AN0955: CRYPTO

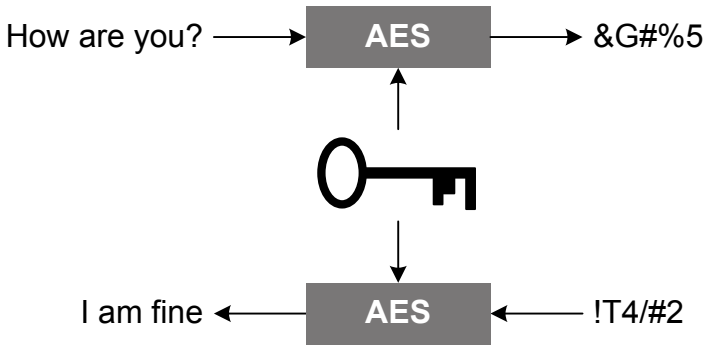
This document describes the use of the CRYPTO acceleration module of the EFM32 Gemstones, including support for ECC, SHA, AES block ciphers, and authenticated encryption algorithms.

This document focuses on the Silicon Labs mbedTLS library functions that support the CRYPTO module, how the library works, and how to integrate the library into an application.

This document assumes familiarity with the crypto algorithms discussed. See the device Reference Manual for more information on the CRYPTO module hardware.

KEY POINTS

- The mbedTLS library includes functions to implement various cryptographic algorithms.
- The Silicon Labs mbedTLS functions use the CRYPTO module to hardware accelerate the cryptographic algorithms.
- The included example demonstrates how to use the Silicon Labs mbedTLS library.



1. Introduction

The EFM32 Gemstones integrate a crypto accelerator providing the capability to include high performance cryptography solutions in a small footprint with low power consumption.

The CRYPTO module allows efficient acceleration of common cryptographic operations and allows these to be used efficiently with low CPU overhead. The CRYPTO module includes hardware accelerators for the Advanced Encryption Standard (AES), Secure Hash Algorithm SHA-1 and SHA-2 (SHA-224 and SHA-256), and modular multiplication used in ECC (Elliptic Curve Cryptography) and GCM (Galois Counter Mode). The CRYPTO module can autonomously execute and iterate a sequence of instructions to aid software and speed up complex cryptographic functions like ECC, GCM, and CCM (Counter with CBC-MAC).

The CRYPTO module is capable of autonomously fetching data, performing block cipher operations and storing data across multiple blocks. Block cipher modes of operation such as Counter Mode (CTR), Cipher Block Chaining (CBC), CBC-MAC (CBC Message Authentication Code), CCM (Counter with CBC-MAC), and GCM (Galois Counter mode) can be implemented using libraries and examples provided.

Silicon Labs provides a library based on the ARM mbedTLS library that takes advantage of the CRYPTO module. There are several code examples available within the Simplicity Studio that provide a starting point for custom applications and illustrate the use of the library. This section follows the code flow of one of the example projects to illustrate the use of the library.

2. Hardware Acceleration and Software Implementations

Any cryptographic algorithm can be implemented in software. However, there are a number of benefits to accelerating cryptographic operations in hardware:

- Faster execution
- More energy efficient
- Offloads the main core
- Saves code space
- More resistant to differential power analysis (DPA) attacks

The Silicon Labs mbedTLS library accelerates all modern algorithms such as AES, ECC, and SHA partly or fully in hardware. In addition, the library supports a number of algorithms such as RSA, DES, 3DES, MD4, MD5, and RC4. Other algorithms should be implemented directly in software using the main core.

By using the provided mbedTLS API, the developer does not have to worry about software versus hardware acceleration. Furthermore, algorithms may be futher accelerated in future releases of the library.

3. CRYPTO Demo Application

The [**Crypto**] demo application highlights the use of the mbedTLS library and the CRYPTO accelerator in the EFM32 Pearl Gecko family. The demo is loosely based on the [**SLSTK3401_aescrypt**] demo accessible using the [**Demos**] tile and uses the same security algorithm, but is implemented using the Cipher Block Chaining (CBC) model instead of the Electronic Code Book (ECB) model. This mbedtls_aescrypt example is based on the aescrypt2 example included in mbedtls-2.1.0.

The messages to be encrypted are generated at runtime as a consecutive sequence of values either 64, 128 or 256 bytes in length. Each time the BTN0 button on the STK is pressed, one of the messages is encrypted, and the resulting Initialization Vector (IV), cipher text, and message digest are created and stored in the user data area in the flash. Pressing BTN1 reads the stored IV, cipher text, and message digest from the flash and decrypts the data. The status and the results of the application are shown on the memory LCD display. The display shows the length of the message being generated and the clock cycle results of the encryption and decryption sequences.

The example application uses the mbedTLS Application Programmer Interface (API) to implement the AES algorithms as part of a CBC block cipher. The example additionally uses SHA256 to hash the IV, key, and hash message authentication code (HMAC). The Silicon Labs mbedTLS implementation is based on the ARM mbedTLS online library. The difference between these libraries is that the Silicon Labs mbedTLS low-level library functions take advantage of the integrated dedicated hardware for the complex algorithms to accelerate the operations and offload the CPU.

Refer to the ARM mbed website for API module, data structure, and file definitions (https://www.mbed.com/en/development/software/mbed-tls/).

4. Simplicity Studio Installation Structure

The mbedTLS library is included in the Simplicity Studio installation. The files can be found in the following directory by default:

```
C:\SiliconLabs\SimplicityStudio\v3\developer\sdks\efm32\v2\reptile\mbedtls
```

Included in the installation are all of the source files for the individual crypto algorithms such AES, SHA256, and many others. An additional directory is provided in the installation called `sl_crypto` that contains all of the Silicon Labs mbedTLS functions that take advantage of the CRYTPO module. These functions are called in place of the standard mbedTLS functions.

The mbed examples can be found in the kit installation paths. For example, the path to all EFM32 Pearl Gecko examples may be:

```
C:\SiliconLabs\SimplicityStudio\v3\developer\sdks\efm32\v2\kits\SLSTK3401A_EFM32PG\examples
```

Several demos and examples are provided to illustrate the use of the CRYPTO module. The [**mbedtls_aescrypt**] example is a symmetric key cryptography example that illustrates the use of the AES and SHA256 algorithms as a single block cipher. It creates the CipherText from a PlainText input and restores the PlainText from the CipherText. The [**mbedtls_esdsa**] (eliptic curve digital signature) is an asymmetric key example illustrating public key encryption.

The example provided with this application note is based on the [**mbedtls_aescrypt**] example.

5. Using mbedTLS

5.1 Include Files

The ARM mbedTLS library includes many of the algorithms required for cryptography applications. The mbedTLS library files use the CPU core to implement the cryptography algorithms. When doing cryptography, there are large data widths and complex processing required when performing the calculations that consume CPU resources, such as a 256-bit private key. The mbedTLS library includes the capability to use other cryptography implementations. The `sl_crypto` directory includes many cryptography functions and enabled by the use of #defines that configure the project build.

All of the alternate configuration settings are set in the `config-aescrypt.h` file. The config file can be found from the project properties ([**Properties**]>[**C/C+ Build**]>[**Settings**]>[**GNU ARM C Compiler**]>[**Symbols**]). This file redirects the build to use the Silicon Labs source files. The project can be built using the standard ARM mbedTLS APIs by changing the name of this definition. For example, changing the variable to [**zzMBEDTLS_CONFIG_FILE="config-aescrypt.h"**] will remove the Silicon Labs alternate functions from the build.

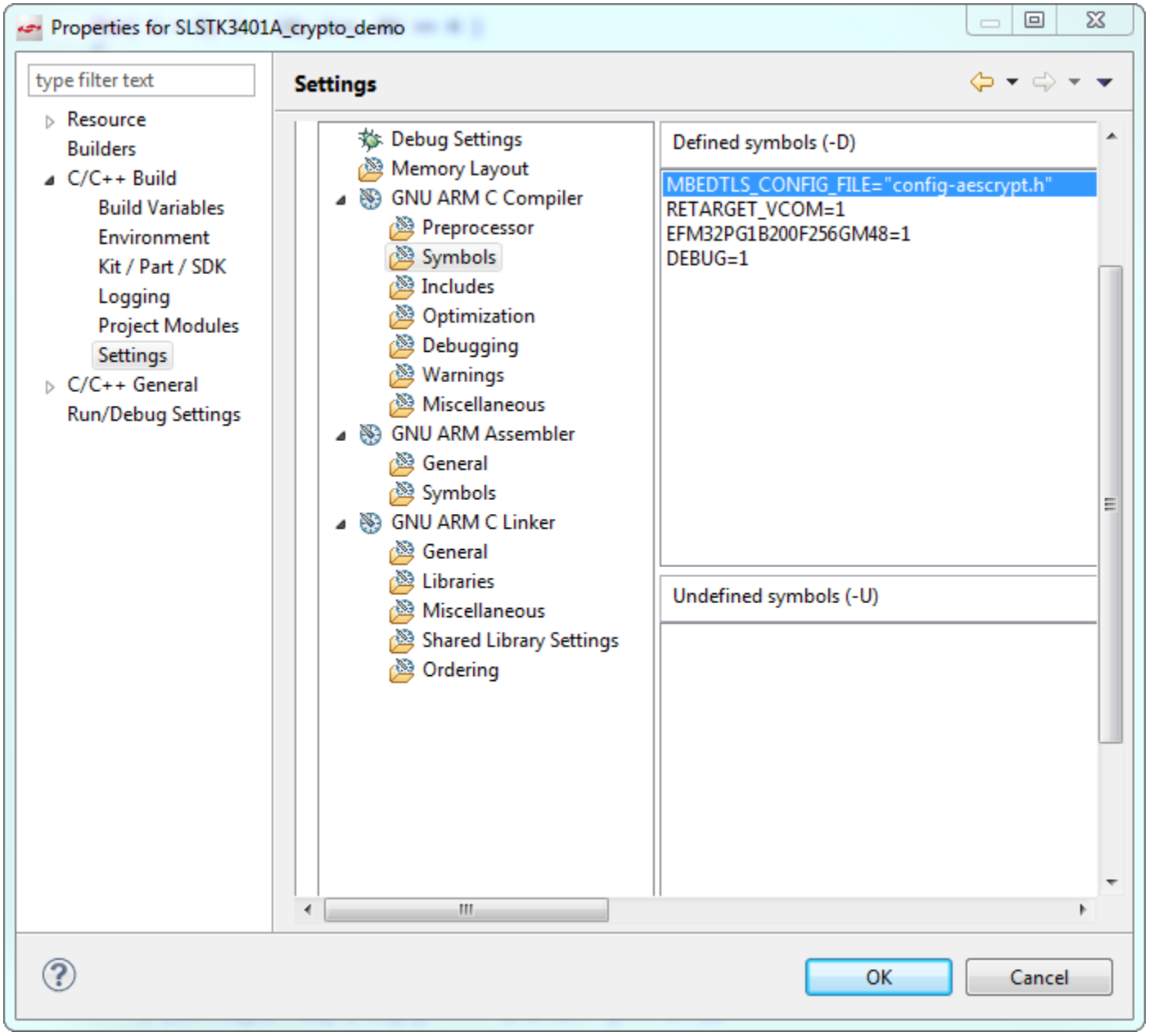


Figure 5.1. Defining the Configuration File

There are alternate defines within the `config_aescrypt.h` file for each of the Silicon Labs files the project will use instead of the original ARM mbedtls files. The mbedTLS library uses two levels of alternate implementations:

1. Per module (header file).
2. Per function.

Silicon Labs provides alternative implementations for the AES and SHA modules (`sl_aes` and `sl_sha1/sl_sha256`), and some alternative implementations of ECC functions defined in `sl_ecp.c`.

When the [**_ALT**] definition for one of the library files is used, the ARM mbedTLS library function will remove its own functions from the build. For example, the AES functions included in the `sl_crypto` library in the `sl_aes.c` file are used instead of the ARM AES functions found in `aes.c` by defining [**MBEDTLS_AES_ALT**] in the configuration file. By defining the alternate function, the AES functions in the mbedTLS c files are removed from the build.

```
/*
 * Minimal configuration for AES test on SiliconLabs devices incorporating
 * CRYPTO hardware accelerator.
 *
 */

#ifndef MBEDTLS_CONFIG_H
#define MBEDTLS_CONFIG_H

#if !defined( NO_CRYPTO_ACCELERATION )
/* SiliconLabs plugins with CRYPTO acceleration support. */
#define MBEDTLS_AES_ALT
#define MBEDTLS_SHA1_ALT
#define MBEDTLS_SHA256_ALT
#define MBEDTLS_SHA256_PROCESS_ALT
#define MBEDTLS_ECP_DEVICE_ALT
#define MBEDTLS_ECP_DOUBLE_JAC_ALT
#define MBEDTLS_ECP_DEVICE_ADD_MIXED_ALT
#define MBEDTLS_ECP_NORMALIZE_JAC_ALT
#define MBEDTLS_ECP_NORMALIZE_JAC_MANY_ALT
#define MBEDTLS_MPI_MODULAR_DIVISION_ALT
#endif

/* mbed TLS modules */
#define MBEDTLS_AES_C
#define MBEDTLS_CIPHER_MODE_CBC
#define MBEDTLS_CIPHER_MODE_CFB
#define MBEDTLS_CIPHER_MODE_CTR
#define MBEDTLS_SHA256_C
#define MBEDTLS_MD_C

#include "mbedtls/check_config.h"

#endif /* MBEDTLS_CONFIG_H */

```

Figure 5.2. Switching to the Silicon Labs Hardware-Accelerated Libraries (`config-aescrypt.h`)

5.2 Project Structure

The project uses the mbedTLS libraries and the Silicon Labs libraries. As previously mentioned, each of these libraries are included as source code and is part of the Simplicity Studio installation. By default, all of the library files are linked from the main source location, so modifying any of these files modifies the library source files. It should not be required to modify any of these files, but if modifications are necessary, then the option to copy the source files to the project location should be used.

Figure 5.3. Project Data Structure

The figure above shows that data structure of the project. The mbedTLS directory includes all of the source files provided by ARM to implement the cryptographic library.

- The `sl_crypto` modules include all of the alternate Silicon Labs source code that replaces the ARM versions and utilizes the CRYPTO module.
- The CMSIS source includes the start-up files for the Pearl Gecko MCU.
- The drivers file includes support functionality for the board and the example, like the display drivers.

5.3 Code

5.3.1 Overview

The easiest method to start on a cryptographic project is to start with one of the example projects provided. These examples provide insight to the use of the libraries and illustrates the use of many of the API functions. Seeing the usage is beneficial to understanding the declarations and how to use the resultant data in an application.

The example implements CBC using the mbedTLS APIs. All of the iterative functions required to process large data arrays for long block ciphers are handled by the API functions. This includes segmenting the data into smaller blocks for the AES block sizes, as well as implementing the individual steps of the CBC cipher mode XOR function and encryption.

The code generates a hashed initialization vector from a pass-phrase defined as part of the build. The IV is used to start the cipher in the initial XOR stage. Next, the key is generated as a hash of the secret key with the IV 8192 times to create the key input to the encryption module and set up the AES and HMAC contexts. Once the IV and the key are generated, the cipher can begin and follows the CBC flow shown below.

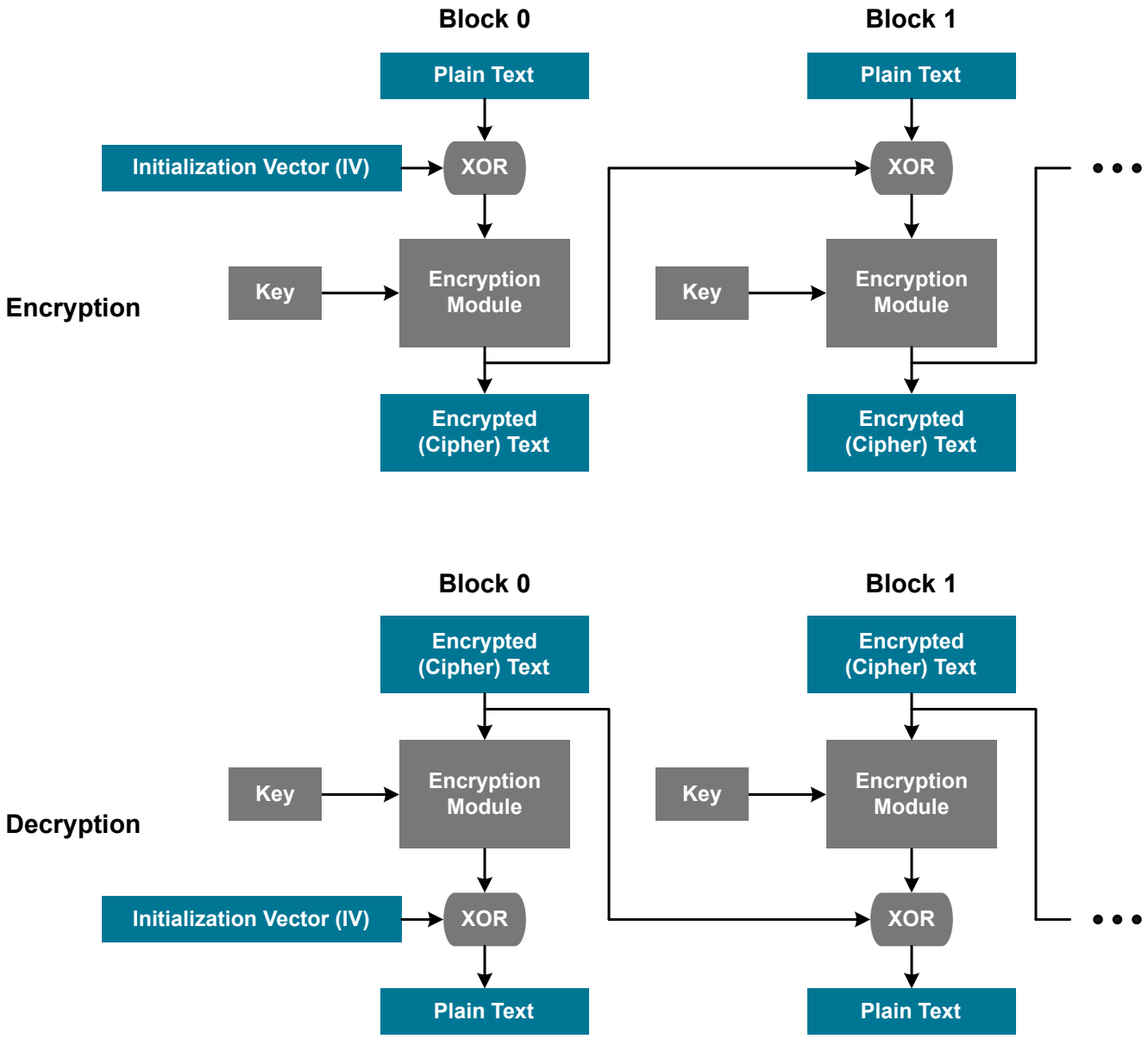


Figure 5.4. Cipher Block Chaining (CBC)

There are several definitions provided to define the sizes of the data blocks for these functions.

```
#define AES_BLOCK_SIZE              (16)
#define IV_SIZE                     (16)
#define TAG_SIZE                    (32)
```

The AES block size is fixed to 16 bytes (`AES_BLOCK_SIZE`) by the standard. The initialization vector is defined to be the same size as the AES block size, which is 16 bytes (`IV_SIZE`). The digest size is defined by `TAG_SIZE` and is set to 32 bytes.

5.3.2 Variable Declaration and Structure Definitions

There are two structures used by the library to allocate memory when using the AES and SHA256. These structures are passed to the API functions for use in the algorithm sequence. The AES structure contains the key size and the key itself and is shown below. When using the AES structure, the APIs will provide the provision for adding a buffer allocated to the input and output data.

```
/**
 * \brief          AES context structure
 */
typedef struct
{
   unsignedint keybits ;       /*!<   size of key */
   uint8_t     key [32];       /*!<   AES key 128 or 256 bits */
}
mbedtls_aes_context ;
```

The structure for the message digest is more complex than the AES structure, as it is composed of structures within a structure. The message digest contains pointers for the different contexts as well as overall information on the operation of the processing for the message digest. The context pointers denoted `md_ctx` and `hmac_ctx` are used to provide status information on the state of the digest, as well as buffer the data for the block being processed. The message digest structure is shown below.

```
/**
 * Generic message digest context.
 */
typedefstruct
{  /** Information about the associated message digest */
   const mbedtls_md_info_t * md_info ;

   /** Digest-specific context */
   void * md_ctx ;

   /** HMAC part of the context */
   void * hmac_ctx ;
}
mbedtls_md_context_t ;
```

An example SHA256 `md_ctx` structure is:

```
/**
 * \brief          SHA-256 context structure
 */
typedefstruct
{
   uint32_t total [2];          /*!< number of bytes processed   */
   unsignedchar buffer [64];    /*!< data block being processed */
   int is224 ;                  /*!< 0=> SHA-256, else SHA-224 */
   uint32_t state [8];          /*!< intermediate digest state   */
}
mbedtls_sha256_context ;
```

The structures defined above are used throughout the library as the main memory area for the processing of the crypto algorithms. Some of the structures are defined explicitly, while others are defined through the parameter passing as part of the function calls.

5.3.3 Using the APIs

The online documentation for the ARM mbedTLS libraries cover the low level details of the APIs. This section covers the use of the APIs in the sample application to show one way of putting all of the pieces together to form usable crypto algorithms. The process shown below is only a single example and not the only method that can be used. The flexibility of the libraries enable many different algorithms and cipher block modes.

Below is the code flow for the SHA256 section of the [**Crypto**] demo code. This example is not a complete application, but provides insight to the overall flow of the mbedTLS libraries.

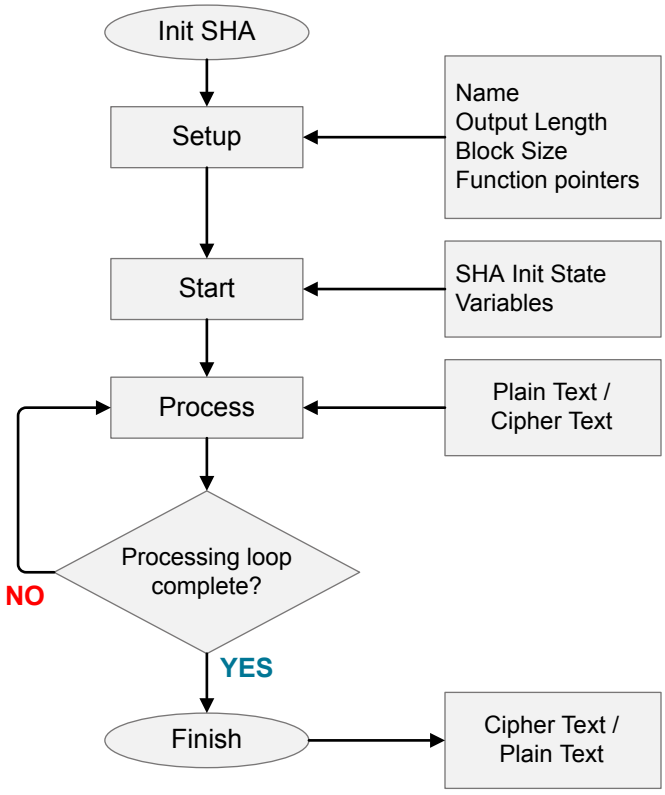


Figure 5.5. Crypto Demo Flow Diagram

The application starts with the initialization of the SHA contexts, which zeros the memory allocated for structure `sha_ctx`.

```
mbedtls_md_init( &sha_ctx );
```

In an attempt to make the library more flexible, mbedTLS uses function pointers for all of the processes. The setup API attaches the different functions to the respective function pointers using the message digest information structure shown below. Each of the function pointer entries in the structure require an associated function address to be assigned by the setup API. The setup function attaches the function pointers based on the second argument of the function.

```
ret = mbedtls_md_setup( &sha_ctx, mbedtls_md_info_from_type(MBEDTLS_MD_SHA256 ), 1);

structmbedtls_md_info_t
{
   /** Digest identifier */
   mbedtls_md_type_t type ;

   /** Name of the message digest */
   constchar * name ;

   /** Output length of the digest function in bytes */
   int size ;

   /** Block length of the digest function in bytes */
   int block_size ;
```

```
   /** Digest initialisation function */
   void (* starts_func )( void *ctx );

   /** Digest update function */
   void (* update_func )( void *ctx, const unsigned char *input, size_t ilen );

   /** Digest finalisation function */
   void (* finish_func )( void *ctx, unsigned char *output );

   /** Generic digest function */
   void (* digest_func )( constunsignedchar *input, size_t ilen, unsigned char *output );

   /** Allocate a new context */
   void * (* ctx_alloc_func )( void );

   /** Free the given context */
   void (* ctx_free_func )( void *ctx );

   /** Clone state from a context */
   void (* clone_func )( void *dst, const void *src );

   /** Internal use only */
   void (* process_func )( void *ctx, const unsigned char *input );
};
```

The mbedTLS library includes wrapper functions for each of the processes defined in the flow diagram. These wrapper functions are called when the high level API is included in the firmware, such as the start or update APIs. The structure below is included in the library and contains the function names to be to the function pointers.

```
const mbedtls_md_info_t mbedtls_sha256_info = {
   MBEDTLS_MD_SHA256,
   "SHA256" ,
   32,
   64,
   sha256_starts_wrap,
   sha224_update_wrap,
   sha224_finish_wrap,
   sha256_wrap,
   sha224_ctx_alloc,
   sha224_ctx_free,
   sha224_clone_wrap,
   sha224_process_wrap,
};
```

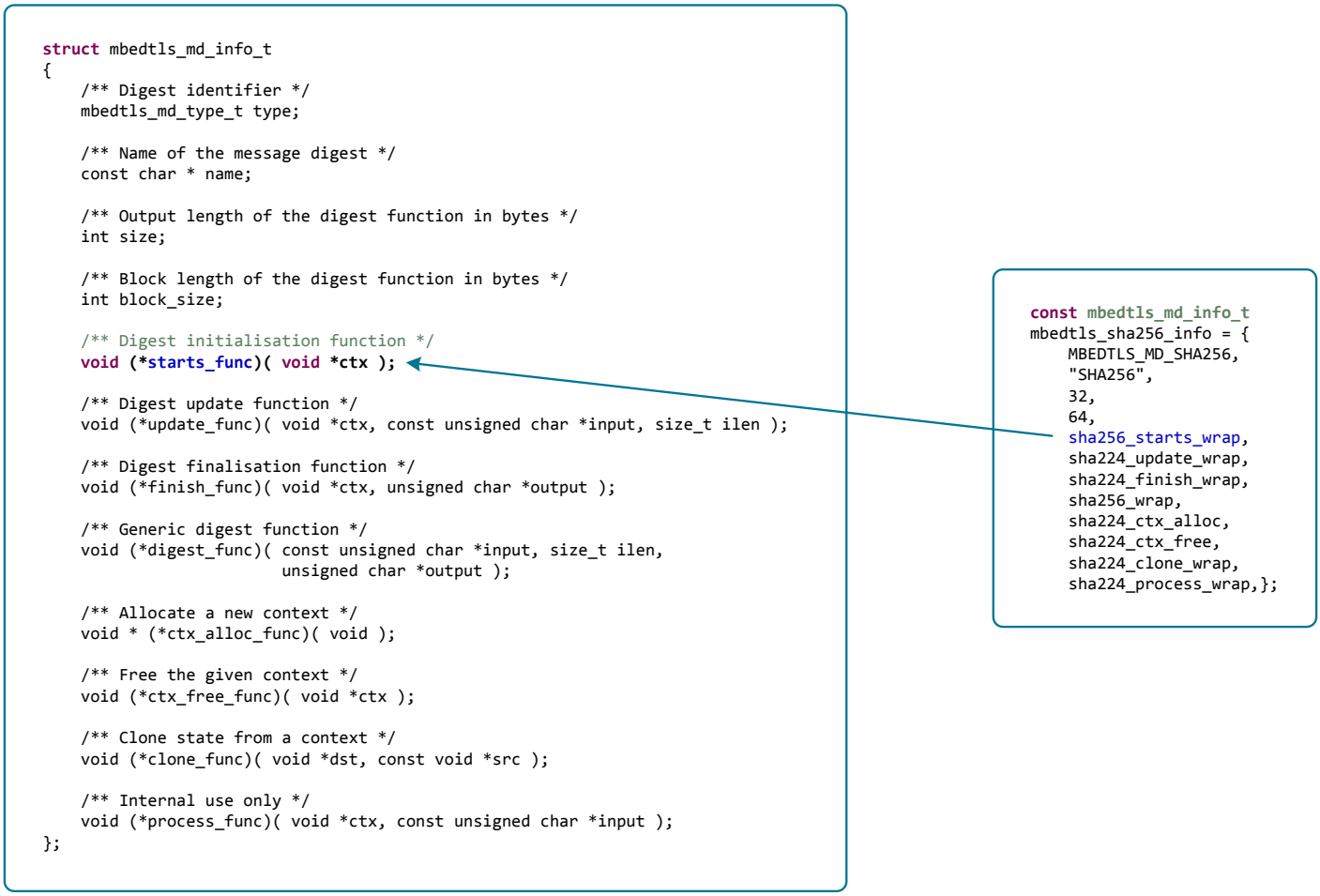


Figure 5.6. Correlating the Two Structures

The processing for the algorithms can begin once the structures have been allocated and the functions have been assigned. For the SHA algorithm, the initial vector is created. This is done via the start function which takes the initial values and stores them into the contexts intermediate state variables.

```
mbedtls_md_starts( &sha_ctx );
```

The first processing of the digest starts with the update function. Within the update function, the `sha256_process` function is called, which outputs the first hash of the input data.

```
mbedtls_md_update( &sha_ctx, buffer, 8 );
```

There are many other API functions that can be used to implement different algorithms. These functions are not discussed in this application note, so refer to the mbedTLS documentation from ARM.

5.3.4 Comparison of Silicon Labs and Standard mbedTLS performance

The Silicon Labs mbedTLS functions use the CRYPTO module to offload the processing from the CPU. The CRYPTO module is a high-performance peripheral capable of handling complex algorithms and large data widths (up to 256 bits) that reduces much of the overhead of the CPU. An example of the difference in the functions and a comparison is shown below. Pre-compiled hex files are available in the application note package for both versions and can be run on the Pearl Gecko STK.

Table 5.1. Demo Application Programming Files

Filename	Description
SLSTK3401A_crypto_cbc_demo.hex	Demo application compiled using the HW accelerator to perform all of the cryptography functions using the Silicon Labs library. CBC cipher mode executed via dedicated API.
SLSTK3401A_no_crypto_cbc_demo.hex	Demo application compiled without using the HW accelerator. This build uses the mbedTLS library as downloaded from ARM. CBC cipher mode executed via dedicated API.
SLSTK3401A_crypto_demo.hex	Demo application compiled using the HW accelerator to perform all of the cryptography functions using the Silicon Labs library. CBC cipher mode executed using XOR and ECB cipher mode.
SLSTK3401A_no_crypto_demo.hex	Demo application compiled without using the HW accelerator. This build uses the mbedTLS library as downloaded from ARM. CBC cipher mode executed using XOR and ECB cipher mode.

The first set of figures below are from the mbedTLS library, and the second is the Silicon Labs function that uses the CRYPTO module. Note that the functionality of the two process functions are the same.

Table 5.2. Comparing the Standard mbedTLS and Silicon Labs mbedTLS Implementations

Operation	Standard mbedTLS Crypto Example (cycles)	Silicon Labs mbedTLS Crypto Example (cycles)	Improvement
8192 Hash Iterations	233193519	28926011	8x
Encrypt			
Setkey Encode	2486	346	7x
Encrypt IV Hash Update	16736	3916	4x
CBC Encrypt	30126	2418	12x
HMAC Update	26214	1460	17x
HMAC Finish	42103	4623	9x
Total	117665	12763	9x
Decrypt			
Setkey Decode	13061	348	37x
Encrypt IV Hash Update	16739	3912	4x
HMAC Update	26217	1453	18x
CBC Decrypt	31351	3480	9x
HMAC Finish	42114	4632	9x
Total	129482	13825	9x

```
#if !defined(MBEDTLS_SHA256_PROCESS_ALT)
static const uint32_t K[] =
{
    0x428A2F98, 0x71374491, 0xB5C0FBCF, 0xE9B5DBA5,
    0x3956C25B, 0x59F111F1, 0x923F82A4, 0xAB1C5ED5,
    0xD807AA98, 0x12835B01, 0x243185BE, 0x550C7DC3,
    0x72BE5D74, 0x80DEB1FE, 0x9BDC06A7, 0xC19BF174,
    0xE49B69C1, 0xEFBE4786, 0x0FC19DC6, 0x240CA1CC,
    0x2DE92C6F, 0x4A7484AA, 0x5CB0A9DC, 0x76F988DA,
    0x983E5152, 0xA831C66D, 0xB00327C8, 0xBF597FC7,
    0xC6E00BF3, 0xD5A79147, 0x06CA6351, 0x14292967,
    0x27B70A85, 0x2E1B2138, 0x4D2C6DFC, 0x53380D13,
    0x650A7354, 0x766A0ABB, 0x81C2C92E, 0x92722C85,
    0xA2BFE8A1, 0xA81A664B, 0xC24B8B70, 0xC76C51A3,
    0xD192E819, 0xD6990624, 0xF40E3585, 0x106AA070,
    0x19A4C116, 0x1E376C08, 0x2748774C, 0x34B0BCB5,
    0x391C0CB3, 0x4ED8AA4A, 0x5B9CCA4F, 0x682E6FF3,
    0x748F82EE, 0x78A5636F, 0x84C87814, 0x8CC70208,
    0x90BEFFFA, 0xA4506CEB, 0xBEF9A3F7, 0xC67178F2,
};

#define  SHR(x,n) ((x & 0xFFFFFFFF) >> n)
#define ROTR(x,n) (SHR(x,n) | (x << (32 - n)))

#define S0(x) (ROTR(x, 7) ^ ROTR(x,18) ^  SHR(x, 3))
#define S1(x) (ROTR(x,17) ^ ROTR(x,19) ^  SHR(x,10))

#define S2(x) (ROTR(x, 2) ^ ROTR(x,13) ^ ROTR(x,22))
#define S3(x) (ROTR(x, 6) ^ ROTR(x,11) ^ ROTR(x,25))

#define F0(x,y,z) ((x & y) | (z & (x | y)))
#define F1(x,y,z) (z ^ (x & (y ^ z)))

#define R(t)                                    \
(                                               \
    W[t] = S1(W[t -  2]) + W[t -  7] +          \
           S0(W[t - 15]) + W[t - 16]            \
)

#define P(a,b,c,d,e,f,g,h,x,K)                  \
{                                               \
    temp1 = h + S3(e) + F1(e,f,g) + K + x;      \
    temp2 = S2(a) + F0(a,b,c);                  \
    d += temp1; h = temp1 + temp2;              \
}
```

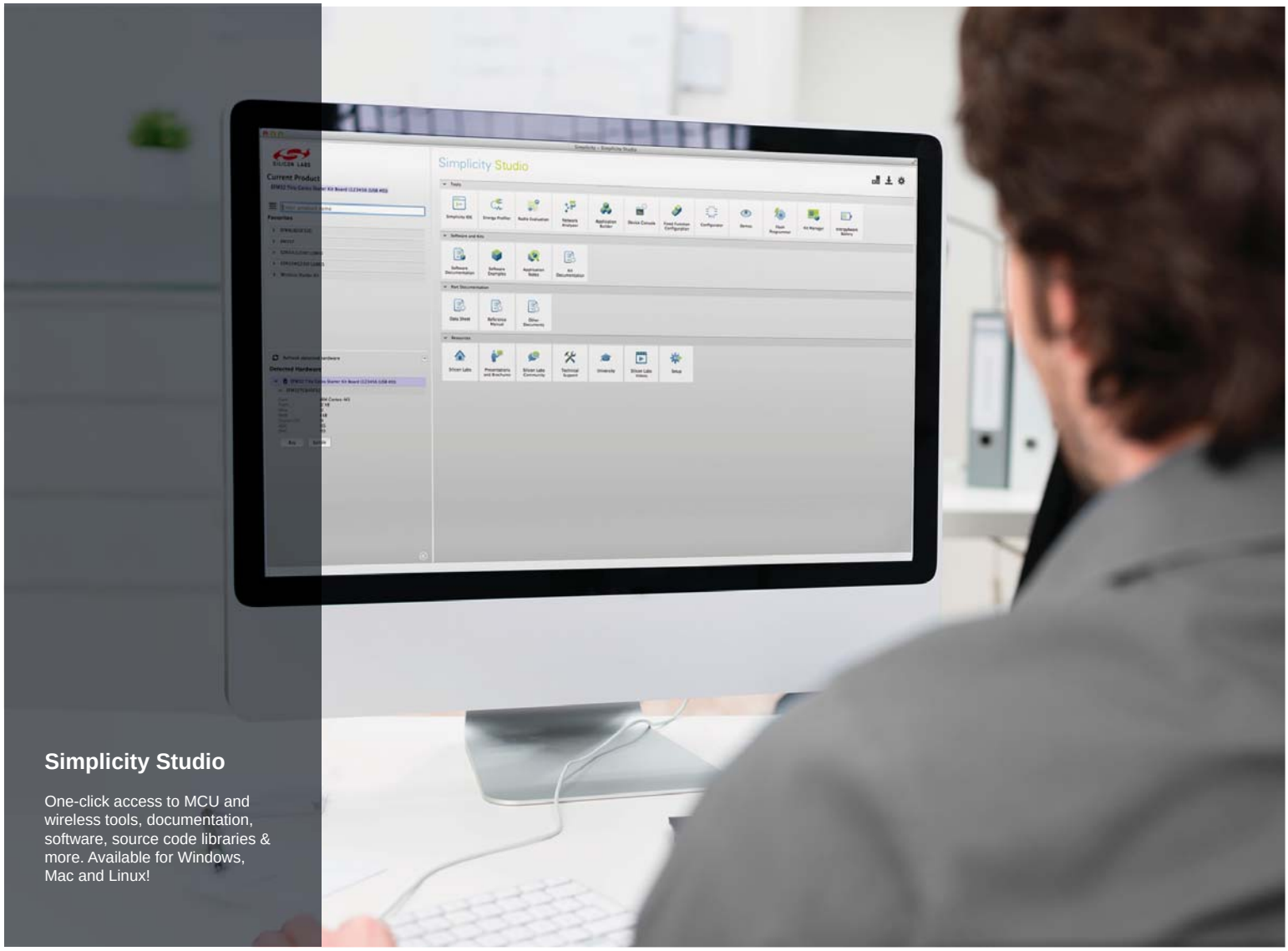
Figure 5.7. Standard mbedTLS Function — Part 1

```
178
179 void mbedtls_sha256_process( mbedtls_sha256_context *ctx, const unsigned char data[64] )
180 {
181     uint32_t temp1, temp2, W[64];
182     uint32_t A[8];
183     unsigned int i;
184
185     for( i = 0; i < 8; i++ )
186         A[i] = ctx->state[i];
187
188 #if defined(MBEDTLS_SHA256_SMALLER)
189     for( i = 0; i < 64; i++ )
190     {
191         if( i < 16 )
192             GET_UINT32_BE( W[i], data, 4 * i );
193         else
194             R( i );
195
196         P( A[0], A[1], A[2], A[3], A[4], A[5], A[6], A[7], W[i], K[i] );
197
198         temp1 = A[7]; A[7] = A[6]; A[6] = A[5]; A[5] = A[4]; A[4] = A[3];
199         A[3] = A[2]; A[2] = A[1]; A[1] = A[0]; A[0] = temp1;
200     }
201 #else /* MBEDTLS_SHA256_SMALLER */
202     for( i = 0; i < 16; i++ )
203         GET_UINT32_BE( W[i], data, 4 * i );
204
205     for( i = 0; i < 16; i += 8 )
206     {
207         P( A[0], A[1], A[2], A[3], A[4], A[5], A[6], A[7], W[i+0], K[i+0] );
208         P( A[7], A[0], A[1], A[2], A[3], A[4], A[5], A[6], W[i+1], K[i+1] );
209         P( A[6], A[7], A[0], A[1], A[2], A[3], A[4], A[5], W[i+2], K[i+2] );
210         P( A[5], A[6], A[7], A[0], A[1], A[2], A[3], A[4], W[i+3], K[i+3] );
211         P( A[4], A[5], A[6], A[7], A[0], A[1], A[2], A[3], W[i+4], K[i+4] );
212         P( A[3], A[4], A[5], A[6], A[7], A[0], A[1], A[2], W[i+5], K[i+5] );
213         P( A[2], A[3], A[4], A[5], A[6], A[7], A[0], A[1], W[i+6], K[i+6] );
214         P( A[1], A[2], A[3], A[4], A[5], A[6], A[7], A[0], W[i+7], K[i+7] );
215     }
216
217     for( i = 16; i < 64; i += 8 )
218     {
219         P( A[0], A[1], A[2], A[3], A[4], A[5], A[6], A[7], R(i+0), K[i+0] );
220         P( A[7], A[0], A[1], A[2], A[3], A[4], A[5], A[6], R(i+1), K[i+1] );
221         P( A[6], A[7], A[0], A[1], A[2], A[3], A[4], A[5], R(i+2), K[i+2] );
222         P( A[5], A[6], A[7], A[0], A[1], A[2], A[3], A[4], R(i+3), K[i+3] );
223         P( A[4], A[5], A[6], A[7], A[0], A[1], A[2], A[3], R(i+4), K[i+4] );
224         P( A[3], A[4], A[5], A[6], A[7], A[0], A[1], A[2], R(i+5), K[i+5] );
225         P( A[2], A[3], A[4], A[5], A[6], A[7], A[0], A[1], R(i+6), K[i+6] );
226         P( A[1], A[2], A[3], A[4], A[5], A[6], A[7], A[0], R(i+7), K[i+7] );
227     }
228 #endif /* MBEDTLS_SHA256_SMALLER */
229
230     for( i = 0; i < 8; i++ )
231         ctx->state[i] += A[i];
232 }
233 #endif /* !MBEDTLS_SHA256_PROCESS_ALT */
```

Figure 5.8. Standard mbedTLS Function — Part 2

```
123 void mbedtls_sha256_process( mbedtls_sha256_context *ctx, const unsigned char data[64] )
124 {
125     CRYPTO_CLOCK_ENABLE;
126
127     /* Setup crypto module to do SHA-2. */
128     CRYPTO->CTRL = CRYPTO_CTRL_SHA_SHA2 |
129       /* Set DMA0 source to DDATA0 and transfer mode */
130       CRYPTO_CTRL_DMA0RSEL_DDATA0    | CRYPTO_CTRL_DMA0MODE_FULL |
131       /* Set DMA1 source to QDATA1BIG and transfer mode */
132       CRYPTO_CTRL_DMA1RSEL_QDATA1BIG | CRYPTO_CTRL_DMA1MODE_FULL;
133
134     /* Set result width of MADD32 operation. */
135     CRYPTO_ResultWidthSet(cryptoResult256Bits);
136
137     /* Set sequence control registers */
138     CRYPTO->SEQCTRL  = 16 & _CRYPTO_SEQCTRL_LENGTHA_MASK;
139     CRYPTO->SEQCTRLB = 0;
140
141     /* Initiate SHA instruction sequence. */
142     CRYPTO_EXECUTE_6( CRYPTO_CMD_INSTR_DMA0TODATA,
143                       CRYPTO_CMD_INSTR_DDATA0TODDATA1,
144                       CRYPTO_CMD_INSTR_SELDDATA0DDATA1,
145                       CRYPTO_CMD_INSTR_DMA1TODATA,
146                       CRYPTO_CMD_INSTR_SHA,
147                       CRYPTO_CMD_INSTR_MADD32);
148
149     /* Write the state to crypto */
150     CRYPTO_DDataWrite(cryptoRegDDATA0, ctx->state);
151
152     /* Write block to QDATA1.  */
153     CRYPTO_QDataWrite(cryptoRegQDATA1BIG, (uint32_t*) data);
154
155     /* Read the state from crypto. */
156     CRYPTO_DDataRead(cryptoRegDDATA0, ctx->state);
157
158     CRYPTO_CLOCK_DISABLE;
159 }
```

Figure 5.9. Example Silicon Labs Hardware Accelerated mbedTLS Function



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com