SmartFusion2 MSS System Services Driver User's Guide

Version 2.9

Microsemi

Table of Contents

Introduction

The SmartFusion2™ microcontroller subsystem (MSS) includes a communication block (COMM_BLK) allowing it to communicate with the SmartFusion2 System Controller. The SmartFusion2 System Controller performs a variety of system wide services. This software driver provides a set of functions to access these System Services. The driver can be adapted for use as part of an operating system, but the implementation of the adaptation layer between the driver and the operating system's driver model is outside the scope of the driver.

Features

The MSS System Services driver provides support for the following features:

- Reading the SmartFusion2 device serial number
- Reading the design version
- Random number generation
- Data encryption and decryption
- Requesting the SmartFusion2 device to enter Flash*Freeze mode.
- Zeroization of sensitive data
- Programming the SmartFusion2 device
- Performing data integrity digest checks

The MSS System Services driver is provided as C source code.

Supported Hardware IP

The SmartFusion2 MSS System Services bare metal driver can be used with the SmartFusion2 MSS version 0.0.500 or higher.

Files Provided

The files provided as part of the MSS System Services driver fall into three main categories: documentation, driver source code and example projects. The driver is distributed via the Microsemi SoC Products Group's Firmware Catalog, which provides access to the documentation for the driver, generates the driver's source files into an application project and generates example projects that illustrate how to use the driver. The Firmware Catalog is available from: www.microsemi.com/soc/products/software/firmwarecat/default.aspx.

Documentation

The Firmware Catalog provides access to these documents for the driver:

- User's guide (this document)
- Release notes

Driver Source Code

The Firmware Catalog generates the driver's source code into a *drivers\mss_sys_services* subdirectory of the selected software project directory. The files making up the driver are detailed below.

mss_sys_services.h

This header file contains the public application programming interface (API) of the MSS System Services software driver. This file should be included in any C source file that uses the MSS System Services software driver.

mss_sys_services.c

This C source file contains the implementation of the MSS System Services software driver.

mss_comblk.h

This header file contains the public application programming interface (API) of the MSS COMM_BLK software driver. This file should not be included in any application C source files. It is only used internally by the MSS System Services driver.

mss_comblk_page_handler.h

This header file contains the prototype of a callback function used by the ISP service. This file should not be included in any application C source files. It is only used internally by the MSS System Services driver.

mss_comblk.c

This C source file contains the implementation of the MSS COMM_BLK software driver.

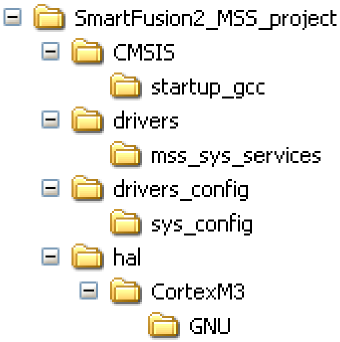
Example Code

The Firmware Catalog provides access to example projects illustrating the use of the driver. Each example project is self contained and is targeted at a specific processor and software toolchain combination. The example projects are targeted at the FPGA designs in the hardware development tutorials supplied with the SoC Products Group's development boards. The tutorial designs can be found on the Microsemi SoC Development Kit web page.

Driver Deployment

This driver is deployed from the Firmware Catalog into a software project by generating the driver's source files into the project directory. The driver uses the SmartFusion2 Cortex Microcontroller Software Interface Standard Hardware Abstraction Layer (CMSIS HAL) to access MSS hardware registers. You must ensure that the SmartFusion2 CMSIS HAL is included in the project settings of the software toolchain used to build your project and that it is generated into your project. The most up-to-date SmartFusion2 CMSIS HAL files can be obtained using the Firmware Catalog.

The following example shows the intended directory structure for a SoftConsole ARM® Cortex™-M3 project targeted at the SmartFusion2 MSS. This project uses the MSS System Services driver. This driver relies on SmartFusion2 CMSIS HAL for accessing the hardware. The contents of the *drivers* directory result from generating the source files for the driver into the project. The contents of the *CMSIS* and *hal* directories result from generating the source files for the SmartFusion2 CMSIS HAL into the project. The contents of the *drivers_config* directory are generated by the Libero project and must be copied into the into the software project.



```
SmartFusion2_MSS_project
    CMSIS
        startup_gcc
    drivers
        mss_sys_services
    drivers_config
        sys_config
    hal
        CortexM3
            GNU
```

Figure 1 · SmartFusion2 MSS Project Example

Driver Configuration

The MSS System Services driver does not require any configuration. It relies on the SmartFusion2 communication block (MSS COMM_BLK) to communicate with the System Controller. The MSS COMM_BLK is always enabled.

The base address, register addresses and interrupt number assignment for the MSS COMM_BLK are defined as constants in the SmartFusion2 CMSIS HAL. You must ensure that the latest SmartFusion2 CMSIS HAL is included in the project settings of the software tool chain used to build your project and that it is generated into your project.

Application Programming Interface

This section describes the driver’s API. The functions and related data structures described in this section are used by the application programmer to access the SmartFusion2 System Services.

Theory of Operation

The System Services driver provides access to the SmartFusion2 System Controller services. These system services are loosely grouped into the following categories:

- Reading system information
- Cryptography
- Non-deterministic random bit generator
- Flash*Freeze
- Zeroization
- Asynchronous event handling
- Programming
- Checking data integrity digests

Note: System services should not be executed from interrupt service routines (ISR) if this can be avoided. If an application needs to execute a system service from an ISR, then the MSS COMM_BLK interrupt to the Cortex-M3 NVIC must be assigned a higher priority than the interrupt source of the ISR calling the system service function.

Initialization

The System Services driver is initialized through a call to the *MSS_SYS_init()* function. The *MSS_SYS_init()* function must be called before any other System Services driver functions are called. The *MSS_SYS_init()* function can also register a callback function with the System Services driver to handle asynchronous messages from the System Controller. The System Controller sends asynchronous messages to the MSS COMM_BLK when certain events are detected during the execution of the following system services:

- Flash*Freeze
- Power-on-reset (POR) digest check
- Tamper detect events

Note: The implementation and registration of an asynchronous event handler function is optional and is the responsibility of the application program.

Reading System Information

The System Services driver can be used to read information about the SmartFusion2 device and the design programmed into it using the following functions:

- *MSS_SYS_get_serial_number()*
- *MSS_SYS_get_user_code()*
- *MSS_SYS_get_design_version()*
- *MSS_SYS_get_device_certificate()*
- *MSS_SYS_get_secondary_device_certificate()*

Cryptography Services

The System Services driver provides cryptographic services using the following functions:

- *MSS_SYS_128bit_aes()*

- *MSS_SYS_256bit_aes()*
- *MSS_SYS_sha256()*
- *MSS_SYS_hmac()*
- *MSS_SYS_key_tree()*
- *MSS_SYS_challenge_response()*

Non-Deterministic Random Bit Generator

The System Services driver provides random number generation services using the following functions:

- *MSS_SYS_nrbg_instantiate()*
- *MSS_SYS_nrbg_self_test()*
- *MSS_SYS_nrbg_generate()*
- *MSS_SYS_nrbg_reseed()*
- *MSS_SYS_nrbg_uninstantiate()*
- *MSS_SYS_nrbg_reset()*

Flash*Freeze

The System Services driver can be used to request the system to enter Flash*Freeze mode using the following function:

- *MSS_SYS_flash_freeze()*

The System Controller sends an asynchronous message to the MSS COMM_BLK when the device is either about to enter or has exited Flash*Freeze mode. The *MSS_SYS_init()* function can register a callback function with the System Services driver to handle these asynchronous messages.

Zeroization

The System Services driver can be used to destroy sensitive information using the following function:

- *MSS_SYS_zeroize_device()*

The zeroization system service erases all user configuration data, user keys, user security settings, NVM, SRAM, FPGA flip-flops, System Controller memory, and crypto-engine registers. The zeroization system service is enabled and configured in the Libero hardware flow.

Note: The zeroization system service can render the SmartFusion2 device permanently and irrevocably disabled depending on the configuration selected in the Libero hardware flow.

Asynchronous Messages

The System Controller sends asynchronous messages to the MSS COMM_BLK when certain events are detected during the execution of the following system services:

- Flash*Freeze
- Power-on-reset (POR) digest check
- Tamper detect events

The *MSS_SYS_init()* function can register a callback function with the System Services driver to handle these asynchronous messages allowing the user application code to take remedial or defensive action. If the application code does not provide an asynchronous event handler function then the driver simply reads and discards these asynchronous messages.

Flash*Freeze Entry and Exit

The System Controller sends a Flash*Freeze entry or exit message to the MSS COMM_BLK when the SmartFusion2 device is either about to enter or has exited Flash*Freeze mode. The driver passes the entry/exit message opcode as a parameter to the event handler callback function.

Power-on Reset (POR) Digest Error

The POR digest check service is enabled in the Libero hardware flow and if enabled is automatically performed as part of the device’s power up sequence. The System Controller sends a POR digest check error message to the MSS COMM_BLK when the result of the POR digest check is a mismatch between the original stored digest and the current digest. The driver passes the command byte and the error flags byte from the error message as parameters to the event handler callback function.

Tamper detect events

The System Controller sends a tamper message to the MSS COMM_BLK when a tamper event is detected. This tamper message is a single byte containing only a command opcode. The driver passes the tamper message opcode as a parameter to the event handler callback function.

Programming Service

The In-System Programming (ISP) system service can be used for field upgrades of the hardware design programmed in the FPGA fabric. The application program running on the Cortex-M3 controls the ISP operations; it must retrieve the programming file from memory or from a communication port and feed it to the System Controller through the ISP system service. The System Controller performs the actual programming of the FPGA (fabric and eNVM) using the programming data it receives from the Cortex-M3 as part of the ISP system service.

Programming files are large and cannot always be entirely retrieved by the application before starting the ISP operation. The ISP system service is designed to work around this issue by handling programming one page at a time. The application initiates the ISP operation through a call to *MSS_SYS_start_isp()*, passing two function pointers as parameters. These two function pointers point to a page read handler function and an ISP completion handler function that must be implemented as part of the application. The system services driver will call the application’s page read handler function every time it is ready to program the FPGA with a new page of programming data. The page read handler function is responsible for splitting the programming file into suitably sized pages. The page size is not fixed and can be chosen to suit the application. The system services driver will call the ISP completion handler function once the last page has been programmed.

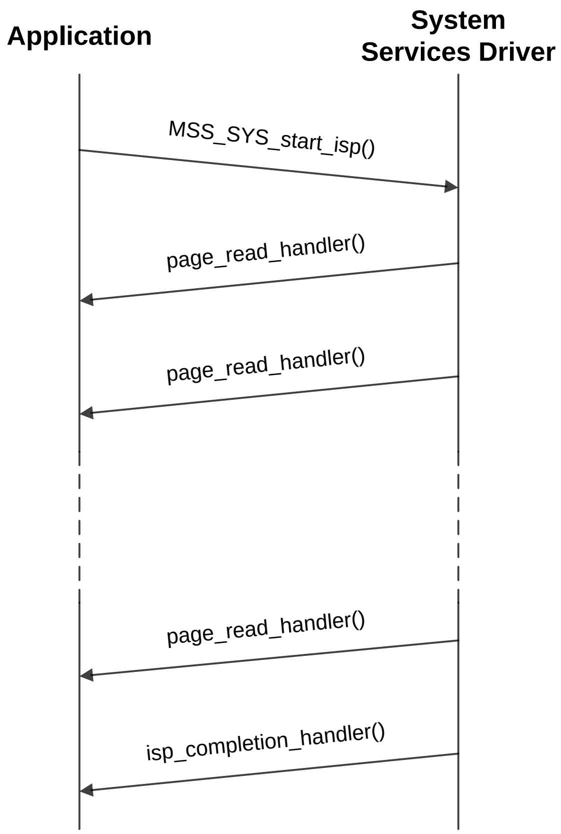


Figure 2 · ISP System Service Flow

The application must keep track of the pages of programming data is has passed to the ISP system service. It must provide the location of the next programming page by writing the address of the page into the location pointed to by the *pp_next_page* pointer passed as parameter to the page read handler function. The page read handler must return the size of the page or zero if the last page has already been given to the ISP system service.

In-Application programming (IAP)

The In-Application Programming (IAP) system service provides another field upgrade method of the hardware design programmed in FPGA fabric. Using this method, the user application first writes the FPGA fabric design into the SPI Flash connected to MSS SPI-0 before invoking the IAP system service. The FPGA fabric is then programmed under control of the SmartFusion2 system controller without intervention of the Cortex-M3 processor. The application must configure the SPI peripheral and provide exclusive access to the IAP service before invoking the IAP system service.

When the IAP service is invoked, the system controller will send the SPI command 0Bh i.e. single read operation command to the SPI flash to retrieve the FPGA fabric design. The system controller will perform one of the following operations based on the mode selected when invoking the service:

a) Authenticate the fabric design.

b) Program the fabric design into FPGA.

c) Verify that the programming was successful.

The system controller will take 2 to 3 minutes to complete each operation. The application must provide the starting location address of the new FPGA fabric design present in SPI Flash when invoking the IAP service.

The system controller will return a response indicating the status of the operation on completing the execution of the IAP service except in the case of successful programming. The system controller will reset the device and start execution of the new FPGA fabric design after successfully programming the device. The system controller will not return any response information when a successful programming operation completes.

Digest Check Service

The System Service driver can be used to recalculate and compare digests of selected components using the following function.

- *MSS_SYS_check_digest()*

Note: This function is not used for the power-on-reset (POR) digest check service. An asynchronous event handler registered with the *MSS_SYS_init()* function is used to support the POR digest check service.

Elliptic Curve Services

The System Service driver can be used to perform elliptic curve cryptography (ECC) mathematical operations over the field defined by the NIST P-384 curve. The following functions are provided:

- *MSS_SYS_ecc_point_multiplication()*
- *MSS_SYS_ecc_point_addition()*
- *MSS_SYS_ecc_get_base_point()*

One example use of these functions is the elliptic curve Diffie-Hellman (ECDH) key establishment protocol. In this protocol, a public key is computed by performing a point multiplication of a 384-bit private key with the base point for the NIST P-384 curve. Both parties involved in the key establishment compute their public keys and send them to each other. A shared secret is established by point multiplication of each party's private key with the remote party's public key. The elliptic curve mathematical properties ensure that the private key point multiplication with the remote party's public key results in the same point on the curve for both parties. The coordinates of this point on the curve is used as the shared secret for further cryptographic operations.

Note: There is no checking done to see if the given input point or points are valid points on the elliptic curve. Supplying illegal X-Y coordinates for a point will result in a garbage output. However, if a valid point or points are given, then the resulting output point is guaranteed to be valid.

PUF Services

The SRAM-PUF system services provide a Physically Unclonable Function (PUF) that can be used for key generation and storage as well as device authentication.

The large SmartFusion2 devices starting from the M2S060 include an SRAM-PUF hardware block as part of the System Controller. This hardware block makes use of a 16Kbit SRAM block to determine an intrinsic secret unique to each device. This intrinsic secret is in turn used to regenerate keys enrolled with the SRAM-PUF hardware block.

The SRAM-PUF is also used in design security. Although the system services are not used for design security, it is possible to retrieve the design security ECC public key(s) for device authentication purpose.

The SRAM-PUF start-up value is also used to transparently generate a random number generator seed, improving the security of the non-deterministic random bit generator in those devices having the SRAM-PUF.

Note: PUF service should be executed from DDR/SRAM/eNVM-0 except for the M2S060 device.

On the M2S060 device, PUF service should be executed from DDR/SRAM.

Activation Code

An activation code is required by the SRAM-PUF to regenerate the intrinsic secret from the SRAM-PUF start-up value. The start-up value of the PUF's SRAM block is slightly different from one power-up to the next. Some processing is performed on the PUF's SRAM start-up value to eliminate randomness and retrieve the exact same intrinsic secret on each power-up cycle. This processing is performed using the activation code, which can be thought of as parity bits that are used to reconstruct the same PUF intrinsic secret each time, in spite of the fact that some SRAM bits are flipped compared to the original snapshot used when the activation code was first enrolled.

The devices are shipped with one activation code that was enrolled during the manufacturing process. The PUF secret computed from this enrollment is used to protect a 376-bit ECC private key that may be used for design security purposes as described elsewhere. The user may optionally enroll a second activation code in special "S" -suffix (i.e., security enabled) devices. The activation code is usually generated only once, typically when the system containing the SmartFusion2 device is being commissioned, using a JTAG or SPI programming command. Alternatively, the activation code is created using the following system service function:

- *MSS_SYS_puf_create_activation_code()*

This might be used if the first user activation code were intentionally deleted and a new one was desired. The activation code is stored the System Controller's private eNVM after being created. Its value is never exported in clear text from the System Controller. Because of the inherent noise in each SRAM start-up, there is a negligible probability two activation codes or the associated PUF intrinsic secret are ever the same, even if the same device is repeatedly re-enrolled.

The activation code can later be destroyed using the following function:

- *MSS_SYS_puf_delete_activation_code()*

This function would typically only be used when the system containing SmartFusion2 is being decommissioned or repurposed.

Enrolling Keys

The SRAM-PUF can be used to store cryptographic keys. The keys are stored in such a way that the key's actual value never appears in the system unless it is retrieved by the user. A so-called "Key Code" is stored in the System Controller's private eNVM instead of the key's value. The key code is generated when a key is enrolled. The key code value is created from the enrolled key's value and the intrinsic secret value. The key's value can then later be regenerated from the key code value and intrinsic secret value upon user request.

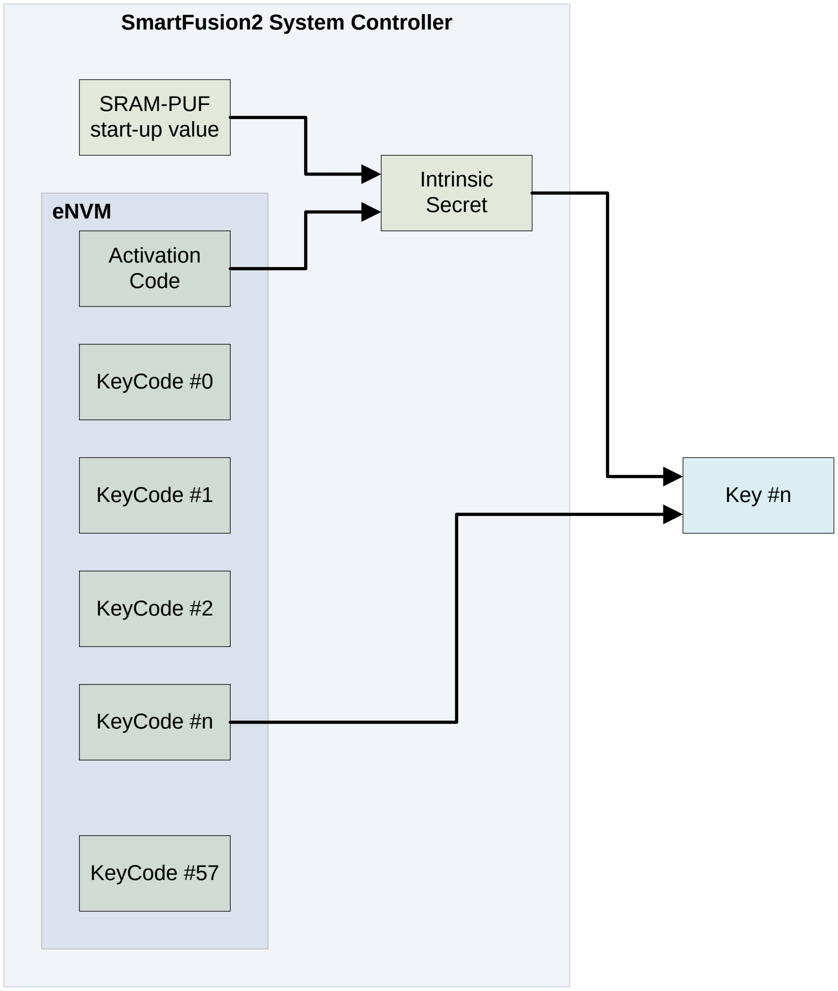


Figure 3 · PUF enrolled keys re-generation

Note: Key #0 and key #1 are used only for design security and are not accessible to the user. Key Code #2 may be used to store a user data security key, but has an important optional use in JTAG- or SPI-based key validation, as will be described below.

The enrolled keys can be “intrinsic” keys or “extrinsic” keys. The value of an intrinsic key is a random number generated from the SRAM-PUF start-up value. Intrinsic keys are useful where a security protocol executing on SmartFusion2 needs to generate a key’s value and store it for later use. For example, the user could request a 384-bit long intrinsic key to be enrolled and use it as private key in an elliptic curve Diffie-Hellman key exchange.

The value of an extrinsic key has a value specified by the user. For example, the user could request a symmetric key obtained from a key exchange protocol to be enrolled for later use.

The following functions are used for key enrollment:

- *MSS_SYS_puf_get_number_of_keys()*
- *MSS_SYS_puf_enroll_key()*
- *MSS_SYS_puf_fetch_key()*
- *MSS_SYS_puf_delete_key()*

Keys are identified by a number and must be enrolled sequentially. Key codes #0 and #1 are reserved as a 256-bit symmetric extrinsic key and a 384-bit private (asymmetric) intrinsic key, both used for design security only, and are enrolled by JTAG (or SPI) programming commands. These commands also enroll the user activation code and key code #2, a 256-bit value that has a special use in key validation, and which is enrolled at the same time as key code #1, as will be described below.

The first step in enrolling a new key is to determine how many keys are already enrolled. This is achieved by a call to function *MSS_SYS_puf_get_number_of_keys()* which returns the number of enrolled keys. Keys are numbered from 0 up to 57 (the maximum number of keys assuming all user keys are less than or equal to 256 bits in length). For example, assuming only the first three key codes mentioned above have been enrolled using the JTAG programming commands, the value returned would be '3'. The number of keys returned by *MSS_SYS_puf_get_number_of_keys()* can then be used as the newly enrolled key identification number since key numbering is zero based. Thus, the first data security key enrolled by the user would generate key code #3.

A key is enrolled through a call to function *MSS_SYS_puf_enroll_key()*. This function takes the following parameters:

- The key identification number that will be used to later retrieve the key's value. This is the value returned by a call to *MSS_SYS_puf_get_number_of_keys()*.
- The key size.
- A pointer to the buffer containing the value of the key to enroll. The value of this pointer must be NULL if enrolling an intrinsic key where the SRAM-PUF will generate the actual key value.
- A pointer to the buffer where the enrolled key will be regenerated when later fetched by the user.

The value of the enrolled keys can be regenerated through a call to function *MSS_SYS_puf_fetch_key()*. The identification number of the key to fetch is passed as parameter to this function. The requested key's value will be regenerated and copied to the buffer specified during key enrollment. The key's value is then available for use until it is no further required and wiped, by the user's application, from the memory buffer it was fetched into.

Note: It is not possible to fetch a key if the key codes have been exported and not re-imported.

A key can be removed from the system through a call to function *MSS_SYS_puf_delete_key()*. This function only requires the key identification number to specify which key should be removed from the SRAM-PUF.

Note: If a new key is enrolled in a key-slot where a key was previously enrolled and deleted and which is not the highest key number enrolled, it must be the same size as the key it replaced or an error will be reported.

Exporting and Importing Key Codes

The activation code and key codes used to regenerate the enrolled keys can be exported out of the SRAM-PUF. The exported activation code and key codes are encrypted using a one-time pad. The one-time pad is stored in the System Controller's private eNVM and is never exported. This means that the exported activation and key codes can only be decrypted by the unique device that exported them. This might be done where the activation code and key codes are stored off-chip for security reasons.

The activation and key code values stored in eNVM are replaced with hash values of themselves as part of the export procedure. This means that enrolled keys cannot be regenerated anymore after the activation and key codes have been exported. The enrolled keys will only be regenerated successfully if the exported activation and key codes for that unique SmartFusion2 device are imported back.

The activation and all the key codes are exported in one single operation using the following function:

- *MSS_SYS_puf_export_keycodes()*

The *MSS_SYS_puf_export_keycodes()* function can only be called once since the activation and key codes are not present anymore in the device after this function completes.

The activation and all the key codes are imported back in one single operation using the following function:

- *MSS_SYS_puf_import_keycodes()*

It is only possible to import activation and key codes exported from the same unique device. The imported activation and key codes are decrypted using the one-time pad stored in eNVM used during the export procedure. The decrypted activation and key codes are then checked again the hash values stored in place of the activation and key codes. The activation and key codes will only be successfully imported back if the hash values for the imported activation and key codes match the hash values stored in eNVM during the export procedure. Imported activation and key codes are never again restored to non-volatile memory; rather, they are imported to volatile scratch-pad memory, used to regenerate a key, and then deleted. Therefore, they must be re-imported each time a key is needed.

Retrieving the Design Security ECC Public Key

When the 384 bit user design security private ECC key is intrinsically generated and key code #1 is stored using the JTAG or SPI programming commands, the associated 768 bit public key is also generated and stored in the system controller's private eNVM. At the same time, the256 bit SHA-256 hash of the ECC public key is enrolled to key code #2.

The 768 bit design security ECC public key can be retrieved using the following system service function:

- *MSS_SYS_puf_fetch_ecc_public_key()*

Note that the public key is also exported via JTAG (or SPI) programming commands when it is used in ECDH design security operations such as bitstream encryption or device authentication.

In order to mitigate man-in-the-middle attacks on the public key when it is exported and used, it is recommended that the ECC public key is validated using an already trusted key. This could be a user design security key, for example. Perhaps the most convenient keys to use are one of the Microsemi certified ECC key pairs, which are trusted because of the verifiable Microsemi signatures on the X.509 public key certificates. This is why when the 384 bit user ECC private key was generated and stored using key code #1, the 256 bit hash of the associated 768 bit public key was stored using key code #2. The JTAG (or SPI) key verification protocol has a special option which includes the value stored in key code #2 in the computation. Thus, besides also validating the device really "knows" the already trusted key, with this option selected, the 256-bit value stored using key code #2 is also validated. If this is the hash of the user ECC public key, then it is proved that the public key is the authentic public key as stored in the device, and not one supplied by an adversary in a man-in-the-middle attack on this key. After the user ECC public key has been thus validated, key code #2 can be deleted and replaced with another 256-bit value the user wishes to validate, such as the hash of user data security key. It is recommended that key code #2 only be used to store hashes of critical security parameters such as secret or private keys, and not the actual secrets, themselves.

Note: The user's application need to reserve a defined block of SRAM when using PUF system services, to prevent the compiler from using the SRAM memory range from address 0x2000800 to 0x2000802F inclusive. This is the default location used by the system controller to automatically enroll KC#2, when KC#1 is enrolled.

Random Number Generator Seed

The PUF's SRAM start-up value randomness can be harvested to generate a 256-bit full entropy true random seed. A random seed can be obtained using function:

- *MSS_SYS_puf_get_random_seed()*

Note: Random seeds can only be generated after the SRAM-PUF has been powered-down for 250 msec. Thus, if the PUF has been used more recently than that, there may be a delay before the seed is generated.

Tamper control service

The tamper control system services provide the following services:

- Enable/disable clock monitoring
- Control power to PUF
- Clear mesh short tamper
- Clear lock parity tamper

The tamper control services are accessed using the following functions:

- *MSS_SYS_start_clock_monitor()*
- *MSS_SYS_stop_clock_monitor()*
- *MSS_SYS_enable_puf_power_down()*
- *MSS_SYS_disable_puf_power_down()*
- *MSS_SYS_power_on_puf()*
- *MSS_SYS_clear_lock_parity()*
- *MSS_SYS_clear_mesh_short()*

Types

sys_serv_async_event_handler_t

Prototype

```
typedef void (*sys_serv_async_event_handler_t)(uint8_t event_opcode, uint8_t response);
```

Description

The *sys_serv_async_event_handler_t* type specifies the function prototype of an asynchronous event handler that can be implemented by the application to handle asynchronous messages from the System Controller. The System Controller sends asynchronous messages to the MSS COMM_BLK when certain events such as Flash*Freeze shutdown, Flash*Freeze exit and power-on-reset (POR) digest check errors are detected during the execution of the following system services:

- Flash*Freeze
- Power-on-reset (POR) digest check

The asynchronous event handler function is registered with the System Services driver through the *MSS_SYS_init()* function. The driver calls the asynchronous event handler when the MSS COMM_BLK receives an asynchronous message.

The *event_opcode* parameter is one of the asynchronous event codes described in Table 3.

The *response* parameter is the error flags byte from the POR digest check error message. The error flags are described in Table 8.

comblk_page_handler_t

Prototype

```
typedef uint32_t (*comblk_page_handler_t)(uint8_t const ** pp_next_page);
```

Description

The *comblk_page_handler_t* typedef specifies the function prototype of the in-system programming (ISP) page read handler that is passed as a parameter to the *MSS_SYS_start_isp()* function. The page read handler function must be implemented by the application program to provide the System Services driver with the address of the next page of programming data to be sent to the System Controller and the number of bytes contained in the next page.

The *pp_next_page* parameter is a pointer to the memory location containing the address of the next page of programming data. The page read handler function must update the content of that memory location to contain a pointer to the next page of programming data. The *pp_next_page* parameter points to the page's location pointer.

The page read handler function must provide the number of bytes contained in the next page as its return value. It must return a value of zero to indicate that all programming data has been passed to the System Controller.

sys_serv_isp_complete_handler_t

Prototype

```
typedef void (*sys_serv_isp_complete_handler_t)(uint32_t status);
```

Description

The *sys_serv_isp_complete_handler_t* typedef specifies the function prototype of the in-system programming (ISP) completion handler that is passed as a parameter to the *MSS_SYS_start_isp()* function. The ISP completion handler function must be implemented by the application program and it is called by the System Services driver when an ISP operation initiated by a call to *MSS_SYS_start_isp()* completes. The

ISP completion handler function receives a *status* parameter indicating the outcome of the ISP operation. Refer to the description of the *MSS_SYS_start_isp()* function for more details.

Constant Values

Asynchronous Event Handler Options

This constant is used as parameter to the *MSS_SYS_init()* function to indicate that the application code does not supply an asynchronous event handler function.

Table 1 · Asynchronous Event Handler Options

Constant	Description
MSS_SYS_NO_EVENT_HANDLER	Indicates that the application does not supply an asynchronous event a handler.

Generic Status Codes

These constants are used by multiple services to communicate the outcome of a system services request. These status codes are used across all types of services.

Table 2 · Generic Status Codes

Constant	Description
MSS_SYS_SUCCESS	Indicates that the system services completed successfully.
MSS_SYS_UNEXPECTED_ERROR	Indicates that the system failed in an unexpected way.
MSS_SYS_MEM_ACCESS_ERROR	Indicates that the System Controller could not access the memory used to pass parameters to the System Controller or to return a service result to the Cortex-M3.
MSS_SYS_SERVICE_NOT_LICENSED	Indicates that the SmartFusion2 device is not licensed to provide this service.
MSS_SYS_SERVICE_DISABLED_BY_FACTORY	Indicates that the requested system service is not available on the SmartFusion2 device.
MSS_SYS_SERVICE_DISABLED_BY_USER	Indicates that the requested system service has been disabled as part of the hardware design.
MSS_SYS_CLK_DIVISOR_ERROR	Indicates that on M2S060 devices, the divisor values for fclk, pclk0, pclk1, clk_fic64 are not equal to each other or the divisor values is set to divide by 32.

Asynchronous Event Codes

These constants are used to specify the *event_opcode* parameter for the *event_handler()* function registered with the *MSS_SYS_init()* function. They are used to specify which asynchronous event is notified to the Cortex-M3 software by the System Controller. Asynchronous events are sent by the System Controller to the Cortex-M3 when some system events of interest occur.

Table 3 · Asynchronous Event Codes

Constant	Description
FLASH_FREEZE_SHUTDOWN_OPCODE	Indicates that the system is being shutdown as a result of entering the Flash*Freeze mode.
FLASH_FREEZE_EXIT_OPCODE	Indicates that the system is exiting Flash*Freeze mode.
POR_DIGEST_ERROR_OPCODE	Indicates that the MSS has received a power-on-reset (POR) digest check error message.
TAMPER_ATTEMPT_BOUNDARY_SCAN_OPCODE	Indicates that the MSS has received the tamper boundary scan attempt detected.
TAMPER_ATTEMPT_BUFFER_ACCESS_OPCODE	Indicates that the MSS has received the tamper buffer access attempt detected.
TAMPER_ATTEMPT_DEBUG_OPCODE	Indicates that the MSS has received the tamper debug attempt detected.
TAMPER_ATTEMPT_CHECK_DIGESTS_OPCODE	Indicates that the MSS has received the tamper check digest attempt detected.
TAMPER_ATTEMPT_ECC_SETUP_INSTRUCTION_OPCODE	Indicates that the MSS has received the tamper ECC setup instruction attempt detected.
TAMPER_ATTEMPT_FACTORY_PRIVATE_OPCODE	Indicates that the MSS has received the tamper factory private attempt detected.
TAMPER_ATTEMPT_KEY_VALIDATION_OPCODE	Indicates that the MSS has received the tamper key validation attempt detected.
TAMPER_ATTEMPT_MISC_OPCODE	Indicates that the MSS has received the tamper misc attempt detected.
TAMPER_ATTEMPT_PASSCODE_MATCH_OPCODE	Indicates that the MSS has received the tamper passcode match attempt detected.
TAMPER_ATTEMPT_PASSCODE_SETUP_INSTRUCTION_OPCODE	Indicates that the MSS has received the tamper passcode setup instruction attempt detected.
TAMPER_ATTEMPT_PROGRAMMING_OPCODE	Indicates that the MSS has received the tamper programming attempt detected.
TAMPER_ATTEMPT_PUBLIC_INFORMATION_OPCODE	Indicates that the MSS has received the tamper public information attempt detected.
TAMPER_ATTEMPT_PUF_KEY_MANAGEMENT_OPCODE	Indicates that the MSS has received the tamper PUF key management attempt detected.

Constant	Description
TAMPER_ATTEMPT_UNUSED_OPCODE	Indicates that the MSS has received the tamper unused attempt detected.
TAMPER_ATTEMPT_USER_JTAG_OPCODE	Indicates that the MSS has received the tamper user JTAG attempt detected.
TAMPER_ATTEMPT_ZEROIZATION_RECOVERY_OPCODE	Indicates that the MSS has received the tamper zeroization recovery attempt detected.
TAMPER_FAILURE_BOUNDARY_SCAN_OPCODE	Indicates that the MSS has received the tamper boundary scan failure detected.
TAMPER_FAILURE_BUFFER_ACCESS_OPCODE	Indicates that the MSS has received the tamper buffer access failure detected.
TAMPER_FAILURE_DEBUG_OPCODE	Indicates that the MSS has received the tamper debug failure detected.
TAMPER_FAILURE_CHECK_DIGESTS_OPCODE	Indicates that the MSS has received the tamper check digest failure detected.
TAMPER_FAILURE_ECC_SETUP_INSTRUCTION_OPCODE	Indicates that the MSS has received the tamper ECC setup instruction failure detected.
TAMPER_FAILURE_FACTORY_PRIVATE_OPCODE	Indicates that the MSS has received the tamper factory private failure detected.
TAMPER_FAILURE_KEY_VALIDATION_OPCODE	Indicates that the MSS has received the tamper key validation failure detected.
TAMPER_FAILURE_MISC_OPCODE	Indicates that the MSS has received the tamper misc failure detected.
TAMPER_FAILURE_PASSCODE_MATCH_OPCODE	Indicates that the MSS has received the tamper passcode match failure detected.
TAMPER_FAILURE_PASSCODE_SETUP_INSTRUCTION_OPCODE	Indicates that the MSS has received the tamper passcode setup instruction failure detected.
TAMPER_FAILURE_PROGRAMMING_OPCODE	Indicates that the MSS has received the tamper programming failure detected.
TAMPER_FAILURE_PUBLIC_INFORMATION_OPCODE	Indicates that the MSS has received the tamper public information failure detected.
TAMPER_FAILURE_PUF_KEY_MANAGEMENT_OPCODE	Indicates that the MSS has received the tamper PUF key management failure detected.
TAMPER_FAILURE_UNUSED_OPCODE	Indicates that the MSS has received the tamper unused failure detected.
TAMPER_FAILURE_USER_JTAG_OPCODE	Indicates that the MSS has received the tamper user jtag failure detected.

Constant	Description
TAMPER_FAILURE_ZEROIZATION_RECOVERY_OPCODE	Indicates that the MSS has received the tamper zeroization recovery failure detected.
TAMPER_CLOCK_ERROR_DETECT_OPCODE	Indicates that the MSS has received the tamper clock monitor error detected.
TAMPER_HARDWARE_MONITOR_JTAGACTIVE_ERROR_OPCODE	Indicates that the MSS has received the tamper jtag active hardware monitor error detected.
TAMPER_HARDWARE_MONITOR_LOCKPARITY_ERROR_OPCODE	Indicates that the MSS has received the tamper lock parity hardware monitor error detected.
TAMPER_HARDWARE_MONITOR_MESHSHORT_ERROR_OPCODE	Indicates that the MSS has received the tamper mesh short hardware monitor error detected.

AES Cryptography Operation Modes

These constants are used to specify the *mode* parameter for the *MSS_SYS_128bit_aes()* and *MSS_SYS_256bit_aes()* functions.

Table 4 · Cryptography Modes

Constant	Description
MSS_SYS_ECB_ENCRYPT	Indicates that the cryptography function should perform encryption using the Electronic Codebook (ECB) mode.
MSS_SYS_ECB_DECRYPT	Indicates that the cryptography function should perform decryption using the Electronic Codebook (ECB) mode.
MSS_SYS_CBC_ENCRYPT	Indicates that the cryptography function should perform encryption using the Cipher-Block Chaining (CBC) mode.
MSS_SYS_CBC_DECRYPT	Indicates that the cryptography function should perform decryption using the Cipher-Block Chaining (CBC) mode.
MSS_SYS_OFB_ENCRYPT	Indicates that the cryptography function should perform encryption using the Output Feedback (OFB) mode.
MSS_SYS_OFB_DECRYPT	Indicates that the cryptography function should perform decryption using the Output Feedback (OFB) mode.
MSS_SYS_CTR_ENCRYPT	Indicates that the cryptography function should perform encryption using the Counter (CTR) mode.
MSS_SYS_CTR_DECRYPT	Indicates that the cryptography function should perform decryption using the Counter (CTR) mode.

Non-Deterministic Random Bit Generator Status Codes

These constants are used by non deterministic random bit generator (NRBG) services to communicate the outcome of a system services request. These status codes are only used by NRBG services.

Table 5 · NRBG Status Codes

Constant	Description
`MSS_SYS_NRBG_CATASTROPHIC_ERROR`	Indicates that a catastrophic error occurred.
`MSS_SYS_NRBG_MAX_INST_EXCEEDED`	Indicates that the maximum number of NRBG instances has been exceeded. You need to release already instantiated NRBG instances using the *MSS_SYS_nrbg_uninstantiate()* function.
`MSS_SYS_NRBG_INVALID_HANDLE`	Indicates that the *handle* parameter has an invalid value.
`MSS_SYS_NRBG_GEN_REQ_TOO_BIG`	Indicates that the requested random number is too long. The requested length is larger than the maximum number of digits that can be generated.
`MSS_SYS_NRBG_MAX_LENGTH_EXCEEDED`	Indicates that the supplied additional data length is exceeded.

Flash*Freeze Options

These constants are used to specify the *options* parameter for the *MSS_SYS_flash_freeze()* function.

Table 6 · Flash*Freeze Options

Constant	Description
`MSS_SYS_FPGA_POWER_DOWN`	Indicates that the *MSS_SYS_flash_freeze()* function should request the FPGA fabric to enter Flash*Freeze mode.
`MSS_SYS_MPLL_POWER_DOWN`	Indicates that the *MSS_SYS_flash_freeze()* function should request the MSS PLL to enter Flash*Freeze mode.

Programming Modes

These constants are used to specify the *mode* parameter for the *MSS_SYS_start_isp()* function.

Table 7 · Programming Modes

Constant	Description
`MSS_SYS_PROG_AUTHENTICATE`	Indicates that the *MSS_SYS_start_isp()* function should request the System Controller to start the ISP service in authenticate mode
`MSS_SYS_PROG_PROGRAM`	Indicates that the *MSS_SYS_start_isp()* function should request the System Controller to start the ISP service in program mode
`MSS_SYS_PROG_VERIFY`	Indicates that the *MSS_SYS_start_isp()* function should request the System Controller to start the ISP service in verify mode

Digest Check Options

These constants are used to specify the *options* parameter for the *MSS_SYS_check_digest()* function. They can also be used as bitmasks for the return value of the *MSS_SYS_check_digest()* function to test for digest mismatch errors and identify which NVM component has a mismatch.

For the POR digest check service, these constants are used to specify the *response* parameter for the *event_handler()* function registered with the *MSS_SYS_init()* function. In this role, the *response* parameter indicates which NVM component has a digest mismatch error.

Table 8 · Digest Check Options/Error Flags

Constant	Description
`MSS_SYS_DIGEST_CHECK_FABRIC`	Indicates that the *MSS_SYS_check_digest()* function should request a digest check for the FPGA fabric (bit mask = 0x01)
`MSS_SYS_DIGEST_CHECK_ENVM0`	Indicates that the *MSS_SYS_check_digest()* function should request a digest check for eNVM block 0 (bit mask = 0x02)
`MSS_SYS_DIGEST_CHECK_ENVM1`	Indicates that the *MSS_SYS_check_digest()* function should request a digest check for eNVM block 1 (bit mask = 0x04)
`MSS_SYS_DIGEST_CHECK_SYS`	Indicates that the *MSS_SYS_check_digest()* function should request a digest check for System Controller ROM (bit mask = 0x08)
`MSS_SYS_DIGEST_CHECK_ENVMFP`	Indicates that the *MSS_SYS_check_digest()* function should request a digest check for private factory eNVM (bit mask = 0x10)
`MSS_SYS_DIGEST_CHECK_ENVMUP`	Indicates that the *MSS_SYS_check_digest()* function should request a digest check for private user eNVM (bit mask = 0x20)

PUF Service Status Codes

These constants are used by PUF services to communicate the outcome of a system services request. These status codes are only used by PUF services.

Table 9 · PUF services status codes

Constant	Description
`MSS_SYS_ENVM_ERROR`	Indicates that the eNVM error occurred for both create and delete user activation code sub-command.
`MSS_SYS_PUF_ERROR_WHEN_CREATING`	Indicates that PUF error occur while creating new user activation code.
`MSS_SYS_INVALID_SUBCMD`	Indicates that the sub-command is invalid.
`MSS_SYS_INVALID_REQUEST_OR_KC`	Indicates that request or Key code is invalid, when exporting or importing.
`MSS_SYS_INVALID_KEYNUM_OR_ARGUMENT`	Indicates that the supplied key number or argument is invalid.
`MSS_SYS_NO_VALID_PUBLIC_KEY`	Indicates that no valid public key present in eNVM.
`MSS_SYS_INVALID_MEMORY_ADDRESS`	Indicates that memory address is invalid.
`MSS_SYS_ENVM_PROGRAM_ERROR`	Indicates that the eNVM program error occur when writing to the private eNVM for both create and delete user activation code sub command.
`MSS_SYS_INVALID_HASH`	Indicates that 32 byte hash is invalid.
`MSS_SYS_INVALID_USER_AC1`	Indicates that invalid user activation code has been imported.
`MSS_SYS_ENVM_VERIFY_ERROR`	Indicates that the eNVM verify error occur when writing to the private eNVM for both create and delete user activation code sub command.
`MSS_SYS_INCORRECT_KEYSIZE_FOR_RENEWING_A_KC`	Indicates that the supplied key size is incorrect while renewing the key code.
`MSS_SYS_PRIVATE_ENVM_USER_DIGEST_MISMATCH`	Indicates that digest mismatch occurs for private the eNVM.
`MSS_SYS_USER_KEY_CODE_INVALID_SUBCMD`	Indicates that the sub-command is invalid.
`MSS_SYS_DRBG_ERROR`	Indicates that DRBG error occurred while populating one time pad reminder by random bits.

Functions

MSS_SYS_init

Prototype

```
void MSS_SYS_init
(
    sys_serv_async_event_handler_t event_handler
);
```

Description

The *MSS_SYS_init()* function initializes the system services communication with the System Controller.

Parameters

event_handler

The *event_handler* parameter specifies an optional asynchronous event handler function. This event handler function is provided by the application. It will be called by the System Services driver whenever an asynchronous event is received from the SmartFusion2 System Controller. This event handler is typically used to handle entry and exit of Flash*Freeze mode and power-on-reset (POR) digest check error messages.

Note: If the application does not provide an asynchronous event handler then the *event_handler* parameter must be set to MSS_SYS_NO_EVENT_HANDLER.

Return Value

This function does not return a value.

MSS_SYS_get_serial_number

Prototype

```
uint8_t MSS_SYS_get_serial_number
(
    uint8_t * p_serial_number
);
```

Description

The *MSS_SYS_get_serial_number()* function fetches the 128-bit device serial number (DSN).

Parameters

p_serial_number

The *p_serial_number* parameter is a pointer to the 16-byte buffer where the serial number will be written by this system service.

Return Value

The *MSS_SYS_get_serial_number()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_UNEXPECTED_ERROR

MSS_SYS_get_design_version

Prototype

```
uint8_t MSS_SYS_get_design_version
(
    uint8_t * p_design_version
);
```

Description

The *MSS_SYS_get_design_version()* function fetches the design version.

Parameters

p_design_version

The *p_design_version* parameter is a pointer to the 2-byte buffer where the design version will be written by this system service.

Return Value

The *MSS_SYS_get_design_version()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_UNEXPECTED_ERROR

MSS_SYS_get_device_certificate

Prototype

```
uint8_t MSS_SYS_get_device_certificate
(
    uint8_t * p_device_certificate
);
```

Description

The *MSS_SYS_get_device_certificate()* function fetches the device certificate.

Parameters

p_device_certificate

The *p_device_certificate* parameter is a pointer to the 512-byte buffer where the device certificate will be written by this system service.

Return Value

The *MSS_SYS_get_device_certificate()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_UNEXPECTED_ERROR

MSS_SYS_get_secondary_device_certificate

Prototype

```
uint8_t MSS_SYS_get_secondary_device_certificate
(
    uint8_t * p_secondary_device_certificate
);
```

Description

The *MSS_SYS_get_secondary_device_certificate()* function fetches the secondary device certificate. The secondary device certificate is the second ECC Key Certificate which is a 640-byte digitally-signed X-509 certificate programmed during manufacturing. This certificate only contains the public key for the ECC key and is otherwise unused by the device.

Parameters

p_secondary_device_certificate

The *p_secondary_device_certificate* parameter is a pointer to the 640-bytes buffer where the secondary device certificate will be written by this system service.

Return Value

The *MSS_SYS_get_secondary_device_certificate()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_UNEXPECTED_ERROR

MSS_SYS_get_user_code

Prototype

```
uint8_t MSS_SYS_get_user_code
(
    uint8_t * p_user_code
);
```

Description

The *MSS_SYS_get_user_code()* function fetches the 32-bit USERCODE.

Parameters

p_user_code

The *p_user_code* parameter is a pointer to the 4-byte buffer where the USERCODE will be written by this system service.

Return Value

The *MSS_SYS_get_user_code()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_UNEXPECTED_ERROR

MSS_SYS_nrbg_instantiate

Prototype

```
uint8_t MSS_SYS_nrbg_instantiate
(
    const uint8_t * personalization_str,
    uint16_t personalization_str_length,
    uint8_t * p_nrbg_handle
);
```

Description

The *MSS_SYS_nrbg_instantiate()* function instantiates a non-deterministic random bit generator (NRBG) instance. A maximum of two concurrent user instances are available.

Parameters

personalization_str

The *personalization_str* parameter is a pointer to a buffer containing a random bit generator personalization string. The personalization string can be up to 128 bytes long.

personalization_str_length

The *personalization_str_length* parameter specifies the byte length of the personalization string pointed to by *personalization_str*.

p_nrbg_handle

The *p_nrbg_handle* parameter is a pointer to a byte that will contain the handle of the instantiated NRBG if this function call succeeds.

Return Value

The *MSS_SYS_nrbg_instantiate()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_NRBG_CATASTROPHIC_ERROR
- MSS_SYS_NRBG_MAX_INST_EXCEEDED
- MSS_SYS_NRBG_INVALID_HANDLE
- MSS_SYS_NRBG_GEN_REQ_TOO_BIG
- MSS_SYS_NRBG_MAX_LENGTH_EXCEEDED
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_UNEXPECTED_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER

MSS_SYS_nrbg_uninstantiate

Prototype

```
uint8_t MSS_SYS_nrbg_uninstantiate
(
    uint8_t nrbg_handle
);
```

Description

The *MSS_SYS_nrbg_uninstantiate()* function releases the non-deterministic random bit generator (NRBG) identified by the *nrbg_handle* parameter.

Parameters

nrbg_handle

The *nrbg_handle* parameter specifies which NRBG instance will be released. The value of *nrbg_handle* is obtained as a result of a previous call to the *MSS_SYS_nrbg_instantiate()* function.

Return Value

The *MSS_SYS_nrbg_uninstantiate()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_NRBG_CATASTROPHIC_ERROR
- MSS_SYS_NRBG_MAX_INST_EXCEEDED
- MSS_SYS_NRBG_INVALID_HANDLE
- MSS_SYS_NRBG_GEN_REQ_TOO_BIG
- MSS_SYS_NRBG_MAX_LENGTH_EXCEEDED
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_UNEXPECTED_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER

MSS_SYS_nrbg_self_test

Prototype

```
uint8_t MSS_SYS_nrbg_self_test(void);
```

Description

The *MSS_SYS_nrbg_self_test()* function performs a self test of the non-deterministic random bit generator (NRBG).

Return Value

The *MSS_SYS_nrbg_self_test()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_NRBG_CATASTROPHIC_ERROR
- MSS_SYS_NRBG_MAX_INST_EXCEEDED
- MSS_SYS_NRBG_INVALID_HANDLE
- MSS_SYS_NRBG_GEN_REQ_TOO_BIG
- MSS_SYS_NRBG_MAX_LENGTH_EXCEEDED
- MSS_SYS_MEM_ACCESS_ERROR

- MSS_SYS_UNEXPECTED_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER

MSS_SYS_nrbg_reseed

Prototype

```
uint8_t MSS_SYS_nrbg_reseed
(
    const uint8_t * p_additional_input,
    uint8_t additional_input_length,
    uint8_t nrbg_handle
);
```

Description

The *MSS_SYS_nrbg_reseed()* function is used to reseed the non-deterministic random bit generator (NRBG) identified by the *nrbg_handle* parameter.

Parameters

p_additional_input

The *additional_input_length* parameter specifies the number of additional input bytes used to reseed the NRBG identified by the *nrbg_handle* parameter.

additional_input_length

The *additional_input_length* parameter specifies the number of additional input bytes used to reseed the NRBG.

nrbg_handle

The *nrbg_handle* parameter specifies which NRBG instance to reseed. The value of *nrbg_handle* is obtained as a result of a previous call to the *MSS_SYS_nrbg_instantiate()* function.

Return Value

The *MSS_SYS_nrbg_reseed()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_NRBG_CATASTROPHIC_ERROR
- MSS_SYS_NRBG_MAX_INST_EXCEEDED
- MSS_SYS_NRBG_INVALID_HANDLE
- MSS_SYS_NRBG_GEN_REQ_TOO_BIG
- MSS_SYS_NRBG_MAX_LENGTH_EXCEEDED
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_UNEXPECTED_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER

MSS_SYS_nrbg_generate

Prototype

```
uint8_t MSS_SYS_nrbg_generate
(
    const uint8_t * p_requested_data,
    const uint8_t * p_additional_input,
    uint8_t requested_length,
    uint8_t additional_input_length,
    uint8_t pr_req,
    uint8_t nrbg_handle
);
```

Description

The *MSS_SYS_nrbg_generate()* function generates a random bit sequence which can be up to 128 bytes long.

Parameters

p_requested_data

The *p_requested_data* parameter is a pointer to the buffer where the requested random data will be stored on completion of this system service.

p_additional_input

The *p_additional_input* parameter is a pointer to the buffer containing additional input data for the random bit generation.

requested_length

The *requested_length* parameter specifies the number of random data bytes requested to be generated. The maximum generated data length is 128 bytes.

additional_input_length

The *additional_input_length* parameter specifies the number of additional input bytes to use in the random data generation.

pr_req

The *pr_req* parameter specifies if prediction resistance is requested.

nrbg_handle

The *nrbg_handle* parameter specifies which non-deterministic random bit generator (NRBG) instance will be used to generate the random data. The value of *nrbg_handle* is obtained as a result of a previous call to the *MSS_SYS_nrbg_instantiate()* function.

Return Value

The *MSS_SYS_nrbg_generate()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_NRBG_CATASTROPHIC_ERROR
- MSS_SYS_NRBG_MAX_INST_EXCEEDED
- MSS_SYS_NRBG_INVALID_HANDLE
- MSS_SYS_NRBG_GEN_REQ_TOO_BIG
- MSS_SYS_NRBG_MAX_LENGTH_EXCEEDED
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_UNEXPECTED_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED

- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER

MSS_SYS_nrbg_reset

Prototype

```
uint8_t MSS_SYS_nrbg_reset
(
    void
);
```

Description

The *MSS_SYS_nrbg_reset()* function is used to reset the non-deterministic random bit generator (NRBG). The reset service removes all NRBG instantiations and resets the NRBG. This service is the only mechanism by which to recover from a catastrophic NRBG error without physically resetting the device. All active instantiations are automatically destroyed.

Return Value

The *MSS_SYS_nrbg_reset()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_NRBG_CATASTROPHIC_ERROR
- MSS_SYS_NRBG_MAX_INST_EXCEEDED
- MSS_SYS_NRBG_INVALID_HANDLE
- MSS_SYS_NRBG_GEN_REQ_TOO_BIG
- MSS_SYS_NRBG_MAX_LENGTH_EXCEEDED
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_UNEXPECTED_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER

MSS_SYS_128bit_aes

Prototype

```
uint8_t MSS_SYS_128bit_aes
(
    const uint8_t * key,
    const uint8_t * iv,
    uint16_t nb_blocks,
    uint8_t mode,
    uint8_t * dest_addr,
    const uint8_t * src_addr
);
```

Description

The *MSS_SYS_128bit_aes()* function provides access to the SmartFusion2 AES-128 cryptography service.

Parameters

key

The *key* parameter is a pointer to a 16-byte array containing the key to use for the requested encryption/decryption operation.

iv

The *iv* parameter is a pointer to a 16-byte array containing the initialization vector that will be used as part of the requested encryption/decryption operation. Its use is different depending on the mode.

Mode	Usage
ECB	Ignored.
CBC	Randomization.
OFB	Randomization.
CTR	Used as initial counter value.

nb_blocks

The *nb_blocks* parameter specifies the number of 128-bit blocks of plaintext/ciphertext to be processed by the AES-128 system service.

mode

The *mode* parameter specifies the cipher mode of operation and whether the source text must be encrypted or decrypted. The modes of operation are:

- Electronic Codebook (ECB)
- Cipher-Block Chaining (CBC)
- Output Feedback (OFB)
- Counter (CTR) – The CTR mode uses the content of the initialization vector as its initial counter value.

The counter increment is 2^64. Allowed values for the mode parameter are:

- MSS_SYS_ECB_ENCRYPT
- MSS_SYS_ECB_DECRYPT
- MSS_SYS_CBC_ENCRYPT
- MSS_SYS_CBC_DECRYPT

- MSS_SYS_OFB_ENCRYPT
- MSS_SYS_OFB_DECRYPT
- MSS_SYS_CTR_ENCRYPT
- MSS_SYS_CTR_DECRYPT

dest_addr

The *dest_addr* parameter is a pointer to the memory buffer where the result of the encryption/decryption operation will be stored.

src_addr

The *src_addr* parameter is a pointer to the memory buffer containing the source plaintext/ciphertext to be encrypted/decrypted.

Return Value

The *MSS_SYS_128bit_aes()* function returns one of following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_UNEXPECTED_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER

MSS_SYS_256bit_aes

Prototype

```
uint8_t MSS_SYS_256bit_aes
(
    const uint8_t * key,
    const uint8_t * iv,
    uint16_t nb_blocks,
    uint8_t mode,
    uint8_t * dest_addr,
    const uint8_t * src_addr
);
```

Description

The *MSS_SYS_256bit_aes()* function provides access to the SmartFusion2 AES-256 cryptography service.

Parameters

key

The *key* parameter is a pointer to a 32-byte array containing the key to use for the requested encryption/decryption operation.

iv

The *iv* parameter is a pointer to a 16-byte array containing the initialization vector that will be used as part of the requested encryption/decryption operation. Its use is different depending on the mode.

Mode	Usage
ECB	Ignored.
CBC	Randomization.
OFB	Randomization.
CTR	Used as initial counter value.

nb_blocks

The *nb_blocks* parameter specifies the number of 128-bit blocks of plaintext/ciphertext requested to be processed by the AES-128 system service.

mode

The *mode* parameter specifies the cipher mode of operation and whether the source text must be encrypted or decrypted. The modes of operation are:

- Electronic Codebook (ECB)
- Cipher-Block Chaining (CBC)
- Output Feedback (OFB)
- Counter (CTR) – The CTR mode uses the content of the initialization vector as its initial counter value.

The counter increment is 2^64. Allowed values for the mode parameter are:

- MSS_SYS_ECB_ENCRYPT
- MSS_SYS_ECB_DECRYPT
- MSS_SYS_CBC_ENCRYPT
- MSS_SYS_CBC_DECRYPT

- MSS_SYS_OFB_ENCRYPT
- MSS_SYS_OFB_DECRYPT
- MSS_SYS_CTR_ENCRYPT
- MSS_SYS_CTR_DECRYPT

dest_addr

The *dest_addr* parameter is a pointer to the memory buffer where the result of the encryption/decryption operation will be stored.

src_addr

The *src_addr* parameter is a pointer to the memory buffer containing the source plaintext/ciphertext to be encrypted/decrypted.

Return Value

The *MSS_SYS_256bit_aes()* function returns one of following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_UNEXPECTED_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER

MSS_SYS_sha256

Prototype

```
uint8_t MSS_SYS_sha256
(
    const uint8_t * p_data_in,
    uint32_t length,
    uint8_t * result
);
```

Description

The *MSS_SYS_sha256()* function provides access to the SmartFusion2 SHA-256 cryptography service.

Parameters

p_data_in

The *p_data_in* parameter is a pointer to the memory location containing the data that will be hashed using the SHA-256 system service.

length

The *length* parameter specifies the length in bits of the data to hash.

result

The *result* parameter is a pointer to a 32-byte buffer where the hash result will be stored.

Return Value

The *MSS_SYS_sha256()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_UNEXPECTED_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER

MSS_SYS_hmac

Prototype

```
uint8_t MSS_SYS_hmac
(
    const uint8_t * key,
    const uint8_t * p_data_in,
    uint32_t length,
    uint8_t * p_result
);
```

Description

The *MSS_SYS_hmac()* function provides access to the SmartFusion2 HMAC cryptography service. The HMAC system service generates message authentication codes using the SHA-256 hash function.

Parameters

key

The *key* parameter is a pointer to a 32-byte array containing the key used to generate the message authentication code.

p_data_in

The *p_data_in* parameter is a pointer to the data to be authenticated.

length

The *length* parameter specifies the number of data bytes for which to generate the authentication code. It is the size of the data pointed to by the p_data_in parameter.

p_result

The *p_result* parameter is a pointer to a 32-byte buffer where the authentication code generated by the HMAC system service will be stored.

Return Value

The *MSS_SYS_hmac()* function returns one of following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_UNEXPECTED_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER

MSS_SYS_key_tree

Prototype

```
uint8_t MSS_SYS_key_tree
(
    uint8_t* p_key,
    uint8_t op_type,
    const uint8_t* path
);
```

Description

The *MSS_SYS_key_tree()* function provides access to a SHA-256 based key-tree cryptography algorithm. The key-tree service begins with a user-supplied root key and derives an output key based on a 7-bit parameter which can be used to create uniqueness for different applications using the same root key, and a 128-bit path variable. Both the 7-bit input parameter and the 128-bit path variable are assumed to be publicly known. One common use for the output key is as a keyed validator, similar to a message authentication code tag. Because of the key tree protocol, the root, intermediate, and final key values are protected against differential power analysis.

Parameters

p_key

The *p_key* parameter is a pointer to a 32-byte array containing a root key that is used by the key-tree service to generate an output key. This 32-byte array is also used to return the generated output key.

op_type

The *op_type* parameter specifies the 7-bit input parameter to be used to generate the output key.

path

The *path* parameter specifies the 128-bit path variable to be used to generate the output key.

Return Value

The *MSS_SYS_key_tree()* function returns one of following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_UNEXPECTED_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_SERVICE_DISABLED_BY_USER

Example

The following example demonstrates how to use key-tree service to generate the output key.

```
uint8_t g_key[32] = {0x00, 0x00, 0x00, 0x00, 0x11, 0x11, 0x11, 0x11,
                     0x22, 0x22, 0x22, 0x22, 0x33, 0x33, 0x33, 0x33,
                     0x44, 0x44, 0x44, 0x44, 0x55, 0x55, 0x55, 0x55,
                     0x66, 0x66, 0x66, 0x66, 0x77, 0x77, 0x77, 0x77};
uint8_t g_optype = 0x00;
uint8_t g_path[16] = {0x00, 0x00, 0x00, 0x00, 0x11, 0x11, 0x11, 0x11,
                      0x22, 0x22, 0x22, 0x22, 0x33, 0x33, 0x33, 0x33};

MSS_SYS_key_tree(g_key, g_optype, g_path);
```

MSS_SYS_challenge_response

Prototype

```
uint8_t MSS_SYS_challenge_response
(
    uint8_t* p_key,
    uint8_t op_type,
    const uint8_t* path
);
```

Description

The *MSS_SYS_challenge_response()* function accepts a challenge comprising a 7-bit optype and a 128-bit path and return a 256-bit response unique to the given challenge and the device. This function is also sometimes referred to as “PUF Emulation” since its outputs are unique for each device, and in the devices in the FPGA family having an SRAM-PUF the response value depends on the intrinsic PUF secret.

Parameters

p_key

The *key* parameter is a pointer to a 32-byte array containing the 256-bit unique response to the given challenge.

op_type

The *op_type* parameter specifies the 7-bit input parameter to be used to generate the 256-bit unique response.

path

The *path* parameter specifies the 128-bit path variable to be used to generate the 256-bit unique response.

Return Value

The *MSS_SYS_challenge_response()* function returns one of following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_UNEXPECTED_ERROR
- MSS_ SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER

Example

The following example demonstrates how to use the Challenge-Response system service to authenticate a device.

```
uint8_t g_key[32] = {0x00};
uint8_t g_optype = 0x00;
uint8_t g_path[16] = {0x00, 0x00, 0x00, 0x00, 0x11, 0x11, 0x11, 0x11,
                      0x22, 0x22, 0x22, 0x22, 0x33, 0x33, 0x33, 0x33};

status = MSS_SYS_challenge_response(g_key, g_optype, g_path);
```

MSS_SYS_flash_freeze

Prototype

```
uint8_t MSS_SYS_flash_freeze
(
    uint8_t options
);
```

Description

The *MSS_SYS_flash_freeze()* function requests the SmartFusion2 device to enter Flash*Freeze mode. The System Controller sends an asynchronous message to the MSS COMM_BLK when the device is either about to enter or has exited Flash*Freeze mode. The *MSS_SYS_init()* function can register a callback function with the System Services driver to handle these asynchronous messages.

Parameters

options

The *options* parameter can be used to power down additional parts of SmartFusion2 when the FPGA fabric enters Flash*Freeze mode. This parameter is a bit mask of the following options:

- MSS_SYS_FPGA_POWER_DOWN
- MSS_SYS_MPLL_POWER_DOWN

Note: MSS_SYS_FPGA_POWER_DOWN on its own will only power down the FPGA fabric. MSS_SYS_MPLL_POWER_DOWN specifies that the MSS PLL is powered down during the Flash*Freeze period.

Return Value

The *MSS_SYS_flash_freeze()* function returns one of following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_UNEXPECTED_ERROR
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER
- MSS_SYS_CLK_DIVISOR_ERROR

Example

The following example demonstrates how to request the FPGA fabric to enter Flash*Freeze mode:

```
MSS_SYS_flash_freeze(MSS_SYS_FPGA_POWER_DOWN | MSS_SYS_MPLL_POWER_DOWN);
```

MSS_SYS_zeroize_device

Prototype

```
void MSS_SYS_zeroize_device
(
    void
);
```

Description

The *MSS_SYS_zeroize_device()* function is used to destroy sensitive information stored on the SmartFusion2 device. The zeroization system service erases all user configuration data, user keys, user security settings, NVM, SRAM, FPGA flip-flops, System Controller memory, and crypto-engine registers. The zeroization system service is enabled and the level of information to be destroyed is configured in the Libero hardware flow of the design programmed into the device.

Note: The zeroization system service can render the SmartFusion2 device permanently and irrevocably disabled depending on the configuration selected in the Libero hardware flow.

Parameters

This function does not take any parameters.

Return Value

This function does not return a value.

MSS_SYS_start_isp

Prototype

```
uint8_t MSS_SYS_start_isp
(
    uint8_t mode,
    comblk_page_handler_t page_read_handler,
    sys_serv_isp_complete_handler_t isp_completion_handler
);
```

Description

The *MSS_SYS_start_isp()* function starts the In-System Programming (ISP) system service which allows the Cortex-M3 processor to directly provide a bitstream for programming the SmartFusion2 device.

The ISP system service can perform these services:

- Authenticate a programming bitstream
- Program the device with a programming bitstream
- Verify that a programming bitstream has been correctly programmed

The *mode* parameter specifies the ISP system service to perform. The application must also provide two functions as parameters to the *MSS_SYS_start_isp()* function. The first function is used by the ISP system service to read the programming bitstream. The second function is used by the ISP system service to notify the application that the ISP system service has completed.

Parameters

mode

The *mode* parameter specifies the ISP system service to perform. It can be one of:

- MSS_SYS_PROG_AUTHENTICATE
- MSS_SYS_PROG_PROGRAM
- MSS_SYS_PROG_VERIFY

page_read_handler

The *page_read_handler* parameter is a pointer to a function with the following prototype:

```
uint32_t (*comblk_page_handler_t)(uint8 const** pp_next_page);
```

The *page_read_handler* function must be implemented by the application program to provide the System Services driver with the address of the next page of programming data to be sent to the System Controller and the number of bytes contained in the next page. Refer to the description of the *comblk_page_handler_t* type for details of the *page_read_handler* function's parameters and return values.

isp_completion_handler

The *isp_completion_handler* parameter is a pointer to a function with the following prototype.

```
void (*sys_serv_isp_complete_handler_t)(uint32_t status);
```

The *isp_completion_handler* function must be implemented by the application program and it is called by the System Services driver when an ISP operation initiated by a call to *MSS_SYS_start_isp()* completes.

The *isp_completion_handler* function receives one of the following status codes through its *status* parameter indicating the outcome of the ISP operation:

- MSS_SYS_SUCCESS
- MSS_SYS_CHAINING_MISMATCH
- MSS_SYS_UNEXPECTED_DATA_RECEIVED
- MSS_SYS_INVALID_ENCRYPTION_KEY
- MSS_SYS_INVALID_COMPONENT_HEADER
- MSS_SYS_BACK_LEVEL_NOT_SATISFIED
- MSS_SYS_DSN_BINDING_MISMATCH

- MSS_SYS_ILLEGAL_COMPONENT_SEQUENCE
- MSS_SYS_INSUFFICIENT_DEV_CAPABILITIES
- MSS_SYS_INCORRECT_DEVICE_ID
- MSS_SYS_UNSUPPORTED_BITSTREAM_PROT_VER
- MSS_SYS_VERIFY_NOT_PERMITTED_ON_BITSTR
- MSS_SYS_ABORT
- MSS_SYS_NVM_VERIFY_FAILED
- MSS_SYS_DEVICE_SECURITY_PROTECTED
- MSS_SYS_PROGRAMMING_MODE_NOT_ENABLED
- MSS_SYS_SERVICE_DISABLED_BY_USER

Return Value

This function return status as MSS_SYS_CLK_DIVISOR_ERROR, if fclk, pclk0, pclk1 and clk_fic64 divisor are not equal to each other or set to divide by 32 on M2S060 device.

Example

The following example demonstrates how to use In-System Programming (ISP) system service to reprogram the device.

```
static uint32_t page_read_handler(uint8_t const **)
{………}
static void isp_completion_handler(uint32_t isp_exit_status)
{………}

Status = MSS_SYS_start_isp(MSS_SYS_PROG_AUTHENTICATE,
                  page_read_handler,
                  isp_completion_handler);
```

MSS_SYS_initiate_iap

Prototype

```
uint8_t MSS_SYS_initiate_iap
(
    uint8_t mode,
    uint32_t bitstream_spi_addr
);
```

Description

The *MSS_SYS_initiate_iap()* function invokes the In-Application Programming (IAP) system service by sending the request to the System Controller to reprogram the device using a bitstream present in SPI Flash connected to MSS SPI-0. The IAP service is initiated by a call to *MSS_SYS_initiate_iapIAP()*. The IAP system service can perform these operations:

- authenticate a programming bitstream
- program a bitstream
- verify that a programming bitstream has been correctly programmed

Parameters

Mode

The *mode* parameter specifies the IAP service to perform. It can be one of:

- MSS_SYS_PROG_AUTHENTICATE
- MSS_SYS_PROG_PROGRAM
- MSS_SYS_PROG_VERIFY

bitstream_spi_addr

The *bitstream_spi_addr* parameter specifies the starting location address of the programmed bitstream present in SPI flash connected to MSS SPI 0. The programming bitstream must be copied by the application into SPI Flash before calling the IAP system service.

Return Value

The *MSS_SYS_initiate_iap* function will receive one of the following status codes on completion of the system service:

- MSS_SYS_SUCCESS
- MSS_SYS_CHAINING_MISMATCH
- MSS_SYS_UNEXPECTED_DATA_RECEIVED
- MSS_SYS_INVALID_ENCRYPTION_KEY
- MSS_SYS_INVALID_COMPONENT_HEADER
- MSS_SYS_BACK_LEVEL_NOT_SATISFIED
- MSS_SYS_DSN_BINDING_MISMATCH
- MSS_SYS_ILLEGAL_COMPONENT_SEQUENCE
- MSS_SYS_INSUFFICIENT_DEV_CAPABILITIES
- MSS_SYS_INCORRECT_DEVICE_ID
- MSS_SYS_UNSUPPORTED_BITSTREAM_PROT_VER
- MSS_SYS_VERIFY_NOT_PERMITTED_ON_BITSTR
- MSS_SYS_INVALID_DEVICE_CERTIFICATE
- MSS_SYS_ABORT
- MSS_SYS_NVM_VERIFY_FAILED
- MSS_SYS_DEVICE_SECURITY_PROTECTED
- MSS_SYS_PROGRAMMING_MODE_NOT_ENABLED

- MSS_SYS_ENVM_PROGRAMMING_OPERATION_FAIL
- MSS_SYS_ENVM_VERIFY_OPERATION_FAIL
- MSS_SYS_ACCESS_ERROR
- MSS_SYS_PUF_ACCESS_ERROR
- MSS_SYS_BAD_COMPONENT
- MSS_SYS_SERVICE_DISABLED_BY_USER
- MSS_SYS_CLK_DIVISOR_ERROR

Example

The following example demonstrates how to use In-Application Programming (IAP) system service to reprogram the device.

```
status = MSS_SYS_initiate_iap(mode, bitstream_spi_addr);
```

MSS_SYS_check_digest

Prototype

```
uint8_t MSS_SYS_check_digest
(
    uint8_t options
);
```

Description

The *MSS_SYS_check_digest()* function is used to recalculate and compare cryptographic digests of selected NVM component(s) – FPGA fabric, eNVM0, and eNVM1 – to those previously computed and saved in NVM.

Note: The FPGA fabric will enter the Flash*Freeze state if it is powered up when its digest is checked.

Parameters

options

The *options* parameter specifies which components' digest to recalculate and check. The allowed values for the *options* parameter are any one of the following bitmask constants or a bitwise OR of more than one:

- MSS_SYS_DIGEST_CHECK_FABRIC (bit mask = 0x01)
- MSS_SYS_DIGEST_CHECK_ENVM0 (bit mask = 0x02)
- MSS_SYS_DIGEST_CHECK_ENVM1 (bit mask = 0x04)
- MSS_SYS_DIGEST_CHECK_SYS (bit mask = 0x08)
- MSS_SYS_DIGEST_CHECK_ENVMFP (bit mask = 0x10)
- MSS_SYS_DIGEST_CHECK_ENVMUP (bit mask = 0x20)

Note: Private factory eNVM and private user ENVM digest checks are only available on the M2S060 and larger devices.

Return Value

The *MSS_SYS_check_digest()* function returns the result of the digest check as an 8-bit unsigned integer. The meaning of the digest check return value is as follows:

- bit 0: Fabric digest error
- bit 1: ENVM0 digest error
- bit 2: ENVM1 digest error
- bit 3: System Controller ROM digest error
- bit 4: Private eNVM factory digest error
- bit 5: Private eNVM user digest error
- bit 7: Service Disable by user lock

A '1' in one of the above bits indicates a digest mismatch. The return value can be compared to the bitmask constants specified as allowed values for the *options* parameter to discover which of the NVM components produced a digest check error.

On M2S060 device, the *MSS_SYS_check_digest()* function returns MSS_SYS_CLK_DIVISOR_ERROR, if the the divisor values of fclk, pclk0, pclk1 and clk_fic64 divisor are not equal to each other or set to divide by 32.

Example

The following example demonstrates to use of digest check service to recalculates and compares digests of selected components.

```
MSS_SYS_check_digest(MSS_SYS_DIGEST_CHECK_FABRIC);
```

MSS_SYS_puf_create_activation_code

Prototype

```
uint8_t MSS_SYS_puf_create_activation_code
(
    void
);
```

Description

The *MSS_SYS_puf_create_activation_code()* function is used to create the SRAM-PUF user activation code. This function is typically only used once when enrolling the device into the overall system it operates in. It creates an activation code which will be used to regenerate the SRAM-PUF secret key after each power-up of the SRAM-PUF. The activation code is used to eliminate the dynamic randomness of the SRAM-PUF power-up content in order to retrieve the device's unique PUF secret key based on the more repeatable (but still initially random) bits.

Note: This system service is only available on large SmartFusion2 devices starting with the M2S060.

Parameters

This function does not take any parameter.

Return Value

The *MSS_SYS_puf_create_activation_code()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_ENVM_ERROR
- MSS_SYS_PUF_ERROR_WHEN_CREATING
- MSS_SYS_INVALID_SUBCMD
- MSS_SYS_ENVM_PROGRAM_ERROR
- MSS_SYS_ENVM_VERIFY_ERROR
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER
- MSS_SYS_UNEXPECTED_ERROR

MSS_SYS_puf_delete_activation_code

Prototype

```
uint8_t MSS_SYS_puf_delete_activation_code
(
    void
);
```

Description

The *MSS_SYS_puf_delete_activation_code()* function is used to delete the user activation code. This function would typically be used only once when decommissioning the device. Any user keys stored using the PUF mechanism will be irreversibly lost and the associated key codes will be useless.

Note: This system service is only available on large SmartFusion2 devices starting with the M2S060.

Parameters

This function does not take any parameter.

Return Value

The *MSS_SYS_puf_delete_activation_code()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_ENVM_ERROR
- MSS_SYS_PUF_ERROR_WHEN_CREATING
- MSS_SYS_INVALID_SUBCMD
- MSS_SYS_ENVM_PROGRAM_ERROR
- MSS_SYS_ENVM_VERIFY_ERROR
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER
- MSS_SYS_UNEXPECTED_ERROR

MSS_SYS_puf_get_number_of_keys

Prototype

```
uint8_t MSS_SYS_puf_get_number_of_keys
(
    uint8_t* p_number_of_keys
);
```

Description

The *MSS_SYS_puf_get_number_of_keys()* function is used to retrieve the number of user keys enrolled with the SRAM-PUF service. The number of enrolled keys is also the key identification number to use for the next key to enroll since keys must be enrolled sequentially and the key numbering starts at zero. Up to 58 keys can be enrolled. Keys number 0 and 1 are used for design security and are not available to the user. Key code #2 also has a special function as describe earlier, so it is not recommended to use it for storing secret keys.

Note: This system service is only available on large SmartFusion2 devices starting with the M2S060.

Parameters

p_number_of_user_keys

The *p_user_key* parameter is a pointer to the byte in which the number of keys will be written on successful completion of the system service.

Return Value

The *MSS_SYS_puf_get_number_of_user_keys()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_ENVM_ERROR
- MSS_SYS_PUF_ERROR_WHEN_CREATING
- MSS_SYS_ENVM_PROGRAM_ERROR
- MSS_SYS_INVALID_HASH
- MSS_SYS_INVALID_USER_AC1
- MSS_SYS_ENVM_VERIFY_ERROR
- MSS_SYS_PRIVATE_ENVM_USER_DIGEST_MISMATCH
- MSS_SYS_DRBG_ERROR
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER
- MSS_SYS_UNEXPECTED_ERROR

Example

The following example demonstrates how to read number of keys.

```
uint8_t key_numbers = 0u;
MSS_SYS_puf_get_number_of_keys(&key_numbers);
```

MSS_SYS_puf_enroll_key()

Prototype

```
uint8_t MSS_SYS_puf_enroll_key
(
    uint8_t key_number,
    uint16_t key_size
    uint8_t* p_key_value,
    uint8_t* p_key_location
);
```

Description

The *MSS_SYS_puf_enroll_key()* function is used to enroll keys into the SRAM-PUF hardware block. Keys can be either intrinsic keys or extrinsic keys. An intrinsic key's value is randomly generated by the SRAM-PUF hardware block during key enrollment. An extrinsic key's value is supplied by the user. A key code is created and stored in the System Controller's private eNVM during key enrollment. The key code along with the activation code created as part of the call to *MSS_SYS_puf_create_activation_code()* and the SRAM-PUF start-up value can then be used later to regenerate the enrolled key's value. Enrolled keys can be later reconstructed and retrieved using the *MSS_SYS_puf_fetch_key()* function. The value of the key is protected until it is fetched since its actual value is not stored anywhere in the system.

Note: This system service is only available on large SmartFusion2 devices starting with the M2S060.

Parameters

key_number

The *key_number* parameter specifies the key number that will be used to identify the key in later use after enrollment. The key number will be used to identify which key to retrieve in subsequent calls to function *MSS_SYS_puf_fetch_key()*.

Keys must be enrolled sequentially. Therefore, an arbitrary value cannot be used for *key_number* when enrolling a new key. The value for *key_number* must be derived from a call to *MSS_SYS_puf_get_number_of_keys()* to find the number of keys currently enrolled and derive the next key number in the sequence.

Note: Valid values for key identification numbers are 2 to 57. Keys number 0 and 1 are used for design security and not available to the user. Key code #2 has a special function as described earlier, and is not recommended for storing secret keys. Thus, typically, user data security keys start with key code #3.

key_size

The *key_size* parameter specifies the size of the key to enroll. The key size is a multiple of 64-bit up to 4096-bit. While it is possible to re-use a key code slot for a key code that has been deleted, the replacement key should be the same length as the original key stored in that slot to prevent memory allocation or overrun violations.

NOTE: Key size 0 means 64 * 64 bit.

p_ key_value

The *p_key_value* parameter is a pointer to the buffer containing the value of the key to enroll. The buffer pointed to by *p_key_value* contains the value of the extrinsic key specified by the user.

Setting this pointer's value to zero specifies that an intrinsic key is to be enrolled. In this case, a random value is generated using the SRAM-PUF as the key value.

p_key_location

The *p_key_location* parameter is a pointer to the buffer where the key will be copied when it is fetched through a call to *MSS_SYS_puf_fetch_key()*.

Return Value

The *MSS_SYS_user_key_code()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_ENVM_ERROR
- MSS_SYS_PUF_ERROR_WHEN_CREATING
- MSS_SYS_INVALID_REQUEST_OR_KC
- MSS_SYS_ENVM_PROGRAM_ERROR
- MSS_SYS_INVALID_HASH
- MSS_SYS_INVALID_USER_AC1
- MSS_SYS_ENVM_VERIFY_ERROR
- MSS_SYS_INCORRECT_KEYSIZE_FOR_RENEWING_A_KC
- MSS_SYS_PRIVATE_ENVM_USER_DIGEST_MISMATCH
- MSS_SYS_USER_KEY_CODE_INVALID_SUBCMD
- MSS_SYS_DRBG_ERROR
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER
- MSS_SYS_UNEXPECTED_ERROR

Example

The following example demonstrates how to enroll a user key.

```
uint8_t status = 0u;
uint8_t key_number = 0u;
uint16_t key_size = 0u;
/*
 * The user_ext_key_addr{} buffer is used to store the extrinsic key value to be
 * enrolled by calling function MSS_SYS_puf_enroll_key ().
 */
uint8_t user_ext_key_addr[MAX_USER_KEY_SIZE] = {0x00};

/*
 * The g_my_user_key[] buffer will be used to store the retrieve user key
 * when it is fetch through a call to MSS_SYS_puf_fetch_key() function.
 */
volatile uint8_t g_my_user_key[256] = {0x00};
uint8_t* p_my_user_key = (uint8_t*)&g_my_user_key;

/* Read the number of keys enrolled. */
status = MSS_SYS_puf_get_number_of_keys(&key_number);
if(MSS_SYS_SUCCESS == status)
{
    if(key_number <= MAX_KEY_NUMBER)
    {
         /* Get the intrinsic/extrinsic key size from UART terminal. */
         key_size = get_key_size();

         if(key_type == 1)
         {
             /* Enroll intrinsic key. */
             status = MSS_SYS_puf_enroll_key(key_number, key_size / 8, 0u,
                         p_my_user_key);
         }
         else
         {
             /* Read the key from UART terminal. */
             get_key(&user_ext_key_addr[0], key_size);

             /* Enroll extrinsic key */
             status = MSS_SYS_puf_enroll_key(key_number, key_size / 8,
                         &user_ext_key_addr[0], p_my_user_key);
         }
     }
}
```

MSS_SYS_puf_delete_key()

Prototype

```
uint8_t MSS_SYS_puf_delete_key
(
    uint8_t key_number
);
```

Description

The *MSS_SYS_puf_delete_key()* function is used to delete a previously enrolled key from the SRAM-PUF.

> Note: This system service is only available on large SmartFusion2 devices starting with the M2S060.

Parameters

key_number

The *key_number* parameter specifies the key number of the key to delete.

Return Value

The *MSS_SYS_user_key_code()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_ENVM_ERROR
- MSS_SYS_PUF_ERROR_WHEN_CREATING
- MSS_SYS_INVALID_REQUEST_OR_KC
- MSS_SYS_ENVM_PROGRAM_ERROR
- MSS_SYS_INVALID_HASH
- MSS_SYS_INVALID_USER_AC1
- MSS_SYS_ENVM_VERIFY_ERROR
- MSS_SYS_INCORRECT_KEYSIZE_FOR_RENEWING_A_KC
- MSS_SYS_PRIVATE_ENVM_USER_DIGEST_MISMATCH
- MSS_SYS_USER_KEY_CODE_INVALID_SUBCMD
- MSS_SYS_DRBG_ERROR
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER
  MSS_SYS_UNEXPECTED_ERROR

MSS_SYS_puf_fetch_key()

Prototype

```
uint8_t MSS_SYS_puf_fetch_key
(
    uint8_t key_number,
    uint8_t ** pp_key
);
```

Description

The *MSS_SYS_puf_fetch_key()* function is used to retrieve a user PUF key from the SRAM-PUF hardware block. The key must have been previously enrolled using the *MSS_SYS_puf_enroll_key()* function. The key to retrieve is identified through the key number used at key enrollment time. The key value is copied into the buffer location specified at key enrollment time. The location of this buffer is returned through function argument *pp_key*.

The key value is reconstructed based on the SRAM-PUF power-on value, the activation code created when the device was commissioned using the *MSS_SYS_puf_create_activation_code()* function and the key code stored in the SRAM-PUF hardware block's private eNVM at key enrollment time. The key value does not exist anywhere in the system until it is retrieved by a call to *MSS_SYS_puf_fetch_key()*. Care must be taken to destroy the key value returned by *MSS_SYS_puf_fetch_key()* once it is not required anymore.

Note: This system services is only available on large SmartFusion2 device starting with the M2S060.

Parameters

key_number

The *key_number* parameter specifies the key number identifying the user key to fetch. The valid range of key numbers is from 2 to 57.

pp_key

The *pp_key* parameter is a pointer to a pointer to the buffer that will contain the user key value on successful completion of the system service. The *pp_key* parameter can be set to zero if your application keeps track of the location of the key specified though the call to function *MSS_SYS_puf_enroll_key()* at key enrollment time.

Return Value

The *MSS_SYS_puf_fetch_key()* function returns one of the following status codes:

- MSS_SYS_PUF_ERROR_WHEN_CREATING
- MSS_SYS_INVALID_KEYNUM_OR_ARGUMENT
- MSS_SYS_INVALID_HASH
- MSS_SYS_PRIVATE_ENVM_USER_DIGEST_MISMATCH
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER
- MSS_SYS_UNEXPECTED_ERROR

Example

The following example demonstrates how to retrieve a user key:

```
uint8_t g_my_user_key_handle;
uint8_t * p_my_user_key;

MSS_SYS_puf_fetch_key(g_my_user_key_handle, &p_my_user_key);
```

MSS_SYS_puf_export_keycodes()

Prototype

```
uint8_t MSS_SYS_puf_export_keycodes
(
    uint8_t * p_keycodes
);
```

Description

The *MSS_SYS_puf_export_keycodes()* function is used to export an encrypted copy of all the key codes used internally by the SRAM-PUF hardware block to reconstruct the enrolled keys. Up to 3894 bytes of data can be exported depending on the number and size of keys enrolled.

Keys cannot be fetched anymore after calling *MSS_SYS_puf_export_keycodes()* until a subsequent call to function *MSS_SYS_puf_import_keycodes()* is made with the exported encrypted key codes.

Calling *MSS_SYS_puf_export_keycodes()* and moving the exported data off chip is similar to removing the enrolled keys from the device. The enrolled keys will only be available again if the exported key codes are imported back into the device.

A SmartFusion2 device will only accept imported key codes generated from the last set of enrolled keys on that specific device. This makes the exported key codes data specific to one unique SmartFuson2 device. Key codes cannot be exported from one SmartFusion2 device and imported into another SmartFusion2 device. The exported key codes data is specific to the unique SmartFusion2 device it was exported from.

Keys cannot be reconstructed from the exported key codes data since the intrinsic secret of the SRAM-PUF is required to reconstruct the keys from the key codes. Furthermore, the key codes data is also encrypted using a random key thus preventing an attacker from deriving useful information from the exported key codes data.

Note: This system services is only available on large SmartFusion2 device starting with the M2S060.

Parameters

p_keycodes

The *p_keycodes* parameter is a pointer to the buffer where the PUF key codes will be exported.

Return Value

The *MSS_SYS_puf_export_keycodes()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_ENVM_ERROR
- MSS_SYS_PUF_ERROR_WHEN_CREATING
- MSS_SYS_INVALID_REQUEST_OR_KC
- MSS_SYS_ENVM_PROGRAM_ERROR
- MSS_SYS_INVALID_HASH
- MSS_SYS_INVALID_USER_AC1
- MSS_SYS_ENVM_VERIFY_ERROR
- MSS_SYS_INCORRECT_KEYSIZE_FOR_RENEWING_A_KC
- MSS_SYS_PRIVATE_ENVM_USER_DIGEST_MISMATCH
- MSS_SYS_USER_KEY_CODE_INVALID_SUBCMD
- MSS_SYS_DRBG_ERROR
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER

- MSS_SYS_UNEXPECTED_ERROR

Example

The following example demonstrates how to export all key codes:

```
#define EXPORTED_KEY_CODES_MAX_SIZE   3894u

uint8_t g_key_codes[EXPORTED_KEY_CODES_MAX_SIZE];

MSS_SYS_puf_export_key_codes(g_key_codes);
```

MSS_SYS_puf_import_keycodes()

Prototype

```
uint8_t MSS_SYS_puf_import_keycodes
(
    uint8_t * p_keycodes
);
```

Description

The *MSS_SYS_puf_import_keycodes()* function is used to import a set of PUF key codes that was previously exported using the *MSS_SYS_puf_export_keycodes()* function. Importing the exported key codes allows the enrolled keys to be regenerated. Enrolled keys cannot be regenerated while the key codes have been exported and not imported back.

Importing back the key codes results in all keys being regenerated and copied to the memory locations specified during key enrollment. The content of the buffer holding the imported key codes is modified by function *MSS_SYS_puf_import_keycodes()* to contain the list of pointers to the locations where the keys have been regenerated. This list starts with the address where key number 2 has been regenerated. The location for keys 0 and 1 are not contained in that list because these keys are used for design security and are not accessible to the user.

Note: This system services is only available on large SmartFusion2 device starting with the M2S060.

Parameters

p_keycodes

The *p_keycodes* parameter is a pointer to the buffer containing the PUF key codes to import.

Return Value

The *MSS_SYS_puf_import_keycodes()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_ENVM_ERROR
- MSS_SYS_PUF_ERROR_WHEN_CREATING
- MSS_SYS_INVALID_REQUEST_OR_KC
- MSS_SYS_ENVM_PROGRAM_ERROR
- MSS_SYS_INVALID_HASH
- MSS_SYS_INVALID_USER_AC1
- MSS_SYS_ENVM_VERIFY_ERROR
- MSS_SYS_INCORRECT_KEYSIZE_FOR_RENEWING_A_KC
- MSS_SYS_PRIVATE_ENVM_USER_DIGEST_MISMATCH
- MSS_SYS_USER_KEY_CODE_INVALID_SUBCMD
- MSS_SYS_DRBG_ERROR
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER
- MSS_SYS_UNEXPECTED_ERROR

Example

The following example demonstrates how to import key codes:

```
#define EXPORTED_KEY_CODES_MAX_SIZE   3894u

uint8_t g_key_codes[EXPORTED_KEY_CODES_MAX_SIZE];
```

```
retrieve_exported_key_codes(g_key_codes);
MSS_SYS_puf_import_key_codes(g_key_codes);
```

MSS_SYS_puf_fetch_ ecc_public_key

Prototype

```
uint8_t MSS_SYS_puf_fetch _ecc_public_key
(
    uint8_t* p_puf_public_key
);
```

Description

The *MSS_SYS_puf_fetch_ecc_public_key()* function is used to fetch the PUF ECC public key.

Note: This system services is only available on large SmartFusion2 device starting with the M2S060.

Parameters

p_puf_public_key

The *p_puf_public_key* parameter is a pointer to the buffer where the PUF ECC public key will be stored on successful completion of system service.

Return Value

The *MSS_SYS_puf_fetch_ecc_public_key()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_ENVM_ERROR
- MSS_SYS_NO_VALID_PUBLIC_KEY
- MSS_SYS_PRIVATE_ENVM_USER_DIGEST_MISMATCH
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER
- MSS_SYS_UNEXPECTED_ERROR

Example

The following example demonstrates how to read puf ecc public key.

```
uint8_t puf_public_key[96];
MSS_SYS_puf_fetch_ ecc_public_key(&puf_public_key[0]);
```

MSS_SYS_puf_get_random_seed

Prototype

```
uint8_t MSS_SYS_puf_get_random_seed
(
    uint8_t* p_puf_seed
);
```

Description

The *MSS_SYS_puf_get_random_seed()* function is used to generate a 256-bit true random number seed using the SmartFusion2 SRAM-PUF.

Note: This system service is only available on large SmartFusion2 devices starting with the M2S060.

Parameters

p_puf_seed

The *p_puf_seed* parameter is a pointer to the buffer where the PUF seed will be stored on successful completion of the system service.

Return Value

The *MSS_SYS_puf_get_random_seed()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_PUF_ERROR_WHEN_CREATING
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER
- MSS_SYS_UNEXPECTED_ERROR.

Example

The following example demonstrates how to generate a PUF random seed.

```
uint8_t puf_seed[32];
MSS_SYS_puf_get_random_seed(&puf_seed[0]);
```

MSS_SYS_ecc_point_multiplication

Prototype

```
uint8_t MSS_SYS_ecc_point_multiplication
(
    uint8_t* p_scalar_d,
    uint8_t* p_point_p,
    uint8_t* p_point_q
);
```

Description

The *MSS_SYS_ecc_point_multiplication()* function provides access to the SmartFusion2 System Controller's Elliptic Curve Cryptography (ECC) point multiplication system service. The *MSS_SYS_ecc_point_multiplication()* function computes the point multiplication of a point P on the NIST-384 curve by a scalar value d. The point multiplication results in point Q as follows:

$$d * P = Q$$

The points are defined by their x and y coordinates as follows:

$$P = (x_1, y_1)$$

$$Q = (x_2, y_2)$$

Each x and y coordinate and the scalar d are 384-bit long big-endian numbers.

Note: The point at infinity is specified arbitrarily using x =0, y = 0. (Thus, the point at x=0, y=0, which is not a point on the curve, cannot be represented).

Note: This system service is only available on large SmartFusion2 devices starting with the M2S060.

Parameters

p_scalar_d

The *p_scalar_d* parameter is a pointer to a buffer containing the 384-bit scalar d. The scalar d is 384-bit long meaning that *p_scalar_d* should point to a 48 bytes buffer.

p_point_p

The *p_point_p* parameter is a pointer to a buffer containing the (x_1, y_1) coordinates of input point P. Each x and y coordinate is 384-bit long meaning that *p_point_p* should point to a 96 bytes buffer.

p_point_q

The *p_point_q* parameter is a pointer to a buffer where the (x_2, y_2) coordinates of result Q will be stored. Each x and y coordinate is 384-bit long meaning that *p_point_q* should point to a 96 bytes buffer.

Return Value

The *MSS_SYS_ecc_point_multiplication()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER
- MSS_SYS_UNEXPECTED_ERROR

Example

The following example demonstrates how to use point multiplication.

```
uint8_t d[48];
uint8_t p[96];
uint8_t q[96];
```

```
get_point_d_value(d);
get_point_p_value(p);
MSS_SYS_ecc_point_multiplication(&d[0], &p[0], &q[0]);
display_result_point_q(q)
```

MSS_SYS_ecc_point_addition

Prototype

```
uint8_t MSS_SYS_ecc_point_addition
(
    uint8_t* p_point_p,
    uint8_t* p_point_q,
    uint8_t* p_point_r
);
```

Description

The *MSS_SYS_ecc_point_addition()* function provides access to the SmartFusion2 System Controller's Elliptic Curve Cryptography (ECC) point addition system service. The *MSS_SYS_ecc_point_addition()* function computes the addition of two points, P and Q, on the NIST P-384 curve. The point addition results in point R as follows:

P + Q = R

The points are defined by their x and y coordinates as follows:

P = (x_1, y_1)

Q = (x_2, y_2)

R = (x_3, y_3)

Each x and y coordinate are 384-bit long big-endian numbers.

Note: The point at infinity is specified arbitrarily using x =0, y = 0. (Thus, the point at x=0, y=0, which is not a point on the curve, cannot be represented).

Note: This system service is only available on large SmartFusion2 devices starting with the M2S060.

Parameters

p_point_p

The *p_point_p* parameter is a pointer to a buffer containing the (x_1, y_1) coordinates of input point P. Each x and y coordinate is 384-bit long meaning that *p_point_p* should point to a 96 bytes buffer.

p_point_q

The *p_point_q* parameter is a pointer to a buffer containing the (x_2, y_2) coordinates of input point Q. Each x and y coordinate is 384-bit long meaning that *p_point_q* should point to a 96 bytes buffer.

p_point_r

The *p_point_r* parameter is a pointer to a buffer where the (x_3, y_3) coordinates of result R will be stored. Each x and y coordinate is 384-bit long meaning that *p_point_r* should point to a 96 bytes buffer.

Return Value

The *MSS_SYS_ecc_point_addition()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_MEM_ACCESS_ERROR
- MSS_SYS_SERVICE_NOT_LICENSED
- MSS_SYS_SERVICE_DISABLED_BY_FACTORY
- MSS_SYS_SERVICE_DISABLED_BY_USER
- MSS_SYS_UNEXPECTED_ERROR

Example

The following example demonstrates how to use point addition.

```
uint8_t p[96];
uint8_t q[96];
uint8_t r[96];
```

```
get_point_p_value(p);
get_point_q_value(q);
MSS_SYS_ecc_point_addition(&p[0], &q[0], &r[0]);
display_result_point_r(r);
```

MSS_SYS_ecc_get_base_point

Prototype

```
void MSS_SYS_ecc_get_base_point
(
    uint8_t* p_point_g
);
```

Description

The *MSS_SYS_ecc_get_base_point()* function provides the value of the base point G for NIST elliptic curve P-384. The value of the base point can then be used to generate a public key by multiplying the base point G with a private key using function *MSS_SYS_ecc_point_multiplication()*.

Note: This system service is only available on large SmartFusion2 devices starting with the M2S060.

Parameters

p_point_g

The *p_point_g* parameter is a pointer to a buffer where the coordinate of base point G (x_G, y_G) for curve P-384 will be stored. Each x and y coordinate is 384-bit long meaning that *p_point_p* should point to a 96 bytes buffer.

Return Value

This function does not return a value.

Example

The following example demonstrates how to use of the base point to generate an ECC public key from a private key.

```
uint8_t base_point[96];
uint8_t private_key[48];
uint8_t public_key[96];

MSS_SYS_ecc_get_base_point(&base_point[0]);
MSS_SYS_ecc_point_multiplication(&private_key[0], &base_point[0], &public_key[0]);
```

MSS_SYS_start_clock_monitor

Prototype

```
uint8_t MSS_SYS_start_clock_monitor
(
    void
);
```

Description

The *MSS_SYS_start_clock_monitor()* function enables clock monitoring based on user configuration. The system controller will start monitoring the 50MHz clock timing reference for four 1MHz clock cycle. The expected count is 200 +/- tolerance. If the clock fall outside the tolerance limits, interrupt will be generated and system controller will send clock monitor tamper detect event through COMBLK. MSS should inform to the application about clock monitoring tamper detect event though call back function.

Note: Do not enable both start clock monitoring and stop clock monitoring at the same time.

Note: This system service is only available on large SmartFusion2 devices starting with the M2S060.

Parameters

This function does not take any parameters.

Return Value

The *MSS_SYS_start_clock_monitor()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_UNEXPECTED_ERROR

Example

The following example demonstrates how to start the clock monitoring service.

```
  status = MSS_SYS_start_clock_monitor();
```

MSS_SYS_stop_clock_monitor

Prototype

```
uint8_t MSS_SYS_stop_clock_monitor
(
    void
);
```

Description

The *MSS_SYS_stop_clock_monitor()* function stops the clock monitoring. The system controller will disable 50 MHZ timing clock monitoring and clock monitor tamper detect interrupt.

Note: Do not enable both start clock monitoring and stop clock monitoring at the same time.

Note: This system service is only available on large SmartFusion2 devices starting with the M2S060.

Parameters

This function does not take any parameter.

Return Value

The *MSS_SYS_stop_clock_monitor()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_UNEXPECTED_ERROR

Example

The following example demonstrates how to stop the clock monitoring service.

```
  status = MSS_SYS_stop_clock_monitor();
```

MSS_SYS_enable_puf_power_down

Prototype

```
uint8_t MSS_SYS_enable_puf_power_down
(
    void
);
```

Description

The *MSS_SYS_enable_puf_power_down()* function is used to instruct the system controller to power down the PUF after each of the following operations:

- key fetch
- key creation
- key import
- random seed generation

Powering down the PUF after these operations is the default PUF behavior.

Note: This system service is only available on large SmartFusion2 devices starting with the M2S060.

Parameters

This function does not take any parameter.

Return Value

The *MSS_SYS_enable_puf_power_down()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_UNEXPECTED_ERROR

Example

The following example demonstrates how to turn OFF PUF power.

```
  status = MSS_SYS_enable_puf_power_down();
```

MSS_SYS_disable_puf_power_down

Prototype

```
uint8_t MSS_SYS_disable_puf_power_down
(
    void
);
```

Description

The *MSS_SYS_disable_puf_power_down()* function is used to retain PUF power after the following operations:

- key fetch
- key creation
- key import
- random seed generation

Note: This system service is only available on large SmartFusion2 devices starting with the M2S060.

Parameters

This function does not take any parameter.

Return Value

The *MSS_SYS_disable_puf_power_down()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_UNEXPECTED_ERROR

Example

The following example demonstrates how to retain PUF power.

```
  status = MSS_SYS_disable_puf_power_down();
```

MSS_SYS_clear_lock_parity

Prototype

```
uint8_t MSS_SYS_clear_lock_parity
(
    void
);
```

Description

The *MSS_SYS_clear_lock_parity()* function clears the lock parity tamper flag. The lock parity tamper is cleared by this function provided the hardware conditions that raised this tamper flag are no longer present. The system controller will generate this tamper on detection of parity error on internal lock data.

Note: This system services is only available on large SmartFusion2 device starting with the M2S060.

Parameters

This function does not take any parameter.

Return Value

The *MSS_SYS_clear_lock_parity()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_UNEXPECTED_ERROR

Example

The following example demonstrates how to clear lock parity tamper flag.

```
  status = MSS_SYS_clear_lock_parity();
```

MSS_SYS_clear_mesh_short

Prototype

```
uint8_t MSS_SYS_clear_mesh_short
(
    void
);
```

Description

The *MSS_SYS_clear_mesh_short()* function clears the previously mesh short tamper flag. The system controller monitors the security flash array for invasive tampering using an active mesh. This mesh is driven by a linear feedback shift register (LFSR) and the return value is monitored for discrepancies against the expected value generated by the LFSR. When a tamper event is detected the MESHSHORT tamper flag is set, generating a tamper interrupt. To clear mesh short tamper flag, use tamper control service with bit 5 of Options set to 1. The MESHSHORT tamper flag is cleared provided the hardware conditions are no longer present to raise this flag.

Note: This system service is only available on large SmartFusion2 devices starting with the M2S060.

Parameters

This function does not take any parameter.

Return Value

The *MSS_SYS_clear_mesh_short()* function returns one of the following status codes:

- MSS_SYS_SUCCESS
- MSS_SYS_UNEXPECTED_ERROR

Example

The following example demonstrates how to clear mesh short tamper flag.

```
  status = MSS_SYS_clear_mesh_short();
```

Product Support

Microsemi SoC Products Group backs its products with various support services, including Customer Service, Customer Technical Support Center, a website, electronic mail, and worldwide sales offices. This appendix contains information about contacting Microsemi SoC Products Group and using these support services.

Customer Service

Contact Customer Service for non-technical product support, such as product pricing, product upgrades, update information, order status, and authorization.

From North America, call **800.262.1060**
From the rest of the world, call **650.318.4460**
Fax, from anywhere in the world **408.643.6913**

Customer Technical Support Center

Microsemi SoC Products Group staffs its Customer Technical Support Center with highly skilled engineers who can help answer your hardware, software, and design questions about Microsemi SoC Products. The Customer Technical Support Center spends a great deal of time creating application notes, answers to common design cycle questions, documentation of known issues and various FAQs. So, before you contact us, please visit our online resources. It is very likely we have already answered your questions.

Technical Support

Visit the Microsemi SoC Products Group Customer Support website for more information and support (http://www.microsemi.com/soc/support/search/default.aspx). Many answers available on the searchable web resource include diagrams, illustrations, and links to other resources on website.

Website

You can browse a variety of technical and non-technical information on the Microsemi SoC Products Group home page, at http://www.microsemi.com/soc/.

Contacting the Customer Technical Support Center

Highly skilled engineers staff the Technical Support Center. The Technical Support Center can be contacted by email or through the Microsemi SoC Products Group website.

Email

You can communicate your technical questions to our email address and receive answers back by email, fax, or phone. Also, if you have design problems, you can email your design files to receive assistance. We constantly monitor the email account throughout the day. When sending your request to us, please be sure to include your full name, company name, and your contact information for efficient processing of your request.

The technical support email address is soc_tech@microsemi.com.

My Cases

Microsemi SoC Products Group customers may submit and track technical cases online by going to My Cases.

Outside the U.S.

Customers needing assistance outside the US time zones can either contact technical support via email (soc_tech@microsemi.com) or contact a local sales office. Sales office listings can be found at www.microsemi.com/soc/company/contact/default.aspx.

ITAR Technical Support

For technical support on RH and RT FPGAs that are regulated by International Traffic in Arms Regulations (ITAR), contact us via soc_tech_itar@microsemi.com. Alternatively, within My Cases, select **Yes** in the ITAR drop-down list. For a complete list of ITAR-regulated Microsemi FPGAs, visit the ITAR web page.

Microsemi Corporate Headquarters
One Enterprise, Aliso Viejo CA 92656 USA
Within the USA: +1 (949) 380-6100
Sales: +1 (949) 380-6136
Fax: +1 (949) 215-4996