AN13152
HASH Engine

Rev. 0 — 19 February 2021

Application Note

Contents

1 Introduction

This application note introduces users with the usage and configuration of SHA engine on RT500. It also demonstrates the performance improvement for speed, memory, and power consumption when SHA engine is used.

The AES and SHA engines are combined in RT500 and are called HASH-AES block which means that only one of the operations either AES or SHA can take place at a time.

2 Overview

The security system on RT5xx has a set of hardware blocks and ROM code to implement the security features of the device. The hardware consists of an AES engine, an SHA engine (Hash-AES block), a random number generator, and a key storage block that render keys from an SRAM-based PUF (Physically Unclonable Function).

2.1 HASH

All RT5xx 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).

For the SHA hardware:

- Even a small change to the input message causes a major change in the digest output. 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 that a message is valid, or corrupted intentionally or unintentionally. Hashing is used for four primary purposes:

- For secure update as it is core of a digital signature model, including certificates.
- To validate a message when used with a Hash-based Message Authentication Code (HMAC) or to support a challenge/response.
- To verify code integrity in a secure boot model.
- To verify a block of memory that has not been compromised.

3 SHA Engine

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. As many blocks as needed may be processed. The last block must be formatted per the SHA model:

1. The last data must be 447 bits or less. If more, then an extra block must be created.

2. After the last bit of data, a ‘1’ bit is appended. Then, as many 0 bits are appended to take it to 448 bits long (so, 0 or more).
3. Finally, the last 64 bits contain the length of the whole message, in bits, formatted as a word.

For example, if a message is an exact multiple of 512 bits, create an extra block. The first bit of the last block is 1 followed by 447 zeroes. The remaining 64 bits contain the length of the whole message including the last block.

The Arm processor uses little-endian and therefore, the SHA engine reverses the bytes in the words written to the data register to big-endian format. It is because a hash is on bytes, so a string such as “abcd”, when read as a word by the processor (or DMA) is reversed into “dcba”. When the input data is provided in little-endian format, the hash block swaps them to process correctly.

3.1 Features

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

3.2 Initialization

1. Pulse the HASHCRYPT reset input using the HASHCRYPT bit in the RSTCTL0_PRSTCTL0 register.

   ```
   RESET_PeripheralReset(kHASHCRYPT_RST_SHIFT_RSTn);
   ```

2. Enable the clock to the SHA engine.

   ```
   CLOCK_EnableClock(kCLOCK_HashCrypt);
   ```

3. Select SHA-1 or SHA-256 mode using the CTRL register.

   ```
   /** Peripheral HASHCRYPT base address */
   #define HASHCRYPT_BASE    (0x40208000u)
   /** Peripheral HASHCRYPT base pointer */
   #define HASHCRYPT         ((HASHCRYPT_Type *)HASHCRYPT_BASE)
   HASHCRYPT->CTRL = HASHCRYPT_CTRL_MODE(algo);
   Where algo can be SHA1 or SHA256.
   ```

4. To start a new Hash, write 1 to the NEW_HASH bit field in CTRL register. This bit automatically self-clears.

   ```
   HASHCRYPT->CTRL = HASHCRYPT_CTRL_NEW_HASH(1);
   ```

5. For AHB master mode,

   a. Enable the interrupt for HASH using HASHCRYPT_IRQn.

      ```
      (void)EnableIRQ(HASHCRYPT_IRQn);
      ```

   b. Enable the DIGEST and ERROR interrupts using INTENSET register which indicates if should interrupt when Digest (or Outdata) is ready (completed a hash/crypto or completed a full sequence) or if should interrupt on an ERROR (as defined in Status).

      ```
      HASHCRYPT ->INTENSET = HASHCRYPT_INTENCLR_DIGEST_MASK | HASHCRYPT_INTENCLR_ERROR_MASK;
      ```

   c. Write to the MEMADDR register with the offset in SRAM or flash.

      ```
      HASHCRYPT ->MEMADDR = HASHCRYPT_MEMADDR_BASE(HASHCRYPT);
      ```

   d. Write to the MEMCTRL register to enable the SHA engine as AHB bus master using the MASTER bit and write the number of 512-bit blocks to process in the COUNT field.

```
HASHCRYPT ->MEMCTRL = HASHCRYPT_MEMCTRL_MASTER(1) | HASHCRYPT_MEMCTRL_COUNT(numBlocks);
Where numBlocks is block to process.
```

6. ISR for AHB master:

- An ERROR would be a bus error, so the algorithm is:

  If ERROR is set in the STAT register, there is AHB master bus fault. The COUNT field in the MEMCTRL register indicates which block it was processing and the MEMADDR register indicates which memory location it was on when the error occurred.

```
if (1U == (HASHCRYPT->STATUS & HASHCRYPT_STATUS_ERROR_MASK))
```

- If the DIGEST bit is set in STAT register:

  -DIGEST is ready and so process the Digest register (for example, copy) and clear the interrupt using INTENCLR register.

```
HASHCRYPT ->INTENCLR = HASHCRYPT_INTENCLR_DIGEST_MASK |    HASHCRYPT_INTENCLR_ERROR_MASK;
HASHCRYPT ->MEMCTRL = HASHCRYPT_MEMCTRL_MASTER(0);
```

One can also wait for DIGEST to be ready using polling.

```
/* Wait until digest is ready */
while (0 == (HASHCRYPT ->STATUS & HASHCRYPT_STATUS_DIGEST_MASK))
{
}
```

3.3 APIs related to HASH engine

The below mentioned APIs are defined & declared in fsl_hashcrypt.c/.h inside the SDK (SDK/devices/MIMXRT595S/drivers):

1.
```
void HASHCRYPT_Init(HASHCRYPT_Type * base) ;
```

   Enable clock and disable reset for HASHCRYPT.

   Parameters:

   `-base HASHCRYPT base address`

2.
```
void HASHCRYPT_Deinit(HASHCRYPT_Type * base) ;
```

   Disable clock and enable reset.

   Parameters:

   `-base HASHCRYPT base address`

3.
```
status_t HASHCRYPT_SHA (    HASHCRYPT_Type *    base,
                            hashcrypt_algo_t    algo,
                            const uint8_t *     input,
                            size_t              inputSize,
                            uint8_t *           output,
                            size_t *            outputSize
);
```

   Perform the full SHA in one function call. The function is blocking.

Parameters:

```
-base                    HASHCRYPT peripheral base address
-algo                    Underlying algorithm to use for hash computation.
-input                   Input data
-inputSize               Size of input data in bytes
-[out]output             Output hash data
-[out]outputSize         Output parameter storing the size of the output hash in bytes
```

Results:

Status of the one call hash operation.

4.
```
status_t HASHCRYPT_SHA_Init    ( HASHCRYPT_Type *          base,
                                 hashcrypt_hash_ctx_t *    ctx,
                                 hashcrypt_algo_t          algo
                                 );
```

This function initializes the HASH.

`-base HASHCRYPT peripheral base address`

`-[out] ctx Output hash context`

`-algo Underlying algorithm to use for hash computation.`

Results:

Status of initialization

5.
```
status_t HASHCRYPT_SHA_Update      ( HASHCRYPT_Type *           base,
                                     hashcrypt_hash_ctx_t *     ctx,
                                     const uint8_t  *           input,
                                     size_t                     inputSize
);
```

Add data to current HASH. This can be called repeatedly with an arbitrary amount of data to be hashed. The functions blocks. If it returns kStatus_Success, the running hash has been updated (HASHCRYPT has processed the input data), so the memory at input pointer can be released back to system. The HASHCRYPT context buffer is updated with the running hash and with all necessary information to support possible context switch.

Parameters:

`-base HASHCRYPT peripheral base address`

`-[in ,out] ctx Hash context`

`-input Input data`

`-inputSize Size of input data in bytes`

Results

Status of the hash update operation.

6.
```
status_t HASHCRYPT_SHA_Finish (    HASHCRYPT_Type *           base,
                                   hashcrypt_hash_ctx_t *     ctx,
                                   uint8_t *                  output,
                                   size_t *                   outputSize
);
```

Outputs the final hash (computed by HASHCRYPT_HASH_Update()) and erases the context. Parameters:

`-base HASHCRYPT peripheral base address`

```
-[in ,out] ctx Hash context

-[out] output Output hash data

-[in ,out] outputSize Optional parameter (can be passed as NULL). On function entry,

it specifies the size of output[] buffer. On function return, it stores the number of updated
output bytes.
```

Results:

Status of the hash finish operation.

To understand how these APIs are used at the application level, see the SDK example hashcrypt inside the driver_example folder . Hashcrypt is a demonstration program that uses the KSDK software to encrypt plain text and decrypt it back using SHA algorithm. SHA-1 and SHA-256 modes are demonstrated.

3.4 Performance

The SHA engine contains two message buffers which are 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. For details on how to format the last block, see SHA Engine.

3.4.1 Input data loaded by CPU

The Cortex-M33 core writes 16 words to start Hashing. The Hash operation takes 64 or 80 clock cycles based on SHA-1 or SHA-256 Hash algorithm. The processor can load the next 16 words during the time when the Hash operation is being performed on the previous loaded data.

3.4.2 Input data loaded by DMA

The DMA loads the 16 words based on requests. The Hash operation takes 64 or 80 clock cycles based on SHA-1 or SHA-256 Hash algorithm. The DMA can request and load the next 16 words during the time when the Hash operation is being performed on the previous loaded data.

3.4.3 Input data loaded by AHB bus master

The AHB bus master loads 16 words from memory. The Hash operation takes 64 or 80 clock cycles based on SHA-1 or SHA-256 Hash algorithm. The AHB master can load the next 16 words during the time when the Hash operation is being performed on the previous loaded data.

4 Demo Application

As part of this application, we run the SDK example to analyze the performance of HASH engine.

The HASH-AES block which is implemented in software uses the CPU resources for performing hashing operations. In hardware hashing, the HASH-AES engine is integrated on-chip, therefore, hashing is performed without involvement of the CPU. This offloads the CPU, allowing it to perform other tasks and improve system performance.

Crypto acceleration has several advantages over software implementation. Crypto acceleration is:

- Fast as a dedicated hardware block is used for hashing process.
- Prevention of brute force and cold boot attacks.
- Independent of operating system.
- Reduction in code size as hashing is done in hardware.

This application note illustrates the performance boost obtained using hardware HASH engine.

4.1 Environment

- Hardware environment
- Software environment

4.1.1 Hardware environment

- Board
  - MIMXRT595EVK
- Debugger
  - Integrated CMSIS-DAP debugger on the board
- Miscellaneous
  - 1 Micro USB cable
  - Personal Computer (PC)
- Board Setup
  - Connect the micro USB cable between PC and J40 link on the board for loading and running a demo.

4.1.2 Software environment

- Tool chain
  - IAR embedded workbench 8.50.9 or MCUXpresso IDE 11.3.0 or Keil 5.33
- Software package
  - SDK_2.9.1_EVK_MIMXRT595

NOTE

Select mbedtls while downloading the SDK.

4.2 Steps and result

1. For details on the steps for running a mbedtls_benchmark demo (*SDK\boards\evkmimxrt595\mbedtls_examples\mbedtls_benchmark*) using MCUXpresso, IAR, or Keil, see the *Getting Started with MCUXpresso SDK for MIMXRT500* found inside *SDK > Docs*).
2. Connect the development platform to your PC via USB cable.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug serial port number to determine the COM port number. Configure the terminal with the following settings.
   - 115200 baud rate
   - 8 data bits
   - No Parity
   - 1 Stop bit
   - No Flow control

   The mbedtls_application now runs and following prints can be seen on serial terminal.

   mbedTLS version 2.16.6

   fsys=198000000

   Using following implementations:

   SHA: HASHCRYPT HW accelerated

SHA-1: 35423.07 KB/s, 4.56 cycles/byte

SHA-256: 34721.67 KB/s, 4.45 cycles/byte

SHA-512: 462.34 KB/s, 417.64 cycles/byte

HMAC_DRBG SHA-1 (NOPR): 960.35 KB/s, 200.79 cycles/byte

HMAC_DRBG SHA-1 (PR): 877.46 KB/s, 219.85 cycles/byte

HMAC_DRBG SHA-256 (NOPR): 1198.57 KB/s, 159.73 cycles/byte

HMAC_DRBG SHA-256 (PR): 1198.58 KB/s, 159.73 cycles/byte

NOTE

Only SHA results are included in above output and other security operations results are excluded as they do not use HASH engine. The above numbers vary on what toolchain is used and what optimization level is set for the toolchain. The numbers are better with higher optimization level (O3 for MCUXpresso and IAR & Ofast for Keil). The above mentioned numbers are w.r.t MCUXpresso at no optimization and running from FLASH.

4. Open the MIMXRT595S_cm33_features.h file (present inside device folder inside mdbedtls_benchmark project inside each IDE) and comment out or remove the below line:

```
#define FSL_FEATURE_HASHCRYPT_HAS_RELOAD_FEATURE (1)
```

This will undefine macros: MBEDTLS_FREESCALE_HASHCRYPT_SHA1 and MBEDTLS_FREESCALE_HASHCRYPT_SHA256 which removes the hardware acceleration and uses software-based implementation for SHA1 and SHA256.

Repeat the steps 1 to 3 and the following prints can be seen on serial terminal.

mbedTLS version 2.16.6

fsys=198000000

Using following implementations:

SHA: Software implementation

SHA-1: 2857.04 KB/s, 66.95 cycles/byte

SHA-256: 1109.25 KB/s, 173.49 cycles/byte

SHA-512: 461.06 KB/s, 418.92 cycles/byte

HMAC_DRBG SHA-1 (NOPR): 207.33 KB/s, 936.38 cycles/byte

HMAC_DRBG SHA-1 (PR): 192.35 KB/s, 1009.73 cycles/byte

HMAC_DRBG SHA-256 (NOPR): 118.60 KB/s, 1639.62 cycles/byte

HMAC_DRBG SHA-256 (PR): 118.70 KB/s, 1639.81 cycles/byte

5. **Results:**

   The performance with hardware acceleration is much better in terms of number of cycles per byte for SHA and HMAC operations indicating that for similar bytes of data with hardware acceleration enabled, hash is computed much faster. Also, the code size for binaries built using hardware acceleration enabled is smaller as compared to software implementation. Also, as the CPU can be unloaded during SHA and HMAC operations, the power consumption would be less when crypto accelerator is used.

5 Conclusion

The example shows that with HASH-AES engine enabled, the SHA, and HMAC operations are executed much faster. The HASH-AES crypto engine takes lesser text/code as compared to software implementation and as the CPU is offloaded while the security operations are executed, it takes lesser power to operate.

6 References

1. RT500 user manual.
2. RT500 data sheet.
3. MCUXpresso SDK Release Notes for EVK-MIMXRT595 (Found inside SDK)
4. Getting Started with MCUXpresso SDK for EVK-MIMXRT595 (Found inside SDK).
5. MCUXpresso SDK API Reference Manual

7 Revision history

This table summarizes the revisions of this document.

Table 1. Revision history

Revision number	Date	Substantive changes
0	19 Feb 2021	Initial Draft

Date of release: 19 February 2021

Document identifier: AN13152

arm