Table of Contents

Prerequisites

If this lab is done in sequence with the other labs, then there are no prerequisites. All of the pre requisites should have been done in the "Getting Started" steps.
If you have not gone through these steps, please go back to the Getting Started lab guide and completed the download and setup of the IDE and SDK.

1. Download the latest MCUXpresso IDE for your platform https://mcuxpresso.nxp.com. It is required that MCUXpresso IDE 11.1.1 or newer is installed.
2. Download the latest SDK for your platform https://mcuxpresso.nxp.com.

Objectives

In this lab, you will learn:

- RT6xx SHA engine architecture
- How to use the SDK API for the HASH-CRYPT SHA engine
- Use of the MCUXpresso SDK API Reference Manual
- Test the results with SHA application note example

Hardware

- NXP i.MX RT600 Evaluation Kit - MIMXRT685-EVK
- Micro-USB cable

Lab high-level description

In this lab you will gain a basic understanding of the RT6xx HASH-CRYPT IP SHA engine features and capabilities.

SHA Engine

All RT6xx devices provide on-chip Hash support to perform SHA-1 and SHA-2 with 256-bit digest (SHA-256). Hashing is a way to reduce arbitrarily large messages or code images to a relatively small fixed size "unique" number called a digest. The SHA-1 Hash produces a 160-bit digest (five words), and the SHA-256 hash produces a 256-bit digest (eight words).

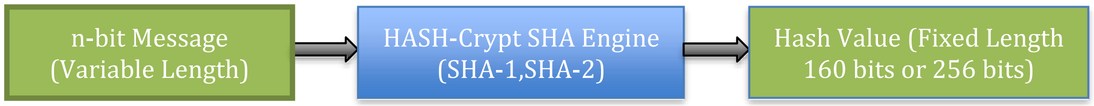


Figure 1. *SHA Engine Block Diagram*

For the SHA hardware:

- Even a small change to the input message will cause a major change in the digestoutput. Therefore, for a given input message or image there is only one digest.
- There is no predictable way to modify one input to result in a specific digest. A message cannot be added, inserted, or modified to get the same Hash in any direct way.

These two properties make it useful for verifying a message is valid, or corrupted intentionally or unintentionally.

Hashing is used for four primary purposes:

- Core of a digital signature model, including certificates, for secure update.
- Support a challenge/response or to validate a message when used with a Hash-based Message Authentication Code (HMAC).
- In a secure boot model, which verifies code integrity.
- Verify a block of memory that has not been compromised.

The key features of the SHA engine are:

- Performs SHA-1 and SHA-2(256) based hashing.
- Used with HMAC to support a challenge/response or to validate a message.

The SHA engine contains two message buffers which can be loaded by CPU, DMA or AHB bus master. The performance of the block depends on the memory from where the input data is fetched (Code RAM, system RAM or flash) and activity on the system bus. The SHA engine processes blocks of 512 bits (16 words) at a time and performs the SHA-1 hashing in 80 clock cycles per block or SHA-256 hashing in 64 clock cycles per block.

SDK API

The SDK HASHCRYPT API makes the SHA engine easy-to-use. The API consists of a handful of function calls to allow use of the SHA engine to perform SHA-1 or SHA-256 operation. Since the HASHCRYPT IP contains both the AES and SHA Engines, they also share the same API source (fsl_hashcrypt.c) and header file (fsl_hashcrypt.h). Only one operation either AES or SHA can run at a time on HASHCRYPT IP.

Application code must include the following header files:

```
#include "fsl_device_registers.h"
#include "fsl_hashcrypt.h"
```

The API documentation is available online:
https://mcuxpresso.nxp.com/api_doc/dev/1581/group_hashcrypt_driver.html
https://mcuxpresso.nxp.com/api_doc/dev/1581/group_hashcrypt_driver__hash.html

Below are the key excerpts from those sections of the SDK API manual.

HASHCRYPT algo type- selects which operation to run on HASHCRYPT

```
enum   hashcrypt_algo_t {
   kHASHCRYPT_Sha1 = HASHCRYPT_MODE_SHA1,
   kHASHCRYPT_Sha256 = HASHCRYPT_MODE_SHA256,
   kHASHCRYPT_Aes = HASHCRYPT_MODE_AES
}
```

HASHCRYPT parameter structure – defined in application code, passed to API functions:

```
/*! @brief Storage type used to save hash context. */
typedef struct _hashcrypt_hash_ctx_t
{
    uint32_t x[HASHCRYPT_HASH_CTX_SIZE]; /*!< storage */
} hashcrypt_hash_ctx_t;
```

Initialization API functions – connect and disconnect clocks, reset IP:

```
void HASHCRYPT_Init(HASHCRYPT_Type *base);
void HASHCRYPT_Deinit(HASHCRYPT_Type *base);
```

Perform the full SHA in one function call (this function internally calls the below mentioned function):

```
status_t HASHCRYPT_SHA (HASHCRYPT_Type *base, hashcrypt_algo_t algo, const uint8_t
*input, size_t inputSize, uint8_t *output, size_t *outputSize);
```

Initialize HASH context:

```
status_t HASHCRYPT_SHA_Init (HASHCRYPT_Type *base, hashcrypt_hash_ctx_t *ctx,
hashcrypt_algo_t algo);
```

Add data to current HASH:

```
status_t HASHCRYPT_SHA_Update (HASHCRYPT_Type *base, hashcrypt_hash_ctx_t *ctx, const
uint8_t *input, size_t inputSize);
```

Finalize hashing:

```
status_t HASHCRYPT_SHA_Finish (HASHCRYPT_Type *base, hashcrypt_hash_ctx_t *ctx, uint8_t
*output, size_t *outputSize);
```

Flow diagram explaining SDK APIs

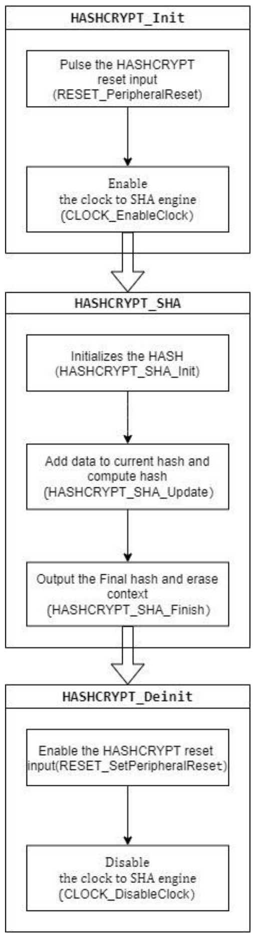

HASHCRYPT_Init
Pulse the HASHCRYPT reset input (RESET_PeripheralReset)
Enable the clock to SHA engine (CLOCK_EnableClock)
HASHCRYPT_SHA
Initializes the HASH (HASHCRYPT_SHA_Init)
Add data to current hash and compute hash (HASHCRYPT_SHA_Update)
Output the Final hash and erase context (HASHCRYPT_SHA_Finish)
HASHCRYPT_Deinit
Enable the HASHCRYPT reset input(RESET_SetPeripheralReset)
Disable the clock to SHA engine (CLOCK_DisableClock)

Figure 1. *Flow chart for SHA SDK API call*

RT685 EVK

The development platform is the NXP i.MX RT600 Evaluation Kit (part # MIMXRT685-EVK). The board contains an NXP MIMXRT685SFVKB device a 300 MHz with an ARM® Cortex® M33 CPU and a 600MHz Cadence® Tensilica® HiFi 4.

https://www.nxp.com/design/development-boards/i-mx-evaluation-and-development-boards/i-mx-rt600-evaluation-kit:MIMXRT685-EVK

The board contains on-board debug and virtual COM port through a single Micro USB interface, on-board flash memory and SD card socket for power-on boot, stereo audio input/output, and multiple expansion ports for prototyping.

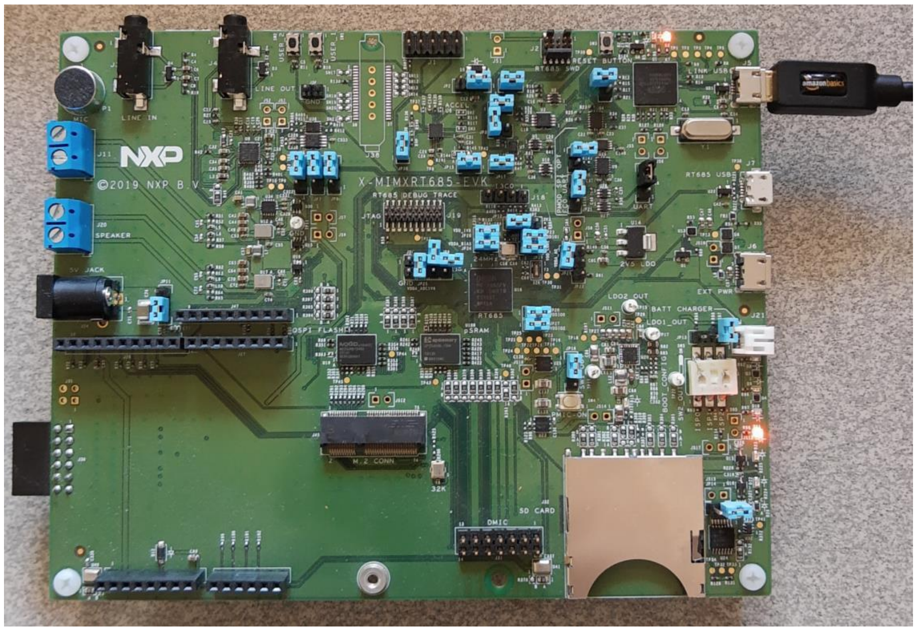


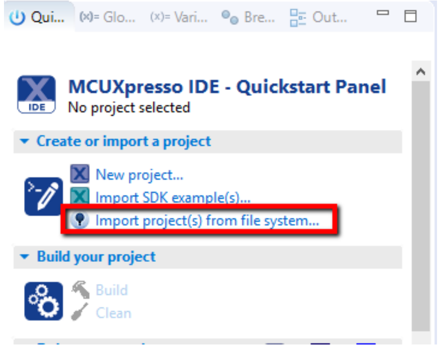
Figure 2. *NXP i.MX RT600 Evaluation Kit - MIMXRT685-EVK*

Running the lab

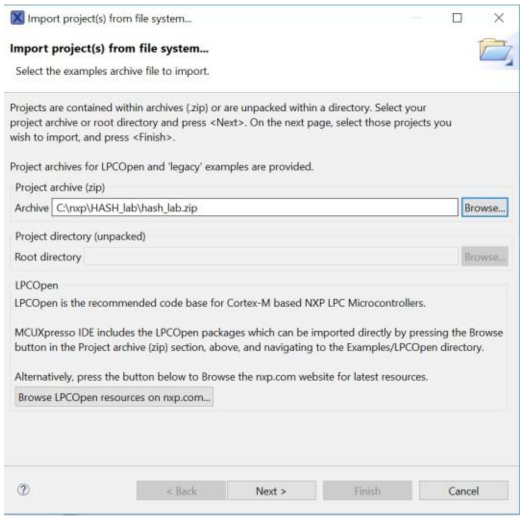
Import the MCUXpresso hash lab projects from file system

1. Download the hash_lab.zip folder from the NXP community from the same folder as this document.
2. Open MCUXpresso IDE 11.1.1 or newer.

3. On the **Quickstart Panel**, select **Import project(s) from file system…**


Qui… Glo… Vari… Bre… Out…
MCUXpresso IDE - Quickstart Panel
No project selected
Create or import a project
New project…
Import SDK example(s)…
Import project(s) from file system…
Build your project
Build
Clean

4. At the **Project archive (zip)** Browse… for the previously downloaded **hash_lab.zip** folder, then click **Next.**



Import project(s) from file system…

Import project(s) from file system…

Select the examples archive file to import.

Projects are contained within archives (.zip) or are unpacked within a directory. Select your project archive or root directory and press <Next>. On the next page, select those projects you wish to import, and press <Finish>.

Project archives for LPCOpen and 'legacy' examples are provided.

Project archive (zip)

Archive C:\nxp\HASH_lab\hash_lab.zip Browse…

Project directory (unpacked)

Root directory Browse…

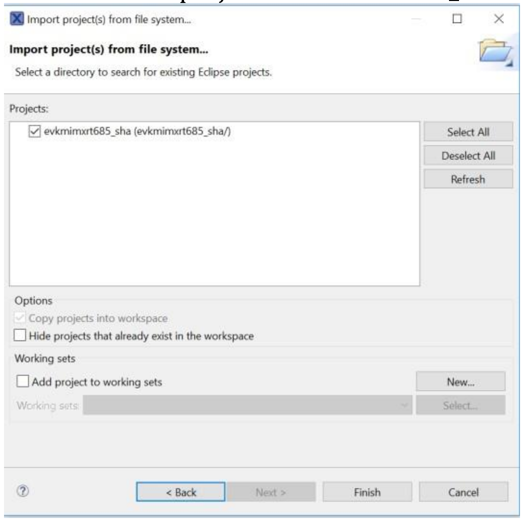
LPCOpen

LPCOpen is the recommended code base for Cortex-M based NXP LPC Microcontrollers.

MCUXpresso IDE includes the LPCOpen packages which can be imported directly by pressing the Browse button in the Project archive (zip) section, above, and navigating to the Examples/LPCOpen directory.

Alternatively, press the button below to Browse the nxp.com website for latest resources.

5. Make sure that project **evkmimxrt685_sha** is selected, then click **Finish.**



Import project(s) from file system…

Import project(s) from file system…

Select a directory to search for existing Eclipse projects.

Projects:

☑ evkmimxrt685_sha (evkmimxrt685_sha/)

Options

☑ Copy projects into workspace

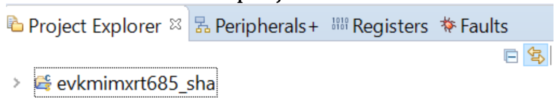
☐ Hide projects that already exist in the workspace

Working sets

☐ Add project to working sets

Working sets:

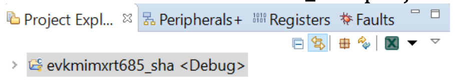
6. You should see new project loaded into the workspace as shown below.

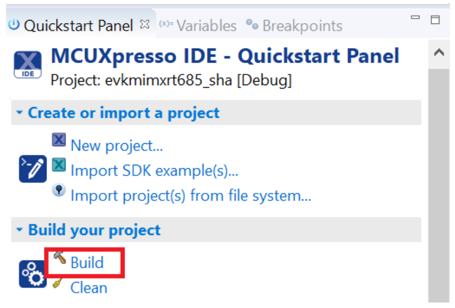


7. Project setting that is useful to modify:
Default is to link to execute in place (XIP) from flash memory. Code may instead reside in RAM when booting from flash memory or code is loaded from a host over a serial interface.
To link the image to load to RAM:
    i. Right click on project"evkmimxrt685_sha", select Properties.
    ii. Select C/C++ Build->Settings.
    iii. Choose the Tools Settings tab.
    iv. Click MCU Linker->Managed Linker Script.
    v. Click the checkbox for Link application to RAM.
    vi. If unchecked, code will be linked to reside in and execute from external flash memory (XIP). The image is programmed into flash as part of the debug step.
    vii. Click Apply and Close.

Build

8. Now it's time to compile.
Select the **evkmimxrt685_sha** project and click **Build.**

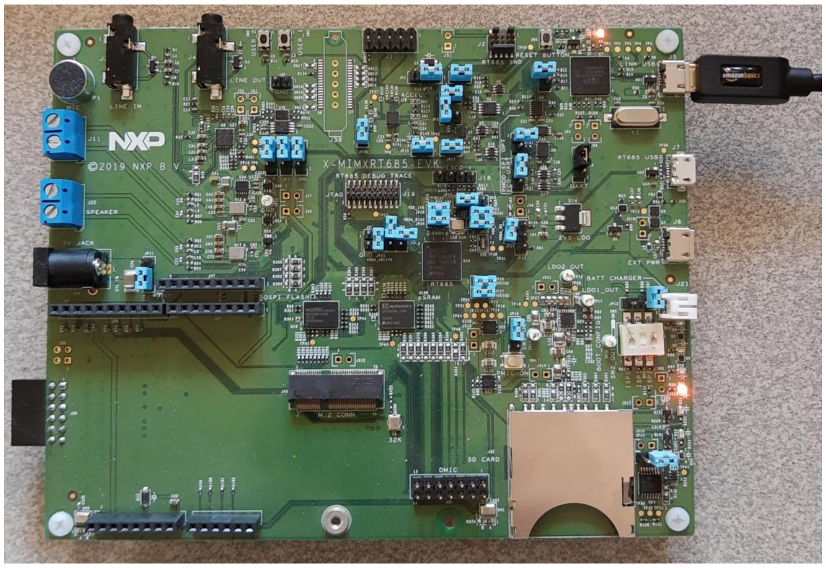



Quickstart Panel
Variables
Breakpoints
MCUXpresso IDE - Quickstart Panel
Project: evkmimxrt685_sha [Debug]
Create or import a project
New project...
Import SDK example(s)...
Import project(s) from file system...
Build your project
Build
Clean

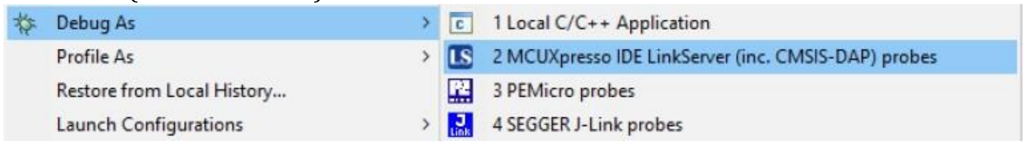
Wait until project finish building without any errors and warnings.

Debug

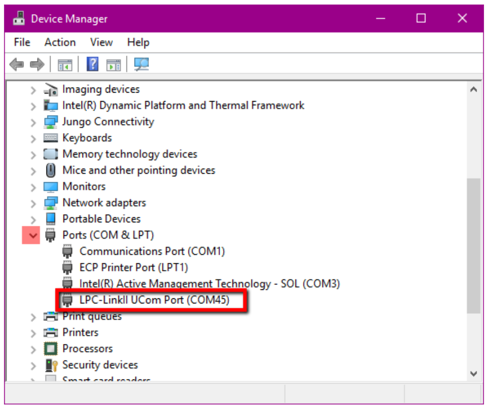
9. Connect the Debug Link port from your RT600 EVK to your PC using a micro-USB cable.



10. Code is built, EVK set up, connected, ready to debug.
    a. Right click on project “evkmimxrt685_sha”, select Debug As, select MCUXpresso IDE LinkServer (inc. CMSIS-DAP).


Debug As
Profile As
Restore from Local History...
Launch Configurations
1 Local C/C++ Application
2 MCUXpresso IDE LinkServer (inc. CMSIS-DAP) probes
3 PEMicro probes
4 SEGGER J-Link probes

    b. Code will download and debugger will point to function “main” in project source file test_sha.c.
    c. The code has similar initialization to the other demos in the SDK, specifically ported from the “hello_world” demo. Clocks, pins and the RS-232 port are all configured before the SHA specific code is executed.
    d. Open the **Device Manager** on your Windows PC and identify the COM port of your board.



Device Manager

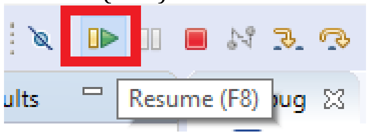
File Action View Help

- Imaging devices
- Intel(R) Dynamic Platform and Thermal Framework
- Jungo Connectivity
- Keyboards
- Memory technology devices
- Mice and other pointing devices
- Monitors
- Network adapters
- Portable Devices
- Ports (COM & LPT)
  - Communications Port (COM1)
  - ECP Printer Port (LPT1)
  - Intel(R) Active Management Technology - SOL (COM3)
  - LPC-LinkII UCom Port (COM45)
- Print queues
- Printers
- Processors
- Security devices
- Smart card readers

e. Open a Terminal emulator software (TeraTerm, PuTTY) and connect the COM port using the following settings:

```
115200 baudrate, 8 data bits, No parity, One stop bit, No flow control
```

f. To single-step over functions in MCUXpresso, type F6. To step into a function, type F5. To run, type F8, to stop click the Suspend button (pause button below). To the left of the Suspend button is the Resume (Run) button.



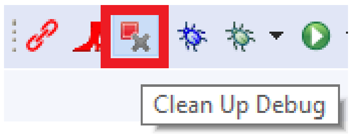
g. Breakpoints are supported, but are not discussed in this lab.

h. To verify the code works, simply type F8 to run.

i. The console window in the terminal program should display the following:

```
SHA-1 Test pass
SHA-256 Test pass
```

j. To leave the debugger, click the "Start/Stop Debug Session" button or select menu Debug->Start/Stop Debug Session.



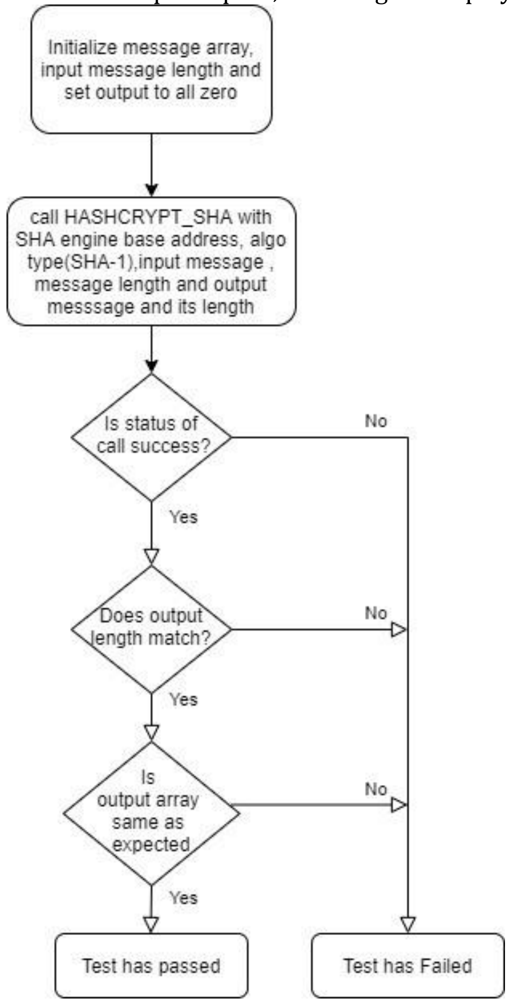
11. Run the code again, this time to step through code.
    a. Follow steps a-b in step 10 above.
    b. Code is at "main" in project source file test_sha.c.
    c. The SHA specific code is:

```
/* Initialize Hashcrypt */
HASHCRYPT_Init(HASHCRYPT);
/* Call HASH APIs */
TestSha1();
TestSha256();
HASHCRYPT_Deinit(HASHCRYPT);
```

    d. The functions with "HASHCRYPT" in the name are SDK API calls and are described above in "SDK API" section.
    e. The functions with "TestSha" in the name test a different SHA mode (SHA-1 and SHA-256) as supported by the SHA engine and are found in project source file test_sha.c.
    f. Function "TestSha1" tests SHA-1 mode.
        i. Press F6 until the cursor (green arrow) is pointed to "TestSha1".
        ii. Press F5 to step into the function.
        iii. The function has two constant arrays defined:
            1) input message of 56 bytes.
            2) expected output of sha1 operation of 20 bytes.
        iv. Call to SDK API function "HASHCRYPT_SHA" passes the hashcrypt base address, algo type (SHA1), input message, length of input and returns with a status of operation along with encrypted results into an output data array and the output length.

v. The result of operation is compared, if the result is success, the code continues to match the output length with the expected length and if it matches, the turned output data is compared to the expected sha as defined in the constant array. If any of compare fails, the program aborts with a message that SHA-1 failed.

vi. If all the compares pass, a message is displayed that SHA-1 Test has passed.


Initialize message array, input message length and set output to all zero
call HASHCRYPT_SHA with SHA engine base address, algo type(SHA-1),input message , message length and output messsage and its length
Is status of call success?
No
Yes
Does output length match?
No
Yes
Is output array same as expected
No
Yes
Test has passed
Test has Failed

vii. Function calls to similar SHA-1 tests are made from "main" for SHA-256.

viii. If results match as expected, a message is displayed that SHA-256 passed.

Conclusion

The lab provided simple example of using the SHA engine for testing SHA1 and SHA256. A SDK example project 'hashcrypt' is similar to this lab. The SDK example application note also includes Keil uVision and IAR Embedded Workbench projects along with NXP MCUXpresso projects.

Additional Resources

RT6xx User's Manual https://www.nxp.com/docs/en/user-guide/UM11147.pdf