



MEC2016 Rom B0 API	
Usage Manual	
Rev 0.8	8 August 2017

MICROCHIP CONFIDENTIAL

Copyright © 2013 Microchip or its subsidiaries. All rights reserved.

The information contained herein is confidential and proprietary to Microchip, shall be used solely in accordance with the agreement pursuant to which it is provided, and shall not be reproduced or disclosed to others without the prior written consent of Microchip. Although the information is believed to be accurate, no responsibility is assumed for inaccuracies. Microchip reserves the right to make changes to this document and to specifications and product descriptions at any time without notice. Neither the provision of this information nor the sale of the described semiconductor devices conveys any licenses under any patent rights or other intellectual property rights of Microchip or others. The product may contain design defects or errors known as anomalies, including but not necessarily limited to any which may be identified in this document, which may cause the product to deviate from published specifications. Microchip products are not designed, intended, authorized or warranted for use in any life support or other application where product failure could cause or contribute to personal injury or severe property damage. Any and all such uses without prior written approval of an officer of Microchip will be fully at the risk of the customer.

MICROCHIP DISCLAIMS AND EXCLUDES ANY AND ALL WARRANTIES, INCLUDING WITHOUT LIMITATION ANY AND ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND AGAINST INFRINGEMENT AND THE LIKE, AND ANY AND ALL WARRANTIES ARISING FROM ANY COURSE OF DEALING OR USAGE OF TRADE. IN NO EVENT SHALL SMSC BE LIABLE FOR ANY DIRECT, INCIDENTAL, INDIRECT, SPECIAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES; OR FOR LOST DATA, PROFITS, SAVINGS OR REVENUES OF ANY KIND; REGARDLESS OF THE FORM OF ACTION, WHETHER BASED ON CONTRACT; TORT; NEGLIGENCE OF SMSC OR OTHERS; STRICT LIABILITY; BREACH OF WARRANTY; OR OTHERWISE; WHETHER OR NOT ANY REMEDY OF BUYER IS HELD TO HAVE FAILED OF ITS ESSENTIAL PURPOSE, AND WHETHER OR NOT SMSC HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

TABLE OF CONTENTS

1	INTRODUCTION	3
1.1	PURPOSE.....	3
1.2	SCOPE.....	3
1.3	REFERENCES	3
1.4	GLOSSARY OF TERMS AND ACRONYMS.....	3
2	OVERALL DESCRIPTION.....	4
2.1	PRODUCT PERSPECTIVE	4
2.2	OPERATING ENVIRONMENT	4
2.3	DESIGN AND IMPLEMENTATION CONSTRAINTS	4
2.4	USER DOCUMENTATION.....	4
2.5	ASSUMPTIONS AND DEPENDENCIES.....	4
2.6	USING THE MANUAL.....	4
3	EXTERNAL INTERFACE REQUIREMENTS	5
3.1	HARDWARE INTERFACES.....	5
3.2	SOFTWARE INTERFACES	5
4	API USAGE	6
4.1	SHA APIs	6
4.1.1	SHA Direct APIs	6
4.1.2	SHA1 APIs.....	6
4.1.3	SHA256 APIs.....	6
4.1.4	SHA512 APIs.....	7
4.2	RSA APIs	8
4.2.1	Method 1: Using older APIs.....	8
4.2.2	Method 2: Using new APIs	9
4.3	AES.....	10
4.3.1	Method 1: Using older APIs.....	10
4.3.2	Method 2: Using new APIs	10
4.4	RANDOM NUMBER GENERATOR	11
4.4.1	Method 1: Using older APIs.....	11
4.4.2	Method 2: Using new APIs	11
4.5	ECDSA VERIFICATION	11
4.5.1	Method 1: Using older APIs.....	11
4.5.2	Method 2: Using new APIs	12
5	BUILD AND LINK.....	15
6	TIMING ANALYSIS.....	16
6.1	AES.....	16
6.2	ECDSA.....	16
6.3	PKE	17
6.4	SHA.....	17
6.5	RNG	17
7	PKE SLOT USAGE.....	19
8	REVISION HISTORY	20

1 Introduction

1.1 Purpose

This document specifies the usage of MEC2016 Rom API's.

1.2 Scope

This document entails the zero code and steps to use the API's provided in MEC2016 ROM

1.3 References

//depot_pcs/FWEng/projects/MEC2016/docs/rom api/MEC2016 Rom API Manual.docx

1.4 Glossary of Terms and Acronyms

AES – Advanced encryption Standard
RSA - Rivest-Shamir-Adleman cryptosystem
PKE – Public Key Encryption
SHA – Secure Hash Algorithm
RNG – Random Number Generator
SCM – Shared Crypto Memory
CRT – Chinese Remainder Theorem
KCDSA – Korean Elliptic Curve Digital Signature Algorithm
ECDSA – Elliptic Curve Digital Signature Algorithm
EC25519 – Elliptic Curve 25519
EC – Elliptic Curve
SRP – Secure Remote Password

2 Overall Description

2.1 Product Perspective

This document is for MEC2016 users who wish to use the crypto API's provided with the MEC2016 ROM.

2.2 Operating Environment

The zero code provided is OS independent and can only be used with MEC2016.

2.3 Design and Implementation Constraints

Many of the API's require buffers and memory to be specified. The onus of maintaining the proper buffers and memory is on the caller of the API's.

2.4 User Documentation

//depot_pcs/FWEng/projects/MEC2016/docs/rom api/MEC2016 Rom API Manual.docx

2.5 Assumptions and Dependencies

The usefulness of this document is contingent upon the knowledge of the MEC2016 target hardware features and the API's available.

2.6 Using the Manual

This document has usage for both older (APIs that are backward compatible with the previous version A1) and new APIs (version B0). Method 1 has the usage for the APIs in version A1; Method 2 has the usage for the APIs in version B0. It is recommended to use the new API functions.

3 External Interface Requirements

3.1 Hardware Interfaces

MEC2016 EVB / FPGA with proper bit map
Keil µVision Ulink Pro Debugger tool –
MCHP Trace debugger Tool
Dediprog SPI programmer

3.2 Software Interfaces

Keil Compiler IDE-Version:
µVision V5.15

Tool Version Numbers:

Toolchain:	MDK-ARM Standard Cortex-M Version: 5.15.0	
Toolchain Path:	C:\Keil_v5\ARM\ARMCC\Bin	
C Compiler:	Armcc.exe	V5.05 update 2 (build 169)
Assembler:	Armasm.exe	V5.05 update 2 (build 169)
Linker/Locator:	ArmLink.exe	V5.05 update 2 (build 169)
Library Manager:	ArmAr.exe	V5.05 update 2 (build 169)
Hex Converter:	FromElf.exe	V5.05 update 2 (build 169)
CPU DLL:	SARMCM3.DLL	V5.15.0
Dialog DLL:	DCM.DLL	V1.13.2.0
Target DLL:	ULP2CM3.DLL	V2.200.17.0
Dialog DLL:	TCM.DLL	V1.14.5.0

4 API Usage

The following section lists the API's available and its usage.

Note1: All blocks need to be powered ON with corresponding APIs before usage of any of the API for crypto operations.

Note2: Some crypto APIs require free running timer to be active. Ensure proper operation of free running timer before calling any of the crypto APIs.

Note3: Refer to Section 2.6 on details about using this document.

4.1 SHA APIs

4.1.1 SHA Direct APIs

- 1) Power on SHA block with *api_aes_hash_power()*.
- 2) Check if Hash Block is busy, if not busy proceed. If busy, wait until busy status is false (*api_hash_busy()*)
- 3) Use *api_sha_direct_init()* with the required mode (1,256 or 512) and the pointer to the buffer where the digest will be stored.
- 4) Check if Hash Block is busy, if not busy proceed. If busy, wait until busy status is false (*api_hash_busy()*)
- 5) Run *api_sha_direct_update()* with a buffer pointer to message on which digest is to be calculated. The input message must be 32 bytes aligned and the number of 64 byte blocks of data in the input message must be mentioned.
- 6) If start hash block was not specified in the flags in the previous call, call *api_hash_start()* to start the Hash engine.
- 7) Check the done status of the block using *api_hash_is_done_status()* and proceed only if the previous operation has completed.
- 8) Run *api_sha_direct_finalize()* to complete the final operations of the SHA calculation.
- 9) Check the done status of the block using *api_hash_is_done_status()* and proceed only if the previous operation has completed.
- 10) The digest calculated will be in the buffer specified in step 3.

4.1.2 SHA1 APIs

- 1) Power on SHA block with *api_aes_hash_power()*.
- 2) Check if Hash Block is busy, if not busy proceed. If busy, wait until busy status is false (*api_hash_busy()*)
- 3) Run *api_sha1_init()* with the SHA12_CONTEXT_T data structure.
- 4) Run *api_sha1_update()* with a buffer pointer to message on which digest is to be calculated and a pointer to the SHA12_CONTEXT_T data structure initialized in step 4.
- 5) Wait until hash operation is complete (*api_hash_busy()*)
- 6) Run *api_sha1_finalize()*.
- 7) Wait until hash operation is complete (*api_hash_busy()*).
- 8) The digest calculated will be in the buffer specified in step 3 (*sha12_ctx.digest.b* or *sha12_ctx.digest.w*).

4.1.3 SHA256 APIs

- 1) Power on SHA block with *api_aes_hash_power()*.

- 2) Check if Hash Block is busy, if not busy proceed. If busy, wait until busy status is false (*api_hash_busy()*)
- 3) Run *api_sha256_init()* with the SHA12_CONTEXT_T data structure.
- 4) Run *api_sha256_update()* with a buffer pointer to message on which digest is to be calculated and a pointer to the SHA12_CONTEXT_T data structure initialized in step 4.
- 5) Check the done status of the block using *api_hash_is_done_status()* and proceed only if the previous operation has completed.
- 6) Run *api_sha256_finalize()*.
- 7) Check the done status of the block using *api_hash_is_done_status()* and proceed only if the previous operation has completed.
- 8) The digest calculated will be in the buffer specified in step 3 (*sha12_ctx.digest.b* or *sha12_ctx.digest.w*).

4.1.3.1 For Input less than 56 Bytes

- 1) Power on SHA block with *api_aes_hash_power()*.
- 2) Check if Hash Block is busy, if not busy proceed. If busy, wait until busy status is false (*api_hash_busy()*)
- 3) Call *api_sha256_under56()* with a pointer to the output buffer, input buffer, length of the input (in bytes) and the flag value for enabling/disabling interrupts and starting the hash block.
- 4) If start hash block was not specified in the flags in the previous call, call *api_hash_start()* to start the Hash engine.
- 5) Check the done status of the block using *api_hash_is_done_status()* and proceed only if the previous operation has completed.
- 6) The calculated digest will be in the buffer specified in step 3.

4.1.3.2 SHA256 Block APIs

- 1) Power on SHA block with *api_aes_hash_power()*.
- 2) Check if Hash Block is busy, if not busy proceed. If busy, wait until busy status is false (*api_hash_busy()*)
- 3) Call *api_sha256_block_init()* with a pointer to the output buffer.
- 4) Call *api_sha256_block_update()*.
- 5) If start hash block was not specified in the flags in the previous call, call *api_hash_start()* to start the Hash engine.
- 6) Check the done status of the block using *api_hash_is_done_status()* and proceed only if the previous operation has completed.
- 7) Call *api_sha256_block_finalize()*.
- 8) If start hash block was not specified in the flags in the previous call, call *api_hash_start()* to start the Hash engine.
- 9) Check the done status of the block using *api_hash_is_done_status()* and proceed only if the previous operation has completed.
- 10) The calculated digest will be in the output buffer specified in step 3.

4.1.4 SHA512 APIs

- 1) Power on SHA block with *api_aes_hash_power()*.
- 2) Check if Hash Block is busy, if not busy proceed. If busy, wait until busy status is false (*api_hash_busy()*)
- 3) Call *api_sha512_init()* with SHA5_CONTEXT_T data structure.
- 4) Check the done status of the block using *api_hash_is_done_status()* and proceed only if the previous operation has completed.
- 5) Call *api_sha512_update()*.
- 6) Check the done status of the block using *api_hash_is_done_status()* and proceed only if the previous operation has completed.

- 7) Call *api_sha512_finalize()*.
- 8) Check the done status of the block using *api_hash_is_done_status()* and proceed only if the previous operation has completed.
- 9) The calculated digest will be available in the data structure provided in Step 3 (*sha5_ctx.hash.b* or *sha5_ctx.hash.w*).

4.2 RSA APIs

4.2.1 Method 1: Using older APIs

- 1) Power on PKE block with *pke_power()*.
- 2) Call the *rsa_load_key()* to load Public-Private Key pairs into the PKE engine. Specify the RSA bite length (1024, 2048 or 4096) and the byte order of data provided. 4 combinations are possible according to which keys may be loaded. The slot numbers are handled by the api. Keys can be explicitly programmed with *pke_write_scm()* calls.
 - a. RSA Encryption with Public Key
 - i. Pointer to private exponent = Not used
 - ii. Pointer to public modulus = your public key modulus -> Slot 0
 - iii. Pointer to public exponent = your public key exponent -> Slot 8
 - b. RSA Decryption with Private Key
 - i. Pointer to private modulus = your private key modulus -> Slot 6
 - ii. Pointer to public modulus = your public key modulus -> Slot 0
 - iii. Pointer to public exponent = your public key exponent -> Slot 8
 - c. RSA Encryption with Private Key
 - i. Pointer to private exponent = Not used
 - ii. Pointer to public modulus = your public key modulus -> Slot 0
 - iii. Pointer to public exponent = your private key exponent -> Slot 8
 - d. RSA Decryption with Public Key
 - i. Pointer to private exponent = your private exponent -> Slot 6
 - ii. Pointer to public modulus = your public modulus -> Slot 0
 - iii. Pointer to public exponent = Not used
- 3) If data is to be encrypted, call *rsa_encrypt()* with the rsa bit len (1024, 2048 or 4096), pointer to structure having byte length of input data & pointer to input data. Start the PKE engine by calling *pke_start()*.
- 4) Wait for PKE engine to complete operation (*pke_busy()*). Once complete, the encrypted message can be found in slot 5 of crypto memory. The data can be read into a local buffer with *pke_read_scm()*.
- 5) If data is to be decrypted, there are two methods – RSA decryption and CRT RSA decryption

RSA Decryption:

 - a. Call *rsa_decrypt()* with rsa bit len (1024, 2048 or 4096), pointer to structure having byte length of encrypted data & pointer to encrypted data.
 - b. Start the PKE engine by calling *pke_start()*.

CRT RSA Decryption:

 - a. Call *rsa_load crt_params()* api with first exponent, second exponent, coefficient to load these parameters into appropriate slots in the shared crypto memory(scm slots). Alternatively, the parameters can be generated using *rsa crt_gen_params()*. Wait for for PKE engine to complete operation (*pke_busy()*).
 - b. Then call *pke_write_scm()* to load the two prime numbers to scm slots 2 and 3.
 - c. Finally, call *pke_rsa crt_decrypt()* with bit len (1024, 2048 or 4096), input data, byte length of input data.
 - d. Pke engine can be started by specifying the appropriate flag parameter. If not started, the pke engine may be explicitly started with *pke_start()*.
 - e. Perform RSA CRT decryption using *pke_rsa crt_decrypt()*.
 - f. Program the command value corresponding to CRT decryption in the PKE command Register and start the pke engine using *pke_start()*.

- 6) Wait for PKE engine to complete operation (*pke_busy()*). Once complete, the decrypted data will be in slot 5 of Shared Crypto memory. This data can be read into a local buffer with *pke_read_scm()*.
- 7) RSA Signature Generation and Verification:
 - i) Call the *rsa_load_key()* to load Public-Private Key pairs into the PKE engine. Specify the RSA bite length (1024, 2048 or 4096) and the byte order of data provided.
 - ii) Wait for PKE engine to complete operation(*pke_busy()*).
 - iii) Call *rsa_signature_gen()* with bit length, hash digest, specifying the byte order of data provided.
 - iv) Signature for the given hash digest will be generated in scm slot 5.
 - v) After completion of PKE operation(*pke_busy()*), copy expected hash digest to slot C using *pke_write_scm()*.
 - vi) Call *rsa_signature_verify()* with pointer to signature generated in the above process.
 - vii) After completion of PKE operation(*pke_busy()*), read PKE status register by *api_pke_done_status()*. If bit 9 is set in the status register, it indicates that the signature is not valid for the given expected hash digest.
 - viii) The regenerated hash digest calculated by the PKE engine may be read from slot 5.

4.2.2 Method 2: Using new APIs

- 1) Power on PKE block with *api_pke_power()*.
- 2) Load the message into Slot 4 of the SCM using *api_pke_copy_to_scm2()*.
- 3) Load RSA keys into the SCM. Two combinations are possible according to which suitable APIs can be called. The slot numbers are handled by the APIs. Keys can be explicitly programmed with *api_pke_rsa_load_param()* calls.
 - a. RSA Encryption with Public Key and Decryption with Private Key – use *api_pke_rsa_load_key()* to load the private exponent, public modulus and public exponent into their respective slots.
 - b. RSA Encryption with Private Key – use *api_pke_rsa_load_prv_key()* to load the public modulus and the private exponent.
RSA Decryption with Public Key – use *api_pke_rsa_load_pub_key()* to load the public keys.
- 4) Perform RSA encryption with *api_pke_rsa_operation()* by passing rsa bit len (1024, 2048 or 4096), operation to be performed and interrupt status.
Alternatively, *api_pke_rsa_crypt()* can be used.
- 5) Start the PKE engine by calling *api_pke_start()*.
- 6) Wait for PKE engine to complete operation (*api_pke_busy()*). Once complete, the encrypted message can be found in slot 5 of crypto memory. The data can be read into a local buffer with *api_pke_copy_from_scm()*.
- 7) If data is to be decrypted, there are two methods – RSA decryption and CRT RSA decryption
RSA Decryption:
 - a. Load the encrypted message into Slot 4 of the SCM using *api_pke_copy_to_scm2()*.
 - b. Perform RSA decryption with *api_pke_rsa_operation()* by passing rsa bit len (1024, 2048 or 4096), operation to be performed and interrupt status.
Alternatively, *api_pke_rsa_crypt()* can be used.
 - c. Start the PKE engine by calling *api_pke_start()*.
- CRT RSA Decryption:
 - a. Load the private key and CRT parameters into the SCM using *api_pke_rsa_load_prv_key()* and *api_pke_rsa_load_crt_key()* respectively.
 - b. Perform RSA CRT Decryption using *api_pke_rsa_crt_decrypt()* and then use *api_pke_rsa_operation()* to start the operation.
 - c. Program the command value corresponding to CRT decryption in the PKE command Register.
 - d. Start the PKE block using *api_pke_start()*.

- 8) Wait for PKE engine to complete operation (`api_pke_busy()`). Once complete, the decrypted data will be in slot 5 of Shared Crypto memory. This data can be read into a local buffer with `api_pke_copy_from_scm()`.
- 9) RSA Signature Generation and Verification:
 - i. Load RSA keys into the SCM using `api_pke_rsa_load_key()`.
 - ii. Wait for PKE engine to complete operation(`api_pke_busy()`).
 - iii. Load the hash digest into Slot 4 of the SCM using `api_pke_rsa_load_param()` and wait for the operation to complete (`api_pke_busy()`).
 - iv. Call `api_pke_rsa_operation()` to generate the signature and pass rsa bit len (1024, 2048 or 4096), operation to be performed and interrupt status.
 - v. Start the PKE engine using `api_pke_start()`.
 - vi. Signature for the given hash digest will be generated in SCM slot 5. After completion of PKE operation (`pke_busy()`), copy the signature generated and the expected hash digest to slots 4 & C respectively using `api_pke_rsa_load_param()`.
 - vii. Call `api_pke_rsa_operation()` to verify the signature and pass rsa bit len (1024, 2048 or 4096), operation to be performed and interrupt status.
 - viii. Start the PKE engine using `api_pke_start()`.
 - ix. After completion of PKE operation (`pke_busy()`), call `api_pke_rsa_is_signature_valid()` to check if the signature is valid.
 - x. The regenerated hash digest calculated by the PKE engine may be read from slot 5.

4.3 AES

4.3.1 Method 1: Using older APIs

- 1) Power on AES block with `aes_hash_power()`.
- 2) Reset the AES hash block with `aes_hash_reset()`.
- 3) Check if AES block is busy with `aes_busy()`. If it is not busy, continue with step 4.
- 4) Set AES Private key (randomly generated) LSB first, optional initialization vector LSB first and also specify the AES key length used using `api_aes_set_key()`
- 5) Check for AES status with rom API `aes_status()`
 - a. clear status for leftover status bits if any using API `aes_iclr()`
- 6) Call API `aes_crypt()` for encryption or decryption. The message should be an aligned input data buffer, a pointer buffer to load the aligned output data buffer and the mode of operation. Supported modes are ECB, CBC, CTR and OFB.
- 7) Start the AES operation to be performed by calling function `aes_start()`
- 8) Wait for the done status by calling API `aes_done_status()`
- 9) Once done the operated data output will be in the buffer provided via API `aes_crypt()`
- 10) Stop AES block using the API `aes_stop()`
- 11) Put the AES block in sleep state using `aes_hash_power(false);`

4.3.2 Method 2: Using new APIs

- 1) Power on the AES block with `api_aes_hash_power()`.
- 2) Reset the AES block using `api_aes_hash_reset()`.
- 3) Check if the AES block is busy using `api_aes_busy()`. If it is not busy, continue with step 4.
- 4) Set AES Private Key (randomly generated), AES Private Key length and the initialization vector (optional) using `api_aes_set_key()`. The initialization vector and the AES Private Key must be in LSB first format.
Alternatively, AES private key and the initialization vector (optional) can be programmed using `api_aes_prog_key()` and `api_aes_prog_iv()` respectively. The initialization vector and the AES Private Key must be in LSB first format.
- 5) Check for AES status using `api_aes_status()`.

- a. Clear the interrupts using *api_aes_iclr()*.
- 6) Encrypt/decrypt the message using *api_aes_crypt()*. The message and the output need to be stored in an aligned data buffer. Supported modes of operation are: ECB, CBC, CTR and OFB.
- 7) Start the AES operation using *api_aes_start()*.
- 8) Check for the done status using *api_aes_is_done_status2()*.
- 9) Once the operation is done, the output is stored in the data buffer provided to *api_aes_crypt()*.
- 10) Stop the AES block using *api_aes_stop()*.
- 11) Put the AES block in sleep state using *api_aes_hash_power(0)*.

4.4 Random number generator

4.4.1 Method 1: Using older APIs

- 1) Power on RND HW block with API *rng_power(true)*.
- 2) Reset the RND HW block with *rng_reset()*
- 3) Two modes of random number are generated asynchronous/true random mode and Non-zero (pseudo-random mode). Select the mode of operation by calling *rng_mode(mode)*
 - a. 0 – asynchronous
 - b. 1 - pseudo-random mode
- 4) Start the HW block run state by calling function *rng_start()*;
- 5) Wait for operation completion by polling *rng_get_fifo_level()* for data in the internal buffer
 - a. Return 0 for not completion
 - b. Non –zero value for completion of operation – FIFO will have random data
- 6) Once completion internal buffer will have 1Kbits of random data use API
 - a. *rng_get_bytes()* number of random bytes to retrieve. Must be less or equal to the size of the buffer or
 - b. *rng_get_words()* Reads the FIFO level register and return the number of 32-bit words of random data currently in the FIFO – max value supported is 1024 bits.
- 7) Stop the RND HW block with *rng_stop()*.

4.4.2 Method 2: Using new APIs

- 1) Power on the RND block using *api_rng_power()*.
- 2) Reset the RND block using *api_rng_reset()*.
- 3) Set the mode of operation using *api_rng_mode()*. The two modes are asynchronous (0) and pseudo random (1).
- 4) Start the RND block using *api_rng_start()*.
- 5) Poll for data in the internal buffer using *api_rng_get_num_random_words()*.
 - a. 0 – Buffer empty
 - b. Non – zero value indicates the number of 32 bit words currently in the internal buffer. Maximum value is 1 KB.
- 6) Data can be retrieved from the buffer using
 - a. *api_rng_get_random_bytes()* to get the specified number of bytes.
 - b. *api_rng_get_random_words()* to get the specified number of words.
- 7) Stop the RND HW block with *api_rng_stop()*.

4.5 ECDSA Verification

4.5.1 Method 1: Using older APIs

- 1) Generate SHA Digest of the message to be validated. (Optional if digest already exists).
- 2) Check if pke block is busy (*pke_busy()*). If not busy, proceed.
- 3) Load the curve into the SCM using *ec_prog_curve()*.

- 4) Check if pke block is busy (*pke_busy()*). If not busy, proceed.
- 5) Call the *ecdsa_verify()* API with the Public key, The signature of message and the digest calculated.
- 6) Start the PKE Engine by calling *pke_start()*.
- 7) Wait until PKE operations are done by polling on *pke_done_status(PKE_STATUS)*.
- 8) Check the 9th bit of *PKE_STATUS*. If it is reset, Signature is valid, if set, the signature is Invalid.

ECDSA Point Operations:

The procedure for ec point operations like *ec_point_add*, *ec_point_double*, *ec_point_scalar_mult2*, *ec_point_scalar_mult3* is explained below.

- 1) Call *pke_power()* to power on the block.
- 2) For ECDSA point operations, the curve should be programmed to slots using API *ec_prog_curve()*.
- 3) Check PKE engine ready using api *pke_busy()*.
- 4) Set slot numbers to operand pointers A,B,C using *pke_set_operand_slots()* API. Operand pointers A and B correspond to input data to the ec point operations. The output of the operation is pointed by pointer C. *pke_set_operand_slots()* api instructs the PKE engine the slot numbers where it should look for operands A and B, and also where it should store output C. The slot numbers used as default for pointers A,B,C are 6,8,C respectively.
- 5) Call API to appropriate point operation with required parameters, data byte order.
- 6) The output of operation may be read from slot number corresponding to pointer C (slot C if using default values).

Curve 25519 Operations:

ec25519_xrecover:

- 1) Call *pke_power()* to power on the block.
- 2) Call *ec25519_xrecover()* API with y coordinate, size and byte order.
- 3) Call *pke_start()* and wait for operation to complete(*pke_busy()*).
- 4) The recovered x coordinate is always loaded to scm slot 6. Read x coordinate using call to *pke_read_scm()*.

ed25519_scalar_mult:

- 1) Call *pke_power()* to power on the block.
- 2) Call *ed25519_scalar_mult()* API with point on curve 25519, scalar and byte order.
- 3) Call *pke_start()* and wait for operation to complete(*pke_busy()*).
- 4) The x and y coordinates of product are always loaded to scm slots A and B. Read using call to *pke_read_scm()*.

ed25519_valid_sig:

- 1) Call *pke_power()* to power on the block.
- 2) Call *ed25519_valid_sig()* API with structure variable of type *Ed25519_SIG_VERIFY*.
- 3) Call *pke_start()* and wait for operation to complete(*pke_busy()*).
- 4) The parameters P1x,P1y,P2x,P2y,P3x,P3y are loaded to slots A,B,C,D,E,F respectively.
- 5) Verify validity of signature by comparing P1 and P3.

4.5.2 Method 2: Using new APIs

- 1) Generate SHA Digest of the message to be validated. (Optional if digest already exists).
- 2) Check if pke block is busy (*api_pke_busy()*). If not busy, proceed.

- 3) Load the curve parameters using `api_pke_ec_prog_curve()`.
- 4) Check if pke block is busy (`api_pke_busy()`). If not busy, proceed.
- 5) Use `api_pke_ecdsa_verify3` with the Public key, the signature of message and the digest calculated to verify the signature.
- 6) Start the PKE Engine by calling `api_pke_start()`.
- 7) Wait until PKE operations are done by polling on `api_pke_is_done_status2()`.
- 8) Check the 9th bit of `PKE_STATUS`. If it is set, the signature is Invalid; otherwise, the signature is valid.

ECDSA Point Operations:

The procedure for ec point operations like `ec_point_add`, `ec_point_double`, `ec_point_scalar_mult2`, `ec_point_scalar_mult3` is explained below.

- 1) Call `api_pke_power()` to power on the block.
- 2) For ECDSA point operations, the curve should be programmed to slots using API `api_pke_ec_prog_curve()`.
- 3) Check PKE engine ready using `api_pke_busy()`.
- 4) Set slot numbers to operand pointers A,B,C `api_pke_set_operand_slots()`. Operand pointers A and B correspond to input data to the ec point operations. The output of the operation is pointed by pointer C. `api_pke_set_operand_slots` instructs the PKE engine the slot numbers where it should look for operands A and B, and also where it should store output C. The slot numbers used as default for pointers A,B,C are 6,8,C respectively.
- 5) Call API to appropriate point operation with required parameters and the appropriate data byte order.
- 6) The output of operation may be read from slot number corresponding to pointer C (slot C if using default values) using `api_pke_copy_from_scm()`.

Curve 25519 Operations:

`api_pke_ed25519_xrecover:`

- 1) Call `api_pke_power()` to power on the block.
- 2) Call `api_pke_ed25519_xrecov()` with a pointer to the y coordinate, size of the y coordinate and byte order.
- 3) Call `api_pke_start()` and wait for operation to complete(`api_pke_busy()`).
- 4) The recovered x coordinate is always loaded to SCM slot 6. Read x coordinate using call to `api_pke_copy_from_scm()`.

`api_pke_ed25519_scalar_mult:`

- 1) Call `api_pke_power()` to power on the block.
- 2) Call `api_pke_ed25519_scalarmult()` API with points on curve 25519, scalar and byte order.
- 3) Call `api_pke_start()` and wait for operation to complete(`api_pke_busy()`).
- 4) The x and y coordinates of product are always loaded to SCM slots A and B. Read using call to `api_pke_copy_from_scm()`.

`api_pke_ed25519_valid_sig:`

- 1) Call `api_pke_power()` to power on the block.
- 2) Call `api_pke_ed25519_valid_sig()` API with structure variable of type `Ed25519_SIG_VERIFY`.
- 3) Call `api_pke_start()` and wait for operation to complete(`api_pke_busy()`).
- 4) The parameters P1x,P1y,P2x,P2y,P3x,P3y are loaded to slots A,B,C,D,E,F respectively.
- 5) Verify validity of signature by comparing P1 and P3.

api_pke_ec25519_ptmult:

- 1) Call `api_pke_power()` to power on the block.
- 2) Call `api_pke_ec25519_ptmult()` API with points the x coordinate, scalar and byte order.
- 3) Call `api_pke_start()` and wait for operation to complete(`api_pke_busy()`).
- 4) The x coordinate of product is loaded to SCM slot C. Read using call to `api_pke_copy_from_scm()`.

SRP Operation:

- 1) Call `api_pke_power()` to power on the block.
- 2) Load SRP parameters into the Crypto Memory by calling `api_pke_srp_sc()`.
- 3) Call `api_pke_start()` to start the operation and wait for operation to complete(`api_pke_busy()`).
- 4) The output can be read from Slot B using `api_pke_copy_from_scm()`.

5 Build and link

Use the provided symdef file and API header file for proper linking of the application code with bootrom.

If running on FPGA need to download the bootcode for proper linking of the object binary on runtime for the API calls.

Use linker script for loading the bootcode binary using load incremental option.

6 Timing Analysis

For all the crypto operations mentioned in section 4 timing measurement is done at CPU clock of 48MHz and the results are added below

6.1 AES

BLOCK	AES MESSAGE LENGTH = 2048 bytes KEY LENGTH = 256 bits									
	ECB		CBC		CTR		CFB		OFB	
BLOCK CONFIGURATION										
BLOCK OPERATION	ENCRYPTION	DECRYPTION	ENCRYPTION	DECRYPTION	ENCRYPTION	DECRYPTION	ENCRYPTION	DECRYPTION	ENCRYPTION	DECRYPTION
TIME (usec) = (1/CPU_CLK) * (CPU	146.55	215.64	146.61	215.61	143.85	215.19	143.88	215.16	143.85	214.05
TOTAL TIME (usec)	362.19		362.22		359.04		359.04		357.1	

6.2 ECDSA

BLOCK	ECDSA MESSAGE LENGTH = 2048 bytes PUBLIC KEY LENGTH = 64 bytes SIGNATURE LENGTH = 64 bytes
	SHA256
BLOCK CONFIGURATION	
BLOCK OPERATION	
TIME (usec) = (1/CPU_CLK) * (CPU CYCLES)	9280
TOTAL TIME (usec)	9280

6.3 PKE

BLOCK	PKE RSA BIT LENGTH = 1024 BITS INPUT MSG LENGTH = 2048 BYTES	
	RSA ENCRYPTION WITH PUBLIC KEY DECRYPTION WITH PRIVATE KEY	RSA ENCRYPTION WITH PRIVATE KEY DECRYPTION WITH PUBLIC KEY
BLOCK CONFIGURATION		
BLOCK OPERATION TIME (usec) = (1/CPU_CLK) * (CPU CYCLES)		
	156346	157719
TOTAL TIME (usec)	156346	157719

6.4 SHA

BLOCK	SHA INPUT MSG LENGTH = 2048 BYTES				
	BLOCK CONFIGURATION	SHA1	SHA12 (MODE 256)	SHA35 (MODE 512)	
BLOCK OPERATION					
TIME (usec) = (1/CPU_CLK) * (CPU CYCLES)					
		477.24	478.74	425.55	
TOTAL TIME (usec)		477.24	478.74	425.55	

6.5 RNG

BLOCK	RNG	
	BLOCK CONFIGURATION	TRUE PSEUDO

	RANDOM NUMBER	RANDOM NUMBER
BLOCK OPERATION		
CPU CYCLES	3285	6407
TIME (usec) = (1/CPU_CLK) * (CPU CYCLES)	68.43	133.47
TOTAL TIME (usec)	68.43	133.47

7 PKE slot usage

MEC2016 has dedicated crypto SRAM (shared crypto memory - SCM) for PKE block usage. This memory is shared by various PKE operations and hence it limits the operations which can be carried out in parallel. Some of the operations supported by PKE are

- Primitive arithmetic operation
- RSA Cryptosystem
- Curve25519
- ECDSA

The SCM is used to program parameters & keys and to upload/download operands/results from the host side. The shared crypto memory of MEC2016 is divided into 31 slots (slot0-slot30) each of 512 bytes

Below table lists the usage of the slots for various operations

PKE operation	Slots used
RSA Encryption with Public Key	Slot 8 - public exponent Slot 0 – public modulus Slot 5 – Encryption output
RSA Encryption with Private Key	Slot 8 – private exponent Slot 0 – public modulus Slot 5 – Encryption output
RSA Decryption with Private key	Slot 0 – public modulus Slot 6 – private exponent Slot 8 – public exponent Slot 5 – Decryption output
EC program curve	Slot 0 – prime Slot 1 – order Slot 2 - generator point x-coordinate Slot 3 - generator point y-coordinate Slot 4 – curve parameter a Slot 5 – curve parameter b
EC25519 recover x coordinate	Slot 6 – result
PKE clear slot	Specified at run time
PKE read slot	Specified at run time
PKE write slot	Specified at run time
EC Modular arithmetic	Specified at run time

Since above operations share multiple slots these PKE crypto operations cannot be run in parallel

8 Revision History

Name	Revision Level	Date	Section	Remarks
Hemal Gujarathi – I15581	0.1	21 Apr 2016	4.4	Created a new copy to include changes for Bootrom A1 version
Akshaya Karthikeyan - I17306	0.2	13 May 2016	4.4	Updated usage information for rsa crt decryption.
Akshaya Karthikeyan - I17306	0.3	10 Aug 2016	4.7	Updated usage information for ecdsa check point operations and rsa signature generation.
Swastik Pramanik	0.4	6 April 2017	4.6	Updated the RNG usage
Manisha Mishra C19242	0.5	26 June 2017	2.6, 4, 4.5,4.6	Added Section 2.6 to include details about using the manual. Added Note3 in Section 4. Updated Section 4.5,4.6 to include usage information for BootROM B0 version.
Manisha Mishra C19242	0.6	30 June 2017	4.4	Updated Section 4.4 to include usage information for BootROM B0 version.
Manisha Mishra C19242	0.7	18 July 2017	1.4 4 4.2 4.5	Updated Section 1.4 Regrouped Section 4. Included usage for CRT decryption Updated SHA APIs for BootROM B0 version. Updated Section 4.5 to include usage information for BootROM B0 version.
Manisha Mishra C19242	0.8	8 August 2017	4.5.2	Added usage for api_pke_srp_sc() and api_pke_ec25519_ptmult()