Embedding it better...

µTasker Document

µTasker – Cryptography

Table of Contents

1. Introduction

This document describes cryptography functions that are integrated in the μTasker project, as well as possible hardware acceleration in conjunction with encryption units in some processors.

The μTasker project encapsulates some common functions in order to achieve high portability which is not directly dependent on the underlying library used. When possible it also uses additional capabilities of the processor or manufacturer libraries to optimise the implementation.

2. AES

A popular symmetric key algorithm is AES (Advanced Encryption Standard), also known as Rijndeal (its original name). It is a specification for the encryption of electronic data by the US National Institute of Standards (NIST). It is a block cypher used with a block size of 128 bits but a choice of key lengths of either 128, 192 or 256 bits.

The AES is defined in the standards

- FIPS PUB 197
- ISO/IEC 18033-3

The input (to be encrypted) is called "plaintext" and the output of the AES encryption process is called "ciphertext".

AES algorithms are included in OpenSSL, WolfSSL and mbedTLS libraries, whereby also assembler code implementations are included for some processors.

Some Coldfire and Kinetis processors contain a CAU (Crypographic Acceleration Unit) which allows acceleration by using a coprocessor to handle some parts of the AES algorithm, for which Freescale/NXP supply assembler code implementations for m0+ and m4 Cortex processors. *The unit is referred to as mmCAU in the Kinetis parts*

In some newer Kinetis parts there may also be an LTC (Low Power Trusted Cryptography) hardware block that supports AES. It is based on an accelerator fed and via an input FIFO and results are retrieved via an output FIFO. This module achieves even better throughput than the mmCAU.

2.1 Basics of AES Operation

A secret key (called the cipher key) of either 128, 192 or 256 bits is available at both sender and receiver. This is used to generate a encryption key schedules that will later be used to encrypt messages.

Therefore before use, the key schedules are initialised (once).

AES is a block cipher and the block size is 16 bytes. Each plain text input to be encrypted to cipher text must be a multiple of the block length, padding being added if needed to ensure this is true.

Encryption of a 16 byte block results in 16 bytes of cipher output and this is the smallest entity of encryption (and decryption). However, when data messages of multiple block lengths is performed it is not usual to simply do multiple block encryptions (known as ECB – *Electronic Code Book*) since it isn't considered as a secure technique and *OpenSSL advises against ever using this form*. The most common method is to perform CBC (Cipher Block Chaining) where an initial vector (IV) is XORed with the data of each plain text block. After encrypting the block the cipher block output then becomes the IV for operation on the input of the next block. The CBC decryption process of a message is the inverse: each block is decrypted and XORed with the original cipher block input to generate the final plain text output again.

Often the IV is set to zero at the start of each new message, although this weakens security over using a random IV content. Random IV content is not practical when the same random seed can not be shared by sender and receiver, but a fixed, non-zeroed IV could be agreed on.

2.2 Configuring and using AES

The µTasker project's AES interface encapsulates the low level implementation and thus allows various libraries to be used without the application layer needing to be aware of which one it actually is.

The following configuration options are supported:

`#define CRYPTO_WOLF_SSL` - use wolfSSL library code for engine implementation.

`#define CRYPTO_MBEDTLS` - use mbedTLS library code for engine implementation.

`#define CRYPTO_OPEN_SSL` - use OpenSSL library code for engine implementation.

`#define CRYPTO_AES` - Enable AES support.

```
#define MBEDTLS_AES_ROM_TABLES
```

– when mbedTLS is used the tables used for internal calculations are calculated rather than being fixed in Flash The resulting tables are in RAM and so occupy more RAM space but this cases program space and also operation with tables in RAM may be faster than in Flash. See comparisons later.

```
#define OPENSSL_AES_FULL_LOOP_UNROLL
```

– when OpenSSL is used loops are unrolled in code to improve performance. See comparisons later.

```
#define NATIVE_AES_CAU
```

- when a crypto accelerator (CAU) or LTC (Low power Trusted Crypto) is available in the processor this define will use it directly from the optimised µTasker interface rather than as hook in the library code. See comparisons later.
Library code is not needed in this case for operation on the HW but a base library should still be selected for use by the simulator.

```
#define AES_DISABLE_CAU
```

- When devices with crypto accelerator (CAU) are used, the HW acceleration is enabled automatically. This define can be used to disable it for performance comparisons.

```
#define AES_DISABLE_LTC
```

- When devices with low power trusted crypto (LTC) are used, the HW acceleration is enabled automatically. This define can be used to disable it for performance comparisons.

In case a device has both mmCAU and LTC the LTC has priority if not explicitly disabled.

The µTasker project includes a single AES initialisation routine as cover function for all possible underlying library implementations:

```
extern int fnAES_Init(int iInstanceCommand,        // command and instance reference
                      const unsigned char *ptrKey, // buffer containing secret key
                      int iKeyLength);             // the key length to be used (bits)
```

The following example initialises an instance for subsequent encryption use based on the passed secret key and its size.

```
    fnAES_Init(AES_COMMAND_AES_SET_KEY_ENCRYPT, key, 256);
```

where `key[]` is a const array of 256/8 bytes in this example.

If multiple instances with different keys are to be used each one can be configured in the same manner. In this case the number of instances in the project (example of 4) can be defined by

```
#define AES_INSTANCE_COUNT   4
```

where this shows 4 instances being reserved (without a define it defaults to 2 instances – one for encryption and one for decryption).

Eg. second instance being initialised for decryption

```
    fnAES_Init((AES_COMMAND_AES_SET_KEY_DECRYPT | 1), key, 256);
```

The initialisation interface can also be used to prime or zero the content of the initial vector.

```
fnAES_Init(AES_COMMAND_AES_PRIME_IV, iv, 0);
```

where `iv[]` is an array of 16 bytes that will be written the the instance's initial vector.

```
fnAES_Init(AES_COMMAND_AES_RESET_IV, 0, 0);     // zero the initial vector
```

The interface used to encrypt and decrypt data is:

```
extern int fnAES_Cipher(int iInstanceCommand,        // command and instance reference
                        const unsigned char *ptrTextIn,// input text (plain or cipher)
                        unsigned char *ptrTextOut,   // output text (cipher or plain)
                        unsigned long ulDataLength);
                                       // length of text (multiple blocks of 16 bytes)
```

This is used for both encryption and decryption whereby the length of the data buffer must be a multiple of 16 bytes otherwise the call will fail with a return value equal to `AES_ENCRYPT_BAD_LENGTH`. It is therefore up to the caller to ensure that all plain text fulfils this requirement by padding to a 16 bytes boundary if necessary.

To encrypt plain text the following call is used:

```
   fnAES_Cipher(AES_COMMAND_AES_ENCRYPT, plaintext, ciphertext, sizeof(plaintext));
```

Multiple calls to encrypt plain text will be chained unless a new sequence is defined by priming the value or calling the encryption with the flag `AES_COMMAND_AES_RESET_IV` as shown here (which zeroes the initial vector when the message encryption/decryption starts):

```
   fnAES_Cipher((AES_COMMAND_AES_ENCRYPT | AES_COMMAND_AES_RESET_IV), plaintext,
                ciphertext, sizeof(plaintext));
```

Decryption is the inverse and always decrypts a multiple of 16 bytes, some possibly being padding.

```
fnAES_Cipher((AES_COMMAND_AES_DECRYPT | AES_COMMAND_AES_RESET_IV),
             (const unsigned char *)ciphertext, recovered, sizeof(ciphertext));
```

The original plain text is finally in the plain text buffer `recovered[]`.

Warning: When using the mmCAU or LTC the secret keys passed to the initialisation routine and the destination buffers used during encryption and decryption must be l*ong word aligned*. *See the AES256 test code in the µTasker project for a technique to ensure alignment without issues with compiler portability.*

When using the option `NATIVE_AES_CAU` also the input buffer must be long word aligned to achieve maximum performance.

2.3 AES Benchmarks

The following tables compare the processing time and memory of various configurations on a 120MHz K64 (Cortex M4 with FPU and mmCAU), with 60MHz bus clock and 24MHz Flash clock. (GCC compiler with optimisation for size).

Processing times [Set encryption key/encrypt 32 bytes/set decryption key/decrypt 32 bytes] **µs** units	WolfSSL AES256	mbedTLS AES256 with ROM tables*	mbedTLS AES256 with generated RAM tables
CAU disabled	**15.5 / 73.0 / 61.3 / 71.1**	**11.5 / 59.7 / 59.6 / 61.5**	**8.5 / 35.6 / 33.8 / 35.8**
mmCAU enabled	**5.3 / 17.3 / 5.3 / 16.0**	**5.0 / 14.7 / 5.1 / 16.5**	**5.0 / 14.7 / 5.1 / 16.5**

Comparison with 48MHz KL43 (Cortex m0+ without mmCAU): 27.8us/159us/147us/154us

Memory [Flash / RAM] **bytes** units (not including application calling the library functions)	WolfSSL AES256	mbedTLS AES256 with ROM tables	mbedTLS AES256 with generated RAM tables
CAU disabled	**13'624 / ~250**	**11'500 / ~250**	**3'276 / ~9'000**
mmCAU enabled	**2'506 / ~250**	**2'470 / ~250**	**2'470 / ~250**

It is interesting to note that the mmCAU not only increases performance by a factor of about 4 for encryption and decryption but it can also save almost 10k of Flash space (code and tables).

When no mmCAU is available there is a further interesting increase in performance by locating tables in RAM (rather than Flash) and a saving in Flash by generating the content rather than taking it from a const look-up table. This is due to the faster RAM access possible in the K64 but there is of course a trade off between the improved performance and the RAM consumption.

The same measurements for OpenSSL 1.0.2 are shown next:

Processing times [Set encryption key/encrypt 32 bytes/set decryption key/decrypt 32 bytes] **µs** units	OpenSSL AES256	OpenSSL AES256 with option OPENSSL_AES_FULL_LOOP_UNROLL
CAU disabled	**13.6** / **66.7** / **58.0** / **59.4**	**13.6** / **53.7** / **58.0** / **51.0**
mmCAU enabled	**5.1** / **13.6** / **5.3** / **11.3**	**5.1 / 13.6 / 5.3 / 11.3**

Memory [Flash / RAM] **bytes** units (not including application calling the library functions)	OpenSSL AES256	OpenSSL AES256 with option OPENSSL_AES_FULL_LOOP_UNROLL
CAU disabled	**12'052** / **~250**	**16'260** / **~250**
mmCAU enabled	**2'982** / **~250**	**2'982** / **~250**

If the processor has an mmCAU the AES256 function can also be performed natively (using the µTasker interface code used to control this). The following figures show the additional improvement in performance in such a case:

Processing times [Set encryption key/encrypt 32 bytes/set decryption key/decrypt 32 bytes] **µs** units	µTasker AES256 using mmCAU*
mmCAU enabled	**4.9** / **8.4** / **4.9** / **8.1**

Some Kinetis devices have both mmCAU and LTC (LP Trusted Crypography) and the follow gives some comparisons with its use:

Processing times	[Set encryption key/encrypt 32 bytes/set decryption key/decrypt 32 bytes] **µs** units
150MHz K82F using mmCAU	**4.5** / **7.0** / **4.5** / **6.5**
150MHz K82F using LTC	**4.4** / **5.5** / **4.4** / **-.-***

72MHz KL82 using LTC	**3.6 / 9.7 / 3.6 / 15.9 (12.8)***

When the LTC performs decryption it first derives the decrypt key from the encrypt key meaning that subsequent block decrypts tend to be faster,

Memory [Flash / RAM] **bytes** units (not including application calling the library functions)	μTasker AES256 using mmCAU
mmCAU enabled	**2'890 / ~250**

To get an idea of the expense of the 256 bit AES the processing time for 128 and 196 bit keys are compared (μTasker interface):

Processing times [Set encryption key/encrypt 32 bytes/set decryption key/decrypt 32 bytes] **μs** units	μTasker AES128 using mmCAU	μTasker AES192 using mmCAU	μTasker AES256 using mmCAU
mmCAU enabled	**4.5 / 7.0 / 4.6 / 6.9**	**4.9 / 7.6 / 5.0 / 7.5**	**4.9 / 8.4 / 4.9 / 8.1**

The final comparison is with the mmCAU disabled, based on a typical library configuration:

Processing times [Set encryption key/encrypt 32 bytes/set decryption key/decrypt 32 bytes] **μs** units	mbedTLS AES128 with ROM tables	mbedTLS AES192 with ROM tables	mbedTLS AES256 with ROM tables
mmCAU disabled	**8.4 / 46.8/ 42.5 / 48.5**	**8.2 / 53.8 / 48.3 / 55.6**	**11.5 / 59.7 / 59.6 / 61.5**

3. SHA-2 (Secure Hash Algorithm 2)

A set of six cryptographic hash functions that take an arbitrary length input and generate digests of 224, 256, 384 or 512 bits in length and used widely in security application (including TLS) for MACs (Message Authentication Codes) and digital signatures. SHA-1 is a predecessor of SHA-2 but is being phased out and can't be used for many modern secured connections. SHA-3 exists but is new and so SHA-2 tends to be still suitable/accepted for present day communication techniques.

The properties of the hash algorithm is that given the digest (the output) it is extremely difficult to compute the original input (*irreversibility*). There should also be only one input that generates the digest output (multiple inputs giving the same output are called *collisions*), whereby this is limited by half of the length and so is often what determines the effectiveness of the chosen length rather than its reversibility strength.

Message digests are very interesting due to the fact that it allows giving the receiver a piece of information that proves that the sender is in possession of a particular piece or information (a secret) without the sender having to divulge this secret. Since only this secret can generate the particular digest, the sender could later prove the existence of the particular secret by re-generating the same hash from it (with any other secret or any modified secret it would be impossible to do this) but the receiver cannot calculate the secret from the received digest value (*irreversibility*). Inherently it also ensures that the sender cannot change the secret from its original since if the sender couldn't later reproduce the same digest value if it were ever modified.

Pseudocode for SHA-256 is available at many sources in the Internet.

The CAU in the Kinetis / Coldfire devices supports SHA-1 and SHA-256 (one useful and well excepted length from SHA-2).

Since SHA-256 is very popular it's use and performance are compared in the following sections.

The SHA-256 digest of an empty string is (32 byte long hex value)

0xe3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855

3.1 Configuring and using SHA

The µTasker project's SHA interface encapsulates the low level implementation and thus allows various libraries to be used without the application layer needing to be aware of which one it actually is.

The following configuration options are supported:

`#define CRYPTO_WOLF_SSL` - use wolfSSL library code for engine implementation

`#define CRYPTO_MBEDTLS` - use mbedTLS library code for engine implementation

`#define CRYPTO_OPEN_SSL` - use OpenSSL library code for engine implementation

`#define CRYPTO_SHA` - Enable SHA support.

mbedTLS option:
`#define MBEDTLS_SHA256_SMALLER`

- produces smaller code at the cost of speed.

`#define NATIVE_SHA256_CAU`

- when a crypto accelerator (CAU) or LTC (Low power Trusted Crypto) is available in the processor this define will use it directly from the optimised µTasker interface rather than as hook in the library code. See comparisons later.
Library code is not needed in this case for operation on the HW but a base library should still be selected for use by the simulator.

`#define SHA_DISABLE_CAU`

- When devices with crypto accelerator (CAU) are used, the HW acceleration is enabled automatically. This define can be used to disable it for performance comparisons.

`#define SHA_DISABLE_LTC`

- When devices with low power trusted crypto (LTC) are used, the HW acceleration is enabled automatically. This define can be used to disable it for performance comparisons.

In case a device has both mmCAU and LTC the LTC has priority if not explicitly disabled.

Single 32 byte SHA-256	**LTC**	**mmCAU**	**Software**
KL82 72MHz	4.96µs	-	120µs
K64 120MHz	-	26µs	35µs
K82F 120MHz			

4. Conclusion

This document has introduced cryptographic functions that are integrated in the µTasker project, explained their practical usage and given some benchmarks as to performance and memory costs.

For details discussions of the underlying algorithms of the Crypto functions the interested reader is referred to academic sources and governing standards where available.

Modifications:

V0.00 10.1.2017:
V0.01 28.1.2017: added mbedTLS and WolfSSL measurements
V0.02 29.1.2017: added OpenSSL and native µTasker CAU measurements
V0.03 9.3.2017: add LTC benchmarks
V0.04 9.11.2018: added further LTC benchmarks and SHA descriptions