­­­

|  |  |
| --- | --- |
| MEC2016 Rom A1 API | |
| Usage Manual | |
| Rev 0.4 | 6 April 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](#_Toc458591136)

[1.1 Purpose 3](#_Toc458591137)

[1.2 Scope 3](#_Toc458591138)

[1.3 References 3](#_Toc458591139)

[1.4 Glossary of Terms and Acronyms 3](#_Toc458591140)

[2 Overall Description 4](#_Toc458591141)

[2.1 Product Perspective 4](#_Toc458591142)

[2.2 Operating Environment 4](#_Toc458591143)

[2.3 Design and Implementation Constraints 4](#_Toc458591144)

[2.4 User Documentation 4](#_Toc458591145)

[2.5 Assumptions and Dependencies 4](#_Toc458591146)

[3 External Interface Requirements 5](#_Toc458591147)

[3.1 Hardware Interfaces 5](#_Toc458591148)

[3.2 Software Interfaces 5](#_Toc458591149)

[4 API Usage 6](#_Toc458591150)

[4.1 SHA API’s 6](#_Toc458591151)

[4.2 SHA12 API’s 6](#_Toc458591152)

[4.3 SHA35 API’s 6](#_Toc458591153)

[4.4 RSA API’s 7](#_Toc458591154)

[4.5 AES 8](#_Toc458591155)

[4.6 Random number generator 8](#_Toc458591156)

[4.7 ECDSA Verification 8](#_Toc458591157)

[5 Build and link 10](#_Toc458591158)

[6 Timing Analysis 11](#_Toc458591159)

[6.1 AES 11](#_Toc458591160)

[6.2 ECDSA 11](#_Toc458591161)

[6.3 PKE 12](#_Toc458591162)

[6.4 SHA 12](#_Toc458591163)

[6.5 RNG 12](#_Toc458591164)

[7 PKE slot usage 14](#_Toc458591165)

[8 Revision History 15](#_Toc458591166)

# Introduction

## Purpose

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

## Scope

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

## References

//depot\_pcs/FWEng/projects/MEC2016/docs/rom api/MEC2016 Rom API Manual.docx

## Glossary of Terms and Acronyms

# Overall Description

## Product Perspective

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

## Operating Environment

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

## 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.

## User Documentation

//depot\_pcs/FWEng/projects/MEC2016/docs/rom api/MEC2016 Rom API Manual.docx

## Assumptions and Dependencies

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

# External Interface Requirements

## Hardware Interfaces

MEC2016 EVB / FPGA with proper bit map

Keil µVision Ulink Pro Debugger tool –

MCHP Trace debugger Tool

Dediprog SPI programmer

## 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

# 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.

## SHA API’s

1. Power on SHA block with *aes\_hash\_power ().*
2. Check if Hash Block is busy, if not busy proceed. If busy, wait until busy status is false (*hash\_busy()*)
3. Run *sha\_init* with the required mode (1,256 or 512) and the pointer to the buffer where the digest will be stored. The buffer must be 4 byte aligned for SHA1 and SHA256 and 8 byte aligned for SHA512.
4. Run *sha\_update* with a buffer pointer to message on which digest is to be calculated. The message must align with the block size requirement for the mode being calculated (64 byte block for SHA1 and SHA256, 128 byte block for SHA512). The number of blocks of data provided must be mentioned.
5. If start hash block was not specified in the flags in the previous call, call *hash\_start*() to start the Hash engine.
6. Wait until hash operation is complete (*hash\_busy()*)
7. Run *sha\_final* with a buffer of size at least one block and message length.
8. Wait until hash operation is complete (*hash\_busy()*).
9. The digest calculated will be in the buffer specified in step 2.

## SHA12 API’s

1. Power on SHA block with *aes\_hash\_power ().*
2. Check if Hash Block is busy, if not busy proceed. If busy, wait until busy status is false (*hash\_busy())*
3. Call *sha12\_init* with the SHA12\_CONTEXT\_T data structure and the mode required. SHA12 API’s only support SHA1 and SHA256.
4. Call *sha12\_update* with the data structure, the message for which digest is to be calculated and the length of the message. The input need not adhere to block size requirement.
5. Wait on hash busy status (*hash\_busy()*). Once hash block is free, proceed.
6. Call *sha12\_finalize* with the data structure provided previously.
7. Wait for Hash operation to complete (*hash\_busy()*).
8. The calculated digest will be in the data structure provided (sha12\_ctx.hash.b or sha12\_ctx.hash.w).

## SHA35 API’s

1. Power on SHA block with *aes\_hash\_power ().*
2. Check if Hash Block is busy, if not busy proceed. If busy, wait until busy status is false (*hash\_busy())*
3. Call *sha35\_init* with SHA35\_CONTEXT\_T data structure and the mode required. Sha35 API’s only support SHA512.
4. Call *sha35\_update* with the data structure, the message on which hash is to be calculated and the length of the message. The input need not adhere to block size requirement.
5. Wait until hash block is not busy (*hash\_busy())*.
6. Call *sha35\_finalize* with the data structure provided previously.
7. Wait until Hash operation is complete (*hash\_busy())*.
8. The calculated digest will be available in the provided data structure (sha35\_ctx.hash.b or sha35\_ctx.hash.w).

## RSA API’s

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.
   1. RSA Encryption with Public Key
      1. Pointer to private exponent = Not used
      2. Pointer to public modulus = your public key modulus -> Slot 0
      3. Pointer to public exponent = your public key exponent -> Slot 8
   2. RSA Decryption with Private Key
      1. Pointer to private modulus = your private key modulus -> Slot 6
      2. Pointer to public modulus = your public key modulus -> Slot 0
      3. Pointer to public exponent = your public key exponent -> Slot 8
   3. RSA Encryption with Private Key
      1. Pointer to private exponent = Not used
      2. Pointer to public modulus = your public key modulus -> Slot 0
      3. Pointer to public exponent = your private key exponent -> Slot 8
   4. RSA Decryption with Public Key
      1. Pointer to private exponent = your private exponent -> Slot 6
      2. Pointer to public modulus = your public modulus -> Slot 0
      3. 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:
   1. call rsa\_decrypt() with rsa bit len (1024, 2048 or 4096), pointer to structure having byte length of encrypted data & pointer to encrypted data. Start the PKE engine by calling *pke\_start()*.

CRT RSA Decryption:

* 1. 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). Folllowing this, call pke\_write\_scm api to load the two prime numbers to scm slots 2 and 3. Finally call pke\_rsa\_crt\_decrypt with bit len (1024, 2048 or 4096), input data, byte length of input data. Pke engine can be started by specifying the appropriate flag parameter. If not started, the pke engine may be explicitly started with *pke\_start().*
  2. call rsa\_crt\_gen\_params() api with first prime number, second prime number and private exponent. Wait for for PKE engine to complete operation (*pke\_busy()*). Call pke\_rsa\_crt\_decrypt with bit len (1024, 2048 or 4096), input data, byte length of input data.The pke engine is to be started with *pke\_start().*

1. 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()*.
2. RSA Signature Generation and Verification:
   * 1. 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.
     2. Wait for PKE engine to complete operation(pke\_busy()).
     3. Call rsa\_signature\_gen() with bit length, hash digest, specifying the byte order of data provided.
     4. Signature for the given hash digest will be generated in scm slot 5.
     5. After completion of PKE operation(pke\_busy()), copy expected hash digest to slot C using pke\_write\_scm().
     6. Call rsa\_signature verify() with pointer to signature generated in the above process.
     7. 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.
     8. The regenerated hash digest calculated by the PKE engine may be read from slot 5.

## AES

1. Power on AES block with *aes\_hash\_power ().*
2. Reset the AES hash block with *aes\_hash\_reset*().
3. Check for AES block busy with *aes\_busy():*If not busy perform the following
4. Set AES Private key LSB first random generated and optional initialization vector LSB first, also specify the AES key length used using api *aes\_set\_key()*
5. Check for AES status with rom API *aes\_status()* 
   1. clear status for any leftover status bits if any using API *aes\_iclr()*
6. Call API *aes\_crypt()* for encryption or decryption providing the message should be aligned input data buffer and pointer buffer to load aligned output data buffer and mode of operation ; supported mode ECB CBC CTR CFB 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 outpu will be in the buffer provided vis API *aes\_crypt()*
10. *Stop AES block using the API aes\_stop()*
11. For power saving and having AES block to sleep state using the routine *aes\_hash\_power(false);*

## Random number generator

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)
4. For random number generation use the function call as follows
5. Select the mode of operation by calling *rng\_mode(mode)*
   1. 0 – asynchronous
   2. 1 - pseudo-random mode
6. Start the HW block run state by calling function *rng\_start();*
7. Wait for operation completion by polling *rng\_get\_fifo\_level()* for data in the internal buffer
   1. Return 0 for not completion
   2. Non –zero value for completion of operation – FIFO will have random data
8. Once completion internal buffer will have 1Kbits of random data use API
   1. *rng\_get\_bytes()* number of random bytes to retrieve. Must be less or equal to the size of the buffer or
   2. *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 in 1024 bits

## ECDSA Verification

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. Call the ecdsa\_verify() API with the Public key, The signature of message and the digest calculated.
4. Start the PKE Engine by calling *pke\_start()*.
5. Wait until PKE operations are done by polling on *pke\_done\_status(PKE\_STATUS)*.
6. 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. Program curve parameters to slots using ec\_prog\_curve().
5. 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.
6. Call API to appropriate point operation with required parameters, data byte order.
7. 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.

# 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.

# 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

## AES



## ECDSA

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

## PKE

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

## SHA



## RNG

|  |  |  |
| --- | --- | --- |
| BLOCK | RNG | |
|
|
| BLOCK CONFIGURATION | TRUE RANDOM NUMBER | PSEUDO 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 |

# 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

# 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 |