-
uint32_t *	data_nvm_size	Pointer to where to store the retrieved NVM UDATA segment size.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_POINTER_RAM_RANGE_INVALID

Stack Usage

The execution of this API function has a maximum stack usage of 36 bytes.

4.2.11 user_nvm_temp_protect_get

Description

This user API function gets the current protection status of the specified NVM segment.

Prototype

```
uint32_t user_nvm_temp_protect_get (
  user_nvm_segment_t segment
)
```

Parameters

Data type	Name	Description	Dir
user_nvm_segment_t	segment	The NVM segment for which to report the current protection status.	-

Return values

Data type	Description
uint32_t	The current protection status of the specified NVM segment.

Stack Usage

The execution of this API function has a maximum stack usage of 16 bytes.

4.2.12 user_nvm_udata_temp_protect_set

Description

This user API function temporarily sets the write protection of the User data NVM segment until the protection is removed by calling user_nvm_udata_temp_protect_clear.

Prototype

```
int32_t user_nvm_udata_temp_protect_set (
  uint32_t passphrase
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	passphrase	The passphrase must be NVM_SEG_PROT_DATA_NO_ERASE.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_PROTECT_WRONG_PASSWORD #ERR_LOG_CODE_SEMAPHORE_RESERVED

Stack Usage

The execution of this API function has a maximum stack usage of 16 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.13 user_nvm_ucode_temp_protect_set

Description

This user API function temporarily sets the write protection of the User code NVM segment until the protection is removed by calling user_nvm_ucode_temp_protect_clear.

Prototype

```
int32_t user_nvm_ucode_temp_protect_set (
  uint32_t passphrase
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	passphrase	The passphrase must be NVM_SEG_PROT_CODE_NO_ERASE.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_PROTECT_WRONG_PASSWORD #ERR_LOG_CODE_SEMAPHORE_RESERVED

Stack Usage

The execution of this API function has a maximum stack usage of 16 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.14 user_nvm_ubsl_temp_protect_set

Description

This user API function temporarily sets the write protection of the User BSL NVM segment until the protection is removed by calling user_nvm_ubsl_temp_protect_clear.

Prototype

```
int32_t user_nvm_ubsl_temp_protect_set (
  uint32_t passphrase
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	passphrase	The passphrase must be NVM_SEG_PROT_UBSL_NO_ERASE.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS, #ERR_LOG_CODE_USER_PROTECT_WRONG_PASSWORD #ERR_LOG_CODE_SEMAPHORE_RESERVED

Stack Usage

The execution of this API function has a maximum stack usage of 16 bytes.

Remarks

This functional is callable only from UBSL segment.
In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.15 user_nvm_udata_temp_protect_clear

Description

This user API function temporarily clears the write protection of the User data NVM segment after enabling the protection by calling user_nvm_udata_temp_protect_set.

Prototype

```
int32_t user_nvm_udata_temp_protect_clear (
  uint32_t passphrase
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	passphrase	The passphrase must be NVM_SEG_PROT_DATA_NO_ERASE.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_PROTECT_WRONG_PASSWORD #ERR_LOG_CODE_SEMAPHORE_RESERVED

Stack Usage

The execution of this API function has a maximum stack usage of 16 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.16 user_nvm_ucode_temp_protect_clear

Description

This user API function temporarily clears the write protection of the User code NVM segment after enabling the protection by calling user_nvm_ucode_temp_protect_set.

Prototype

```
int32_t user_nvm_ucode_temp_protect_clear (
  uint32_t passphrase
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	passphrase	The passphrase must be NVM_SEG_PROT_CODE_NO_ERASE.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_PROTECT_WRONG_PASSWORD #ERR_LOG_CODE_SEMAPHORE_RESERVED

Stack Usage

The execution of this API function has a maximum stack usage of 16 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.17 user_nvm_ubsl_temp_protect_clear

Description

This user API function temporarily clears the write protection of the User BSL NVM segment after enabling the protection by calling user_nvm_ubsl_temp_protect_set.

Prototype

```
int32_t user_nvm_ubsl_temp_protect_clear (
  uint32_t passphrase
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	passphrase	The passphrase must be NVM_SEG_PROT_UBSL_NO_ERASE.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_PROTECT_WRONG_PASSWORD #ERR_LOG_CODE_SEMAPHORE_RESERVED

Stack Usage

The execution of this API function has a maximum stack usage of 16 bytes.

Remarks

This functional is callable only from UBSL segment.
In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.18 user_nvm_page_erase

Description

This user API function erases a specified flash page.

When asked to erase an unused (new) page in a mapped sector, the function does nothing and returns success. When asked to erase a page in a linear sector, the function always performs the erase.

Prototype

```
int32_t user_nvm_page_erase (
  uint32_t page_address
  uint32_t options
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	page_address	Address of the NVM page to erase. Non-aligned addresses are accepted.	-
uint32_t	options	Page erase options. Supported options: • NVM_OPTIONS_NONE: Background read-while-write (RWW) enabled. • NVM_OPTIONS_RWW_DISABLE: Background read-while-write (RWW) disabled.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS.

Stack Usage

The execution of this API function has a maximum stack usage of 152 bytes.

Execution Time

The execution time of this API function is composed by the time needed to execute the code and the time needed by the flash operation. It further depends on the RWW setting. The timing behavior in case RWW is enabled is depicted below.

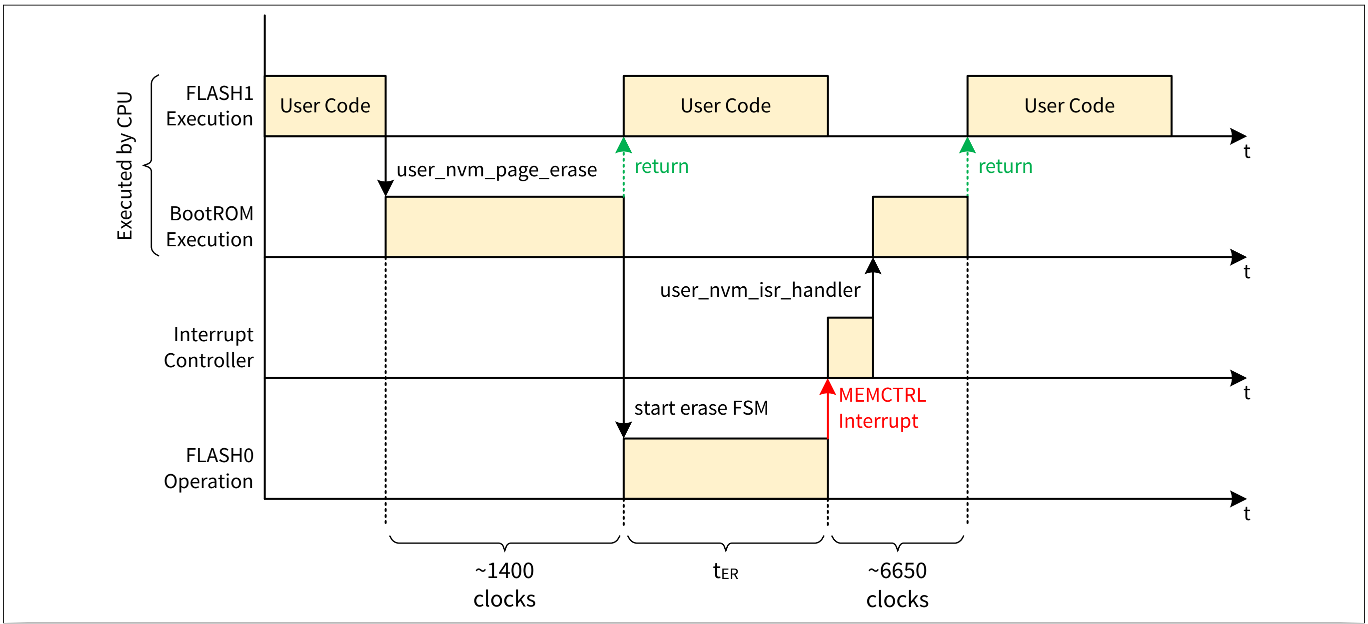

Figure 8 **Erasing a mapped page with RWW enabled.**

Remarks

The status of background read-while-write (RWW) is available in the NVM_OP_STS register within MEMCTRL. It can be accessed by MEMCTRL->NVM_OP_STS.

The result of background read-while-write (RWW) is available in the NVM_OP_RESULT register within MEMCTRL. It can be accessed by MEMCTRL->NVM_OP_RESULT. For the result encoding see Table 45.

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.19 user_nvm_sector_erase

Description

This user API function erases a specified flash sector.
For a mapped sector, upon successful sector erase, the map RAM is initialized and a new spare page is selected.

Prototype

```
int32_t user_nvm_sector_erase (
  uint32_t sector_address
  uint32_t options
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	sector_address	Address of the NVM sector to erase. Non-aligned addresses are accepted.	-
uint32_t	options	Sector erase options. Supported options: • NVM_OPTIONS_NONE: Background read-while-write (RWW) enabled. • NVM_OPTIONS_RWW_DISABLE: Background read-while-write (RWW) disabled.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS.

Stack Usage

The execution of this API function has a maximum stack usage of 152 bytes.

Execution Time

The execution time of this API function is composed by the time needed to execute the code and the time needed by the flash operation. It further depends on the RWW setting. The timing behavior in case RWW is enabled is depicted below.

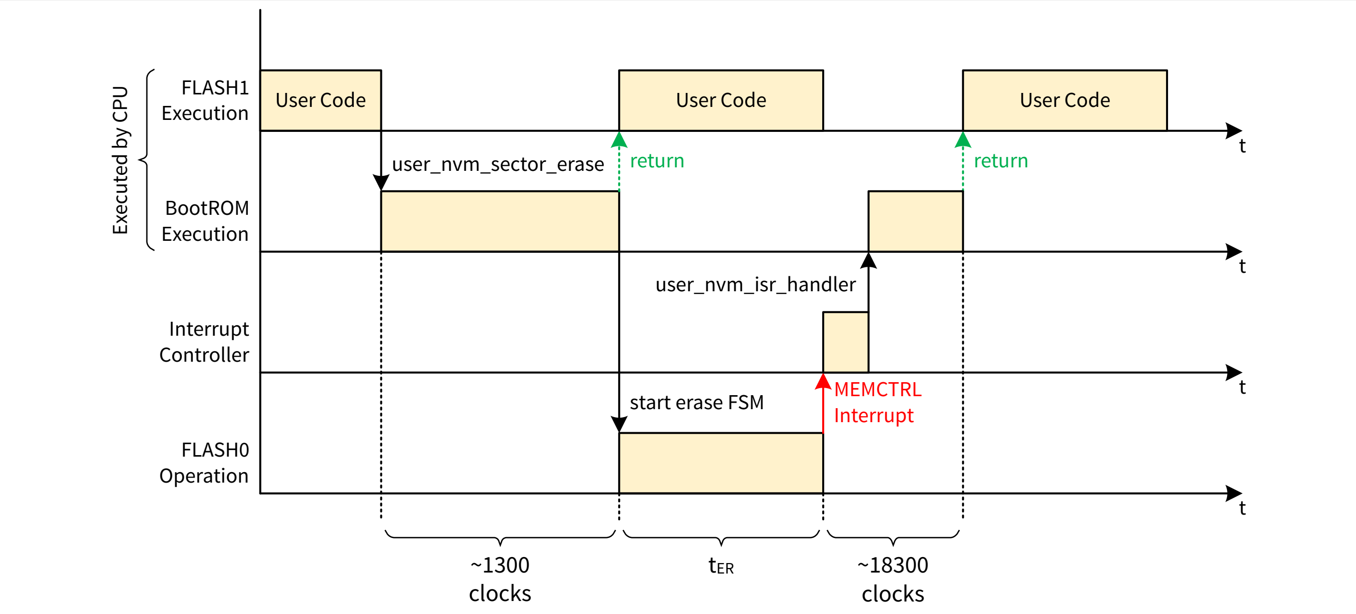

Figure 9 **Erasing a mapped sector with RWW enabled.**

Remarks

The status of background read-while-write (RWW) is available in the NVM_OP_STS register within MEMCTRL. It can be accessed by MEMCTRL->NVM_OP_STS.

The result of background read-while-write (RWW) is available in the NVM_OP_RESULT register within MEMCTRL. It can be accessed by MEMCTRL->NVM_OP_RESULT. For the result encoding see Table 45.

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.20 user_nvm_page_write

Description

This user API function writes a number of bytes to the specified flash address.

Prototype

```
int32_t user_nvm_page_write (
  uint32_t page_address
  user_nvm_page_write_t * params
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	page_address	The address of the NVM page to which to write the data.	-
user_nvm_page_write_t *	params	NVM write parameters. Supported parameter options: • NVM_OPTIONS_NONE: Corrective action disabled, RWW enabled, failpage erase enabled • NVM_OPTIONS_CORR_ACT: Enables retrying the write operation if the first write operation verification failed. For EEPROM specific, it enables disturb handling, which refreshes other pages in the background for around every 1K write • NVM_OPTIONS_NO_FAILPAGE_ERASE: This option applies only to mapped sectors. If it is specified, the failed written page remains. If it is not specified, the failed written page gets erased • NVM_OPTIONS_RWW_DISABLE: Background read-while-write (RWW) disabled.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS.

Stack Usage

The execution of this API function has a maximum stack usage of 200 bytes.

Execution Time

The execution time of this API function is composed by the time needed to execute the code and the time needed by the flash operation. It further depends on the RWW setting and preconditions of the page to be written. The different scenarios in case RWW is enabled are depicted below.

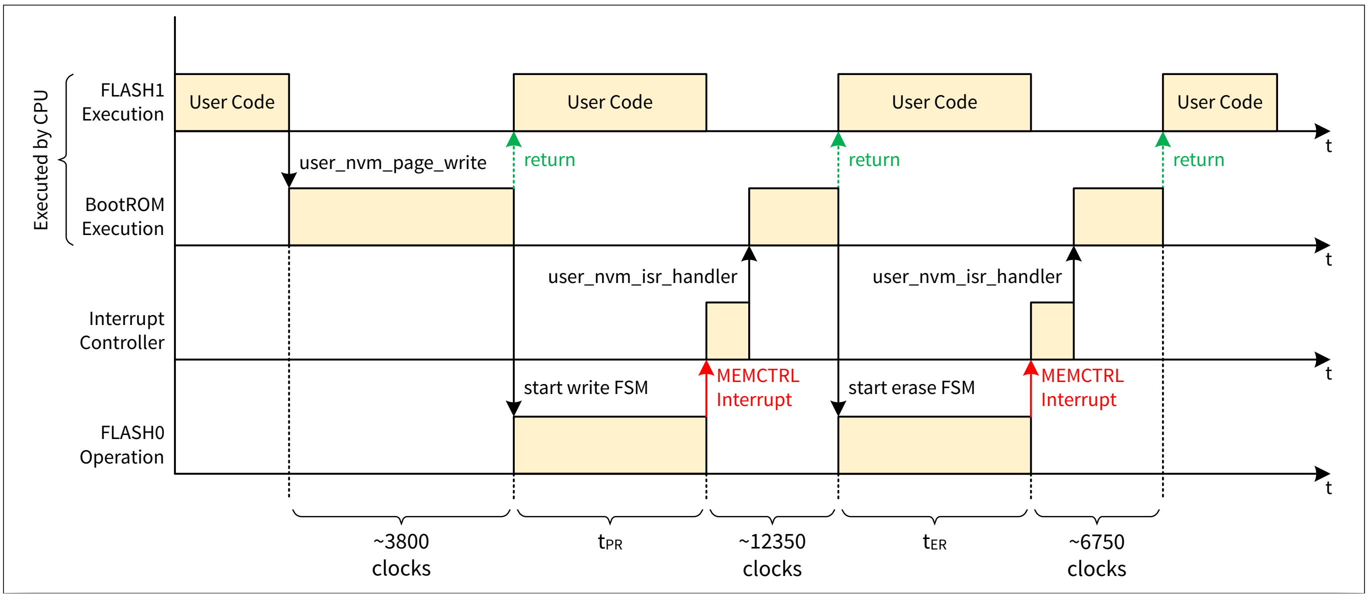

Figure 10 **Writing a mapped page with RWW enabled.**

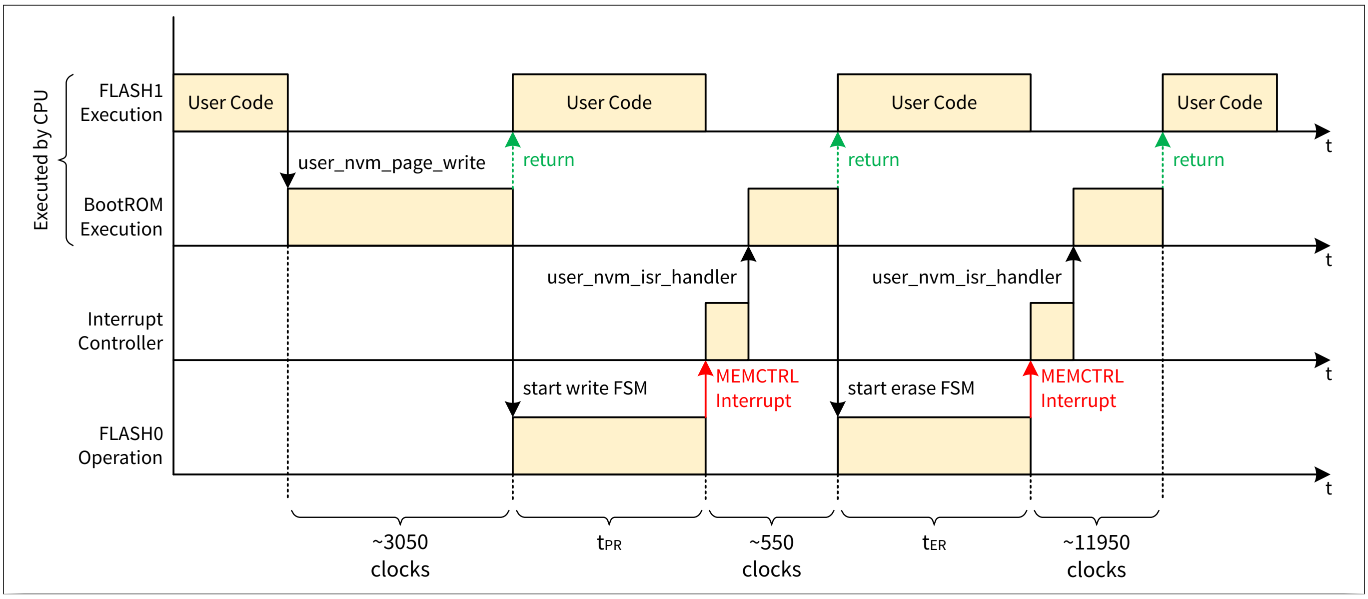

Figure 11 **Writing a programmed linear page with RWW enabled.**

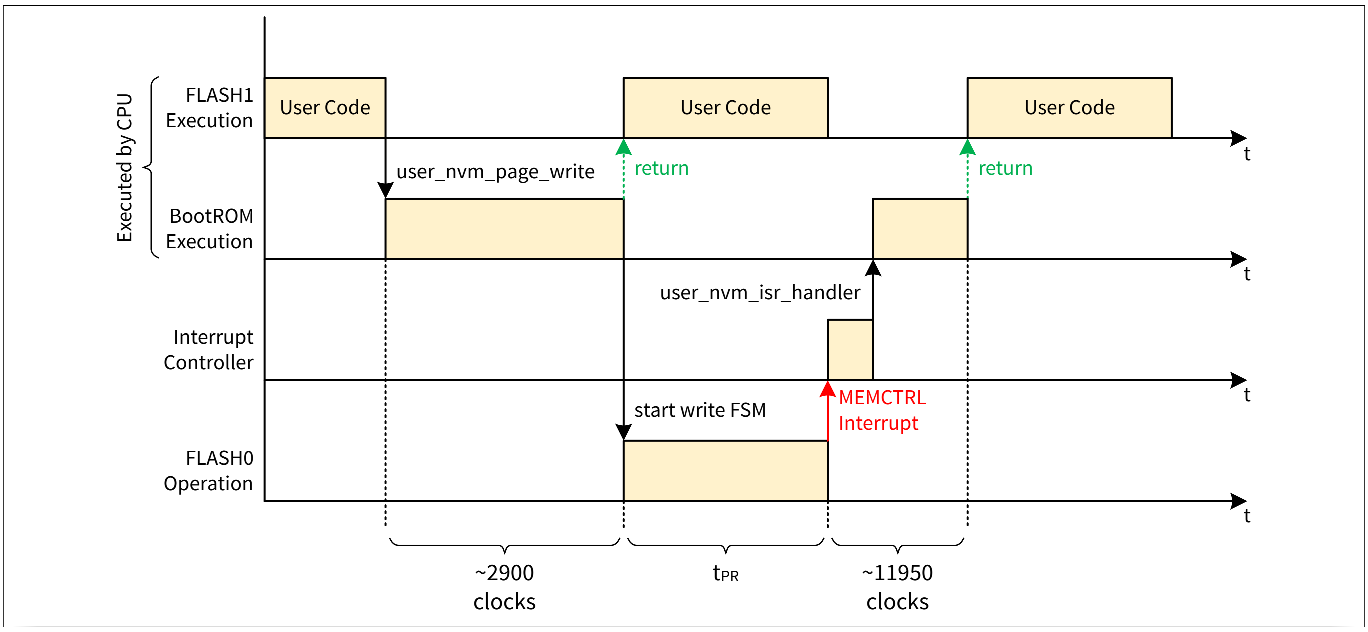

Figure 12 **Writing an erased linear page with RWW enabled.**

Remarks

The status of background read-while-write (RWW) is available in the NVM_OP_STS register within MEMCTRL. It can be accessed by MEMCTRL->NVM_OP_STS.

The result of background read-while-write (RWW) is available in the NVM_OP_RESULT register within MEMCTRL. It can be accessed by MEMCTRL->NVM_OP_RESULT. For the result encoding see Table 45.

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.21 user_ram_mbist

Description

This user API function performs an MBIST on the specified SRAM range. The value for start_address has to be smaller than end_address.

Prototype

```
int32_t user_ram_mbist (
  uint32_t start_address
  uint32_t end_address
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	start_address	RAM memory address at which to start the MBIST test. Highest valid address is 0x18000000 + device RAM size.	-
uint32_t	end_address	RAM memory address up to which to perform the MBIST test. Highest valid address is 0x18000000 + device RAM size.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_MBIST_RAM_RANGE_INVALID #ERR_LOG_CODE_MBIST_FAILED

Stack Usage

The execution of this API function has a maximum stack usage of 456 bytes.

Remarks

The execution of MBIST changes the RAM content in the specified address range. Make sure that the user stack does not get destroyed.

This function is not interruptible. Interrupts must be disabled before the call and only re-enabled after it has finished.

4.2.22 user_crypto_aes_cmac_generate_start

Description

This user API function initializes a CMAC generation.

Prototype

```
int32_t user_crypto_aes_cmac_generate_start (
  uint32_t key_id
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	key_id	Key ID used for CMAC generation. Key ID range is 1 to 12.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_PARAM_INVALID #ERR_LOG_CODE_KEY_SLOT_CORRUPTED #ERR_LOG_CODE_AES_UNSUPPORTED_ERROR #ERR_LOG_CODE_AES_BUFFER_SMALL_ERROR #ERR_LOG_CODE_AES_UNAVAILABLE_ERROR #ERR_LOG_CODE_AES_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 116 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.23 user_crypto_aes_cmac_generate_update

Description

This user API function updates the ongoing CMAC generation.
Call user_crypto_aes_cmac_generate_start routine before the first update operation. The function can be called multiple times.

Prototype

```
int32_t user_crypto_aes_cmac_generate_update (
  user_crypto_inp_buf_t * buf
)
```

Parameters

Data type	Name	Description	Dir
user_crypto_inp_buf_t *	buf	Input buffer for crypto operation.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_POINTER_RAM_RANGE_INVALID #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_AES_UNSUPPORTED_ERROR #ERR_LOG_CODE_AES_BUFFER_SMALL_ERROR #ERR_LOG_CODE_AES_UNAVAILABLE_ERROR #ERR_LOG_CODE_AES_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 176 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.24 user_crypto_aes_cmac_generate_finish

Description

This user API function finalizes the ongoing CMAC generation.
It concludes the entire CMAC generation operation and clears the cryptographic context from the reserved secure RAM.

Prototype

```
int32_t user_crypto_aes_cmac_generate_finish (
  user_crypto_io_buf_t * buf
  bool truncation_allowed
)
```

Parameters

Data type	Name	Description	Dir
user_crypto_io_buf_t *	buf	Output buffer for crypto operation.	-
bool	truncation_allowed	Whether the function may output a partial MAC.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_POINTER_RAM_RANGE_INVALID #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_AES_UNSUPPORTED_ERROR #ERR_LOG_CODE_AES_BUFFER_SMALL_ERROR #ERR_LOG_CODE_AES_UNAVAILABLE_ERROR #ERR_LOG_CODE_AES_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 248 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.25 user_crypto_aes_cmac_verify_start

Description

This user API function initializes a CMAC verification operation.

Prototype

```
int32_t user_crypto_aes_cmac_verify_start (
  uint32_t key_id
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	key_id	Key ID used for CMAC verification operation. Key ID range is 1 to 12.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_PARAM_INVALID #ERR_LOG_CODE_KEY_SLOT_CORRUPTED #ERR_LOG_CODE_AES_UNSUPPORTED_ERROR #ERR_LOG_CODE_AES_BUFFER_SMALL_ERROR #ERR_LOG_CODE_AES_UNAVAILABLE_ERROR #ERR_LOG_CODE_AES_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 116 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.26 user_crypto_aes_cmac_verify_update

Description

This user API function updates the ongoing CMAC verification.
Call user_crypto_aes_cmac_verify_start routine before the first update operation. The function can be called multiple times.

Prototype

```
int32_t user_crypto_aes_cmac_verify_update (
  user_crypto_inp_buf_t * buf
)
```

Parameters

Data type	Name	Description	Dir
user_crypto_inp_buf_t *	buf	Input buffer for crypto operation.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_POINTER_RAM_RANGE_INVALID #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_AES_UNSUPPORTED_ERROR #ERR_LOG_CODE_AES_BUFFER_SMALL_ERROR #ERR_LOG_CODE_AES_UNAVAILABLE_ERROR #ERR_LOG_CODE_AES_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 176 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.27 user_crypto_aes_cmac_verify_finish

Description

This user API function finalizes the ongoing CMAC verification.
The function concludes the entire CMAC verification operation and clears the cryptographic context from the reserved secure RAM.

Prototype

```
int32_t user_crypto_aes_cmac_verify_finish (
  user_crypto_cmac_t * mac
)
```

Parameters

Data type	Name	Description	Dir
user_crypto_cmac_t *	mac	Buffer for crypto operation.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_POINTER_RAM_RANGE_INVALID #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_AES_UNSUPPORTED_ERROR #ERR_LOG_CODE_AES_BUFFER_SMALL_ERROR #ERR_LOG_CODE_AES_UNAVAILABLE_ERROR #ERR_LOG_CODE_AES_ERROR #ERR_LOG_CODE_CMAC_VERIFY_FAIL

Stack Usage

The execution of this API function has a maximum stack usage of 272 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.28 user_crypto_aes_start

Description

This user API function initializes an AES operation.

Prototype

```
int32_t user_crypto_aes_start (
  user_crypto_fid_t fid
  uint32_t key_id
  user_crypto_cbc_t * cbc_ctx
)
```

Parameters

Data type	Name	Description	Dir
user_crypto_fid_t	fid	The ID of the desired operation.	-
uint32_t	key_id	The key ID used for AES operation. Key ID range is 1 to 12.	-
user_crypto_cbc_t *	cbc_ctx	Initial vector for the CBC encryption operation. For other operations (CBC decryption or ECB operation), set this to NULL.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_POINTER_RAM_RANGE_INVALID #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_PARAM_INVALID #ERR_LOG_CODE_KEY_SLOT_CORRUPTED #ERR_LOG_CODE_AES_UNSUPPORTED_ERROR #ERR_LOG_CODE_AES_BUFFER_SMALL_ERROR #ERR_LOG_CODE_AES_UNAVAILABLE_ERROR #ERR_LOG_CODE_AES_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 148 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.29 user_crypto_aes_update

Description

This user API function updates the ongoing AES operation.
Call user_crypto_aes_start routine before the first update operation. The function can be called multiple times.

Prototype

```
int32_t user_crypto_aes_update (
  user_crypto_io_buf_t * buf
)
```

Parameters

Data type	Name	Description	Dir
user_crypto_io_buf_t *	buf	I/O buffer for crypto operation.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_POINTER_RAM_RANGE_INVALID #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_PARAM_INVALID #ERR_LOG_CODE_AES_UNSUPPORTED_ERROR #ERR_LOG_CODE_AES_BUFFER_SMALL_ERROR #ERR_LOG_CODE_AES_UNAVAILABLE_ERROR #ERR_LOG_CODE_AES_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 240 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.30 user_crypto_aes_finish

Description

This user API function finalizes the ongoing AES operation.
The function concludes the entire AES operation and clears the cryptographic context from the reserved secure RAM.

Prototype

```
int32_t user_crypto_aes_finish (
  user_crypto_io_buf_t * buf
)
```

Parameters

Data type	Name	Description	Dir
user_crypto_io_buf_t *	buf	I/O buffer for crypto operation.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_POINTER_RAM_RANGE_INVALID #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_PARAM_INVALID #ERR_LOG_CODE_AES_UNSUPPORTED_ERROR #ERR_LOG_CODE_AES_BUFFER_SMALL_ERROR #ERR_LOG_CODE_AES_UNAVAILABLE_ERROR #ERR_LOG_CODE_AES_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 264 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.31 user_crypto_key_write

Description

This user API function writes a cryptographic key to the target key slot.

Prototype

```
int32_t user_crypto_key_write (
  user_key_write_t * key_write_params
)
```

Parameters

Data type	Name	Description	Dir
user_key_write_t *	key_write_params	Key write parameters.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_POINTER_RAM_RANGE_INVALID #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_PARAM_INVALID #ERR_LOG_CODE_AES_UNSUPPORTED_ERROR #ERR_LOG_CODE_AES_BUFFER_SMALL_ERROR #ERR_LOG_CODE_AES_UNAVAILABLE_ERROR #ERR_LOG_CODE_AES_ERROR #ERR_LOG_CODE_CMAC_VERIFY_FAIL #ERR_LOG_CODE_KEY_SLOT_CORRUPTED #ERR_LOG_CODE_KEY_PROTECTED #ERR_LOG_CODE_KEY_VERSION #ERR_LOG_CODE_KEY_SIZE #ERR_LOG_CODE_ACCESS_AB_MODE_ERROR #ERR_LOG_CODE_NVM_VER_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 464 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.32 user_crypto_key_erase

Description

This user API function erases a cryptographic key.

Prototype

```
int32_t user_crypto_key_erase (
  user_key_erase_t * key_erase_params
)
```

Parameters

Data type	Name	Description	Dir
user_key_erase_t *	key_erase_params	Key erase parameters.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_USER_POINTER_RAM_RANGE_INVALID #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_PARAM_INVALID #ERR_LOG_CODE_AES_UNSUPPORTED_ERROR #ERR_LOG_CODE_AES_BUFFER_SMALL_ERROR #ERR_LOG_CODE_AES_UNAVAILABLE_ERROR #ERR_LOG_CODE_AES_ERROR #ERR_LOG_CODE_CMAC_VERIFY_FAIL #ERR_LOG_CODE_KEY_SLOT_CORRUPTED #ERR_LOG_CODE_KEY_PROTECTED #ERR_LOG_CODE_KEY_VERSION #ERR_LOG_CODE_KEY_SIZE #ERR_LOG_CODE_KEY_ERASE_FAIL

Stack Usage

The execution of this API function has a maximum stack usage of 376 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.33 user_crypto_key_verify

Description

This user API function verifies an existing cryptographic key. An additional key repair operation can be enabled by setting do_repair.

Prototype

```
int32_t user_crypto_key_verify (
  uint8_t key_id
  bool do_repair
)
```

Parameters

Data type	Name	Description	Dir
uint8_t	key_id	Key ID to verify. Key ID range is 1 to 12.	-
bool	do_repair	The repair option. • false: Performs a verification operation only. • true: Performs a verification operation. Additionally, in case of a verification failure, attempts to repair the key slot using redundancy.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_AES_UNAVAILABLE_ERROR #ERR_LOG_CODE_PARAM_INVALID #ERR_LOG_CODE_ACCESS_AB_MODE_ERROR #ERR_LOG_CODE_KEY_SLOT_MISMATCH #ERR_LOG_CODE_KEY_SLOT_CORRUPTED #ERR_LOG_CODE_KEY_VERIFY_FAIL #ERR_LOG_CODE_NVM_VER_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 200 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.34 user_nvm_isr_handler

Description

The NVM read-while-write interrupt handler called by the NVM state machine.

It is needed for the background write/erase operation. The handler shall be specified in the corresponding vector table. Upon completion of NVM RWW state machine operation, NVM invokes the handler to perform the rest of the operation.

Prototype

```
void user_nvm_isr_handler (void)
```

Parameters

```
void
```

Stack Usage

The execution of this API function has a maximum stack usage of 144 bytes.

Remarks

It is an interrupt handler that can not be called directly.

4.2.35 user_secure_download_start

Description

This user API function initializes the secure container and starts the secure download process of data to the secure container.

The function decrypts the first [0:31] bytes of the input streaming data. Upon successful start of secure download operation, the next call of user_secure_download_update routine expects [32:159] byte of input streaming data.

Prototype

```
int32_t user_secure_download_start (
  uint8_t key_id
  uint8_t n_sectors
  uint8_t * data
)
```

Parameters

Data type	Name	Description	Dir
uint8_t	key_id	The ID of the key for decryption, key ID range is 0 to 12.	-
uint8_t	n_sectors	Size in sectors, of the new secure container.	-
uint8_t *	data	Address of the input buffer.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_USER_POINTER_RAM_RANGE_INVALID #ERR_LOG_CODE_SIZE_INVALID #ERR_LOG_CODE_SEGMENT_PROTECTED #ERR_LOG_CODE_ACCESS_AB_MODE_ERROR #ERR_LOG_CODE_PARAM_INVALID #ERR_LOG_CODE_AES_UNSUPPORTED_ERROR #ERR_LOG_CODE_AES_BUFFER_SMALL_ERROR #ERR_LOG_CODE_AES_UNAVAILABLE_ERROR #ERR_LOG_CODE_AES_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 304 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.36 user_secure_download_update

Description

This user API function continues the secure download process of data to the secure container.

Call the user_secure_download_start routine before the first update operation. The function can be called multiple times. Each call decrypts 128 bytes of input streaming data and writes the decrypted data (128 bytes) into the target page. The user shall feed in new data with each call.

Prototype

```
int32_t user_secure_download_update (
  uint32_t page_index
  uint8_t * data
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	page_index	The index of the page to which to write, starting from a secure container start address.	-
uint8_t *	data	Address of the input buffer.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_USER_POINTER_RAM_RANGE_INVALID #ERR_LOG_CODE_PARAM_INVALID #ERR_LOG_CODE_AES_UNSUPPORTED_ERROR #ERR_LOG_CODE_AES_BUFFER_SMALL_ERROR #ERR_LOG_CODE_AES_UNAVAILABLE_ERROR #ERR_LOG_CODE_AES_ERROR #ERR_LOG_CODE_NVM_VER_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 432 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.37 user_secure_download_finish

Description

This user API function finalizes the secure download process of data to the secure container.
The function concludes the entire secure download process and clears the cryptographic context.

Prototype

```
int32_t user_secure_download_finish (void)
```

Parameters

```
void
```

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_AES_UNSUPPORTED_ERROR #ERR_LOG_CODE_AES_BUFFER_SMALL_ERROR #ERR_LOG_CODE_AES_UNAVAILABLE_ERROR #ERR_LOG_CODE_AES_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 448 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.38 user_cache_operation

Description

This user API function provides an alternative to writing to cache registers in addition to direct register access.

Prototype

```
int32_t user_cache_operation (
  user_cache_op_t op
  uint32_t address
)
```

Parameters

Data type	Name	Description	Dir
user_cache_op_t	op	The code for the cache operation to perform.	-
uint32_t	address	The memory address, namely the FLASH1 access.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_PARAM_INVALID

Stack Usage

The execution of this API function has a maximum stack usage of 12 bytes.

4.2.39 user_secure_dualboot

Description

This user API function configures and enables the secondary UBSL image.

Prototype

```
int32_t user_secure_dualboot (
  uint32_t image_offset
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	image_offset	New image address offset (the offset of startup page address), starting from the UBSL segment start address.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_MEM_ADDR_RANGE_INVALID #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_ACCESS_AB_MODE_ERROR #ERR_LOG_CODE_NVM_VER_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 144 bytes.

Remarks

This functional is callable only from UBSL segment.
In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.

4.2.40 user_ubsl_size_restore

Description

This user API function is used to restore the UBSL size in case of a Stop mode exit.

Prototype

```
void user_ubsl_size_restore (void)
```

Parameters

```
void
```

Stack Usage

The execution of this API function does not need stack memory.

Remarks

If user has an UBSL size configuration different than the default setting, in case of stop mode exit, this function must be called after exit from the Stop mode.

4.2.41 user_nvm_perm_protect_set

Description

This user API function sets permanent protection on all NVM segments.

Prototype

```
int32_t user_nvm_perm_protect_set (
  uint32_t passphrase
)
```

Parameters

Data type	Name	Description	Dir
uint32_t	passphrase	An encoding of the target segment and the erase flag.	-

Return values

Data type	Description
int32_t	Function execution status. #ERR_LOG_SUCCESS #ERR_LOG_CODE_SEMAPHORE_RESERVED #ERR_LOG_CODE_USER_PROTECT_WRONG_PASSWORD #ERR_LOG_CODE_SEGMENT_PROTECTED #ERR_LOG_CODE_NVM_APPLY_PROTECTION_FAIL #ERR_LOG_CODE_ACCESS_AB_MODE_ERROR #ERR_LOG_CODE_NVM_VER_ERROR #ERR_LOG_CODE_ECC2READ_ERROR

Stack Usage

The execution of this API function has a maximum stack usage of 168 bytes.

Remarks

In an interrupt or multithreaded environment, this function cannot be called in a re-entrant context.
It is recommended to disable the interrupt before calling the function.

4.3 Data types and structure reference

This chapter contains the reference of data types and structures of all modules.

4.3.1 User API data types

These routines are exported by the BootROM to the customer user mode software.

Table 42 User API data types structure overview

Name	Description
user_100tp_read_t	100TP read parameters.
user_100tp_write_t	100TP write parameters.
user_crypto_inp_buf_t	Input buffer for crypto operation.
user_crypto_out_buf_t	Output buffer for crypto operation.
user_crypto_io_buf_t	I/O buffer for crypto operation.
user_crypto_cmac_t	Buffer for crypto operation.
user_crypto_cbc_t	Initial vector for the CBC encryption operation.
user_key_write_t	Key write configuration.
user_key_write_params_t	Key write parameters.
user_key_erase_t	Key erase configuration.
user_key_erase_params_t	Key erase parameters.
user_nvm_page_write_t	NVM write parameters.
user_key_t	User key data structure.

4.3.1.1 user_100tp_read_t

Prototype

```
typedef struct user_100tp_read_t
{
  uint32_t   offset;
  uint8_t    *data;
  uint16_t   nbyte;
} user_100tp_read_t;
```

Parameters

Name	Description
offset	Byte offset inside the selected page address, where to start reading. Maximum is 127 bytes.
data	Data pointer where to write data into. Pointer plus valid count must be within valid RAM range or an error code is returned
nbyte	Amount of data bytes to read. If nbyte is zero, there is no read operation done and an error code is returned. Maximum is 128 bytes.

4.3.1.2 user_100tp_write_t

Prototype

```
typedef struct user_100tp_write_t
{
  uint32_t   offset;
  uint8_t    *data;
  uint8_t    nbyte;
  uint8_t    counter;
} user_100tp_write_t;
```

Parameters

Name	Description
offset	Byte offset inside the selected page address, where to start writing. Maximum is 123 bytes.
data	Data pointer where to read the data to write. Pointer plus valid count must be within valid RAM range or an error code is returned
nbyte	Amount of data bytes to write. If nbyte is zero, there is no write operation done and an error code is returned. Maximum is 124 bytes.
counter	Counter value to update internal 100TP counter (only updates if value is greater than current, otherwise is ignored)

4.3.1.3 user_crypto_inp_buf_t

Prototype

```
typedef struct user_crypto_inp_buf_t
{
  uint8_t   *buffer;
  uint32_t   length;
} user_crypto_inp_buf_t;
```

Parameters

Name	Description
buffer	Crypto algorithm input buffer address
length	Crypto algorithm input buffer length

4.3.1.4 user_crypto_out_buf_t

Prototype

```
typedef struct user_crypto_out_buf_t
{
  uint8_t   *buffer;
  uint32_t *length;
} user_crypto_out_buf_t;
```

Parameters

Name	Description
buffer	Crypto algorithm output buffer address
length	Crypto algorithm output buffer length

4.3.1.5 user_crypto_io_buf_t

Prototype

```
typedef struct user_crypto_io_buf_t
{
  user_crypto_inp_buf_t inp;
  user_crypto_out_buf_t out;
} user_crypto_io_buf_t;
```

Parameters

Name	Description
inp	Crypto algorithm input buffer
out	Crypto algorithm output buffer

4.3.1.6 user_crypto_cmac_t

Prototype

```
typedef struct user_crypto_cmac_t
{
  user_crypto_inp_buf_t inp;
  user_crypto_inp_buf_t mac;
} user_crypto_cmac_t;
```

Parameters

Name	Description
inp	CMAC generate input buffer
mac	CMAC generate output buffer

4.3.1.7 user_crypto_cbc_t

Prototype

```
typedef struct user_crypto_cbc_t
{
  void      *iv;
  uint32_t   iv_length;
} user_crypto_cbc_t;
```

Parameters

Name	Description
iv	CBC input vector
iv_length	CBC input vector length

4.3.1.8 user_key_write_t

Prototype

```
typedef struct user_key_write_t
{
  user_key_write_params_t params;
  uint8_t                 signature[USER_CMAC_SIGNATURE_SIZE];
} user_key_write_t;
```

Parameters

Name	Description
params	Input parameters (signature checked)
signature	New key CMAC signature

4.3.1.9 user_key_write_params_t

Prototype

```
typedef struct user_key_write_params_t
{
  uint8_t    encrypted_key_buf[USER_KEY_PARAM_SIZE];
  uint16_t   target_key_id;
  uint16_t   encrypt_key_id;
} user_key_write_params_t;
```

Parameters

Name	Description
encrypted_key_buf	Encrypted buffer with new key parameters
target_key_id	Key slot ID for parameter decryption
encrypt_key_id	Key slot ID used for the new key parameters

4.3.1.10 user_key_erase_t

Prototype

```
typedef struct user_key_erase_t
{
  user_key_erase_params_t params;
  uint8_t                 signature[USER_CMAC_SIGNATURE_SIZE];
} user_key_erase_t;
```

Parameters

Name	Description
params	Input parameters (signature checked)
signature	CMAC signature

4.3.1.11 user_key_erase_params_t

Prototype

```
typedef struct user_key_erase_params_t
{
  uint16_t   target_key_id;
  uint16_t   version;
} user_key_erase_params_t;
```

Parameters

Name	Description
target_key_id	Key slot ID for parameter decryption
version	New key version number

4.3.1.12 user_nvm_page_write_t

Prototype

```
typedef struct user_nvm_page_write_t
{
  uint8_t    *data;
  uint32_t   nbyte;
  uint32_t   options;
} user_nvm_page_write_t;
```

Parameters

Name	Description
data	Pointer to the data where to read the programming data. Pointer must be within valid RAM range or an error code is returned.
nbyte	Amount of bytes to program. Range from 1-128 bytes.
options	NVM programming options (e.g. NVM_OPTIONS_CORR_ACT or NVM_OPTIONS_NO_FAILPAGE_ERASE, see for a full list)

4.3.1.13 user_key_t

Prototype

```
typedef struct user_key_t
{
  uint8_t    key[USER_KEY_SIZE_MAX];
  uint16_t   version;
  uint8_t    length;
  uint8_t    protection;
} user_key_t;
```

Parameters

Name	Description
key	Key value
version	New key version number
length	Key size in bytes (16 or 32)
protection	Key protection

4.3.2 User API enumerations

This chapter contains the enumerator reference.

Table 43 Enumerator overview

Name	Description
user_crypto_fid_t	
user_cache_op_t	
user_nvm_segment_t	
erase_scope_e	

4.3.2.1 user_crypto_fid_t

Prototype

```
typedef enum user_crypto_fid_t
{
  CRYPTO_ECB_ENCRYPT    = 0,
  CRYPTO_ECB_DECRYPT    = 1,
  CRYPTO_CBC_ENCRYPT    = 2,
  CRYPTO_CBC_DECRYPT    = 3
} user_crypto_fid_t;
```

Parameters

Name	Value	Description
CRYPTO_ECB_ENCRYPT	00_H	Encrypt with ECB
CRYPTO_ECB_DECRYPT	01_H	Decrypt with ECB
CRYPTO_CBC_ENCRYPT	02_H	Encrypt with CBC
CRYPTO_CBC_DECRYPT	03_H	Decrypt with CBC

4.3.2.2 user_cache_op_t

Prototype

```
typedef enum user_cache_op_t
{
  CACHE_OP_AC  = 0,
  CACHE_OP_SC  = 1,
  CACHE_OP_BC  = 2,
  CACHE_OP_BT  = 3,
  CACHE_OP_BL  = 4,
  CACHE_OP_BU  = 5,
  CACHE_OP_EN  = 6,
  CACHE_OP_DIS = 7
} user_cache_op_t;
```

Parameters

Name		Description
CACHE_OP_AC	00_H	Cache all clean operation
CACHE_OP_SC	01_H	Cache set clean operation
CACHE_OP_BC	02_H	Cache block clean operation
CACHE_OP_BT	03_H	Cache block touch operation
CACHE_OP_BL	04_H	Cache block lock operation
CACHE_OP_BU	05_H	Cache block unlock operation
CACHE_OP_EN	06_H	Cache enable operation
CACHE_OP_DIS	07_H	Cache disable operation

4.3.2.3 user_nvm_segment_t

Prototype

```
typedef enum user_nvm_segment_t
{
  NVM_PASSWORD_SEGMENT_BOOT = 0,
  NVM_PASSWORD_SEGMENT_CODE = 1,
  NVM_PASSWORD_SEGMENT_DATA = 2,
  NVM_PASSWORD_SEGMENT_TOTAL = 3
} user_nvm_segment_t;
```

Parameters

Name	Value	Description
NVM_PASSWORD_SEGMENT_BOOT	00_H	NVM password for customer segment, used for customer bootloader (FLASH0).
NVM_PASSWORD_SEGMENT_CODE	01_H	NVM password for customer code segment, which is not used by the customer bootloader (FLASH1).
NVM_PASSWORD_SEGMENT_DATA	02_H	NVM password for customer data mapped segment (FLASH0).
NVM_PASSWORD_SEGMENT_TOTAL	03_H	Can be ignored and should not be used

4.3.2.4 erase_scope_e

Description

Erase scope.

Prototype

```
typedef enum {
  NVM_ERASE_PAGE = 0x00,
  NVM_ERASE_SECTOR = 0x01,
  NVM_ERASE_COMPLETE = 0x02,
} erase_scope_e;
```

Parameters

Name	Value	Description
NVM_ERASE_PAGE	00_H	Page erase
NVM_ERASE_SECTOR	01_H	Sector erase
NVM_ERASE_COMPLETE	02_H	Mass erase

4.3.3 Constant reference

This chapter contains the constant reference.

Table 44 Constant overview

Name	Value	Description
NVM_FLASH_0	00_H	Target NVM FLASH0.
NVM_FLASH_1	01_H	Target NVM FLASH1.
NVM_SEG_PROT_UBSL_NO_ERASE	$BB005555_H$	UBSL segment passphrase without erase flag.
NVM_SEG_PROT_UBSL_DO_ERASE	$BBFF5555_H$	UBSL segment passphrase with erase flag.
NVM_SEG_PROT_CODE_NO_ERASE	$CC005555_H$	UCODE segment passphrase without erase flag.
NVM_SEG_PROT_CODE_DO_ERASE	$CCFF5555_H$	UCODE segment passphrase with erase flag.
NVM_SEG_PROT_DATA_NO_ERASE	$DD005555_H$	UDATA segment passphrase without erase flag.
NVM_SEG_PROT_DATA_DO_ERASE	$DDFF5555_H$	UDATA segment passphrase with erase flag.
NVM_OPTIONS_NONE	00_H	NVM operation options. No options provided, the default setting: corrective action disabled, RWW enabled, failpage erase enabled.
NVM_OPTIONS_RWW_DISABLE	01_H	Disable RWW.
NVM_OPTIONS_CORR_ACT	02_H	Disturb handling and retry enabled (data mapped mode only).
NVM_OPTIONS_NO_FAILPAGE_ERASE	04_H	Erasing of programmed data on fail enabled (data linear mode only).
NVM_OP_STS_FLASH_READY	0_H	NVM flash NOT busy status operation return code.
NVM_OP_STS_FLASH_0_BUSY	01_H	NVM FLASH0 busy status operation return code.
NVM_OP_STS_FLASH_1_BUSY	02_H	NVM FLASH1 busy status operation return code.
NVM_RET_PROTECTED	01_H	NVM segment is protected.
NVM_RET_NOT_PROTECTED	00_H	NVM segment is not protected.

Table 45 NVM_OP_RESULT register error log codes

Error Log Name	Error Log Code
ERR_LOG_SUCCESS	00_H
ERR_LOG_CODE_ACCESS_AB_MODE_ERROR	$FFFFFFD9_H$
ERR_LOG_CODE_NVM_ECC2_DATA_ERROR	$FFFFFFD8_H$
ERR_LOG_CODE_NVM_VER_ERROR	$FFFFFFD7_H$
ERR_LOG_CODE_MAPRAM_INIT_FAIL	$FFFFFFD6_H$
ERR_LOG_CODE_VERIFY_AND_MAPRAM_INIT_FAIL	$FFFFFFD5_H$

5 Glossary

100TP	100-time-programming
AES	Advanced Encryption Standard
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BSL	Bootstrap Loader
BootROM	Boot code in ROM
CAN	Controller Area Network
CBC	Cipher Block Chaining
CFS0	Configuration Sector 0
Ciphertext	Encrypted text (confer Plaintext)
CS	Configuration Sector
CMAC	Cipher-based Message Authentication Code
CPU	Central Processing Unit
DSRAM	Data Static Random Access Memory
EEPROM	Electrically Erasable Programmable Read-Only Memory
ECB	Electronic Code Book
ECC	Error Correcting Code
EOT	End of Transmission
FAR	Fault, Asset, and Risk
FLASH0	First flash bank (NVM0)
FLASH1	Second flash bank (NVM1)
FS_WDT	Fail-safe Watchdog
FSM	Finite State Machine
ID	Identifier
ISR	Interrupt Service Routine
LSB	Least Significant Bit
MAC	Message Authentication Code
MBIST	Memory Built-in Self-test
MCU	Microcontroller Unit
MSB	Most Significant Bit
NAC	No-activity Counter
NAD	Node Address
NVM	Non-volatile Memory
Plaintext	Unencrypted or decrypted text (confer Ciphertext)
PSRAM	Program Static Random Access Memory

RAM	Random Access Memory
ROM	Read Only Memory
RWW	Read-while-write
SA	Service Algorithm
SDK	Software Development Kit
SRAM	Static Random Access Memory
SSC	Secured Software Container
SWD	Serial Wire Debug
UBSL	User Bootstrap Loader
UCODE	User Code
UDATA	User Data

6 Revision history

Revision	Date	Changes
1.0	2023-05-16	Initial version

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Edition 2023-05-16

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
IFX-Z8F80177275