ADVANCED ENCRYPTION STANDARD

RELEVANT DEVICES

All Silicon Labs MCUs.

1. Introduction

The Advanced Encryption Standard (AES) is an algorithm used to encrypt and decrypt data for the purposes of protecting the data when it is transmitted electronically. The AES algorithm allows for the use of cipher keys that are 128, 192, or 256 bits long to protect data in 16-byte blocks.

AES is a U.S. Federal Information Processing Standards approved algorithm that is also approved for commercial and private applications. Since its acceptance in 2001, AES has become widely used in a variety of applications.

The AES algorithm is a reduced version of the Rijndael algorithm, though the names are sometimes used interchangeably. The Rijndael algorithm allows for additional key sizes and data sizes that are not supported by AES.

The purpose of this application note is to provide a sample implementation of the AES algorithm for Silicon Labs microcontrollers and to detail the performance of the implementation. The provided example code is intended for C8051F326/7 devices, but, since the code is not hardware-specific, it can easily be ported to any Silicon Labs microcontroller.

This application note does not describe the mathematics used in the algorithm. An explanation of the mathematics, along with other information about AES, is available in the official AES document provided by the National Institute of Standards and Technology, FIPS PUB 197 (available at http://csrc.nist.gov/publications/fips/).

1.1. Potential Applications

Since the minimum key size of AES is 128-bits, it is considered to be immune to brute force attacks for the near future. Given the strength of the cipher, implementing AES requires relatively few resources in terms of memory and system cycles, which makes it a good choice for an encryption algorithm. Some sample applications where AES is useful are:

- Wireless communication, such as wireless keyboards
- Point-of-sale terminals
- Surveillance applications

2. Implementation

The AES algorithm is a symmetric-key algorithm. A symmetric-key algorithm uses the same or related keys to encrypt and decrypt the data. In the AES algorithm, the input data is 16 bytes, and the resulting encrypted data is also 16 bytes. The encryption and decryption routines use the same private key that is 128, 192, or 256 bits. The larger the key size used, the more difficult it is to break the algorithm and obtain the encrypted data.

The example code provided with this application note is a mostly straightforward implementation of the algorithm provided in FIPS PUB 197. In order to maintain easy readability, the example code uses the same terminology and function names provided in the specification. The optimizations used in this example that deviate from the example implementation provided in the specification are described in more detail in "3.2. Optimization" on page 4.

2.1. Firmware Organization

The code is divided into three independent modules: encryption, decryption, and key expansion. The encryption module includes the firmware necessary to convert the input data to cipher text. The decryption module converts cipher text back to plain text or unencrypted data. The key expansion module expands the cipher key into a global array that is used by both the encryption and decryption routines.

If the cipher key is known before the program is compiled, the expanded cipher key can be compiled into the program, and the key expansion module is not required. If the cipher key is only known after the program is compiled, the key expansion routine is required. Table 1 shows which files are common to all modules and which files are module-specific.

Table 1. Firmware Organization

Module	Relevant Files
Common Files	F326_AES_Typedef.h F326_AES_Parameters.h
Encryption	F326_AES_Cipher.c F326_AES_Cipher.h F326_AES_Sbox.h
Decryption	F326_AES_InvCipher.c F326_AES_InvCipher.h
Key Expansion	F326_AES_KeyExpander.c F326_AES_KeyExpander.h F326_AES_Sbox.h

2.2. How to Add AES Functionality to a Project

The first step in adding AES to a project is to determine which components of AES (encryption, decryption, and/or key expansion) are required. Add the appropriate files from Table 1 to the project. The global declaration of the variable, `EXP_KEYS`, will need to be moved to a common file if F326_AES_KeyExpansion.c is not included in the project.

The second step is to customize two options. The first option is the cipher key length (128, 192, or 256 bits). The cipher key length is defined in F326_AES_Parameters.h using #define `CIPHER_KEY_LENGTH`. The second option is the choice of the cipher key. If the key is known before compile time, the key can be stored in the array, `CIPHER_KEY`, or in F326_AES_KeyExpander.h, or the expanded key can be stored in the array, `EXP_KEYS`.

A cipher key can be selected by choosing any random 128, 192, or 256-bit number. Since the cipher key is not required to have any special properties, such as being a multiple or factor of another number, all keys are equally cryptographically strong. The final step is to call the encryption, decryption, and key expansion routines from the main program using the following functions:

```
void Cipher (byte *in, byte *out);
void InvCipher (byte *in, byte *out);
void KeyExpansion ();
```

`Cipher()` and `InvCipher()` both accept a 16-byte array as the input and also output a 16-byte array. `KeyExpansion()` uses a global array, `CIPHER_KEY`, as the input and outputs the expanded keys to another global array, `EXP_KEYS`.

See F326_AES_Main.c for an example of how to use these functions.

2.3. Porting the Firmware to Other Silicon Labs MCUs

The firmware used to implement the encryption and key expansion routines is fully hardware-independent and uses C code compatible with any Silicon Labs microcontroller without any changes.

The decryption module includes the SFR definition file for the target hardware. The `FFMultiply()` function in F326_AES_Decrypt.c directly references a hardware register to check the carry bit after an addition. This hardware register is defined in C8051F326.h, which is included in F326_AES_Decrypt.c. When using the decryption function on another MCU, change the header file to the one appropriate for the target MCU.

3. Algorithm Performance and Memory Requirements

The following section describes the number of system clock cycles necessary to execute the encryption, decryption, and key expansion routines for the three cipher key sizes. It also lists the amount of RAM, external RAM, and code space required by each module.

The system clock cycles were measured using an on-chip Timer. The firmware used to measure the system clock cycles is included in the example project.

The system clock cycle count and memory requirements were obtained from a project built using the Keil CA-51 Compiler (Version 7.5) using the standard optimization settings.

3.1. Measurements

The cycle count values for encryption or decryption shown in Table 2 indicate the number of system clock cycles required to encrypt or decrypt 16-bytes of data. The cycle count for key expansion indicates the number of system clock cycles to expand the keys. This function will need to be called only once for each cipher key that is used.

Table 2. System Cycle Count and Execution Times for Common System Clock Frequencies

	Module	Cycle Count	CLK = 24 Mhz	CLK = 50 Mhz	CLK = 100 Mhz
128-bit	Encryption	11053	460 µs	221 µs	111 µs
	Decryption	34634	1443 µs	693 µs	346 µs
	Key Expansion	25491	1062 µs	510 µs	255 µs
192-bit	Encryption	12955	540 µs	259 µs	130 µs
	Decryption	41590	1733 µs	832 µs	416 µs
	Key Expansion	29605	1234 µs	592 µs	296 µs
256-bit	Encryption	14857	619 µs	297 µs	149 µs
	Decryption	48609	2025 µs	972 µs	486 µs
	Key Expansion	34158	1423 µs	683 µs	342 µs

The amount of time required to execute one of these routines for any system-clock frequency can easily be calculated by using the following formula:

$$\text{Time(s)} = \frac{\text{Cycle Count}}{\text{System Clock Frequency (Hz)}}$$

Table 3 lists the amount of RAM, external RAM, and code space required by each of the modules.

Table 3. Memory Requirements for the Modules

Module		RAM (bytes)	External RAM (bytes)	Code Space (bytes)
Encryption		43	0	1056
Decryption		48	0	2100
Key Expansion	128-bit	13	352	825
	192-bit	13	416	833
	256-bit	13	480	841

The memory requirement for the cipher key and expanded key are included with the key expansion numbers. The memory requirement for the `Sbox[]` array is included with the encryption numbers. See “3.2. Optimization” for more information about reducing the memory requirements of the algorithm.

3.2. Optimization

The example code provided with this application note is a modular version of the algorithm presented in the AES specification. The encryption and decryption routines keep the same functional structure and organization. The general differences are that some loops are unrolled for speed, and most of the data is passed through global variables. The following sections describe specific choices made for the example code and provide alternate implementation options.

3.2.1. Dynamic Key Expansion

The AES algorithm for both encryption and decryption is divided into multiple rounds, which is a function of the size of the cipher key. During each of these rounds, an operation is performed using one row of the expanded keys. In the provided example, the full set of expanded keys is stored in an array in the first page of external RAM to optimize access. For a 128-bit cipher key, this array is 176 bytes. If the external RAM space is limited, the expanded keys can instead be generated dynamically 16 bytes at a time. This helps save external RAM at the cost of additional system cycles.

3.2.2. Finite Field Multiply

The AES algorithm performs its calculations using finite field mathematics, which is described in more detail in the official specification. The finite field multiply operation is used in the `MixColumns()` and `InvMixColumns()` functions of encryption and decryption and can be implemented in various ways. Optimizing this operation is important because it is performed 576 times when using a 128-bit cipher key. Replacing the finite field multiply with the `xtime()` function for the encryption process greatly reduces the required system clock cycles. The `xtime()` function is an optimization on the finite field multiply operation, which takes advantage of the limited range of operands when performing an encryption. Since the range of operands used in the finite field multiply during decryption is larger, the `xtime()` function is not as efficient for decryption, and a different solution must be used.

The current example uses a log table and an exponentiation table to perform the finite field multiplications during decryption at the expense of code space. The additional log table and exponentiation table, defined in F326_AES_InvCipher.h, require an additional 512 bytes of code space.

An alternate implementation option is to perform the multiply using the algorithm provided in the specification. This algorithm does not require additional code space for the lookup tables but requires many more system clock cycles.

3.2.3. Combining Encryption and Decryption Routines

In the example code, the encryption and decryption modules are fully modular. For this reason, both modules include some functions that are the same. If both encryption and decryption are necessary for the target application, the following functions can be shared between the two modules to save code space:

- `StateIn()`
- `StateOut()`
- `AddRoundKey()`
- `LoadKeys()`

3.3. Test Vectors and Intermediate Results

This section includes the same example input vectors from the official specification for the input data and cipher keys. All values are presented in hexadecimal format.

3.3.1. 128-Bit Cipher Key

```
Input Data     : 0x00112233445566778899AABBCCDDEEFF
Cipher Key     : 0x000102030405060708090A0B0C0D0E0F
Encrypted Data : 0x69C4E0D86A7B0430D8CDB78070B4C55A
```

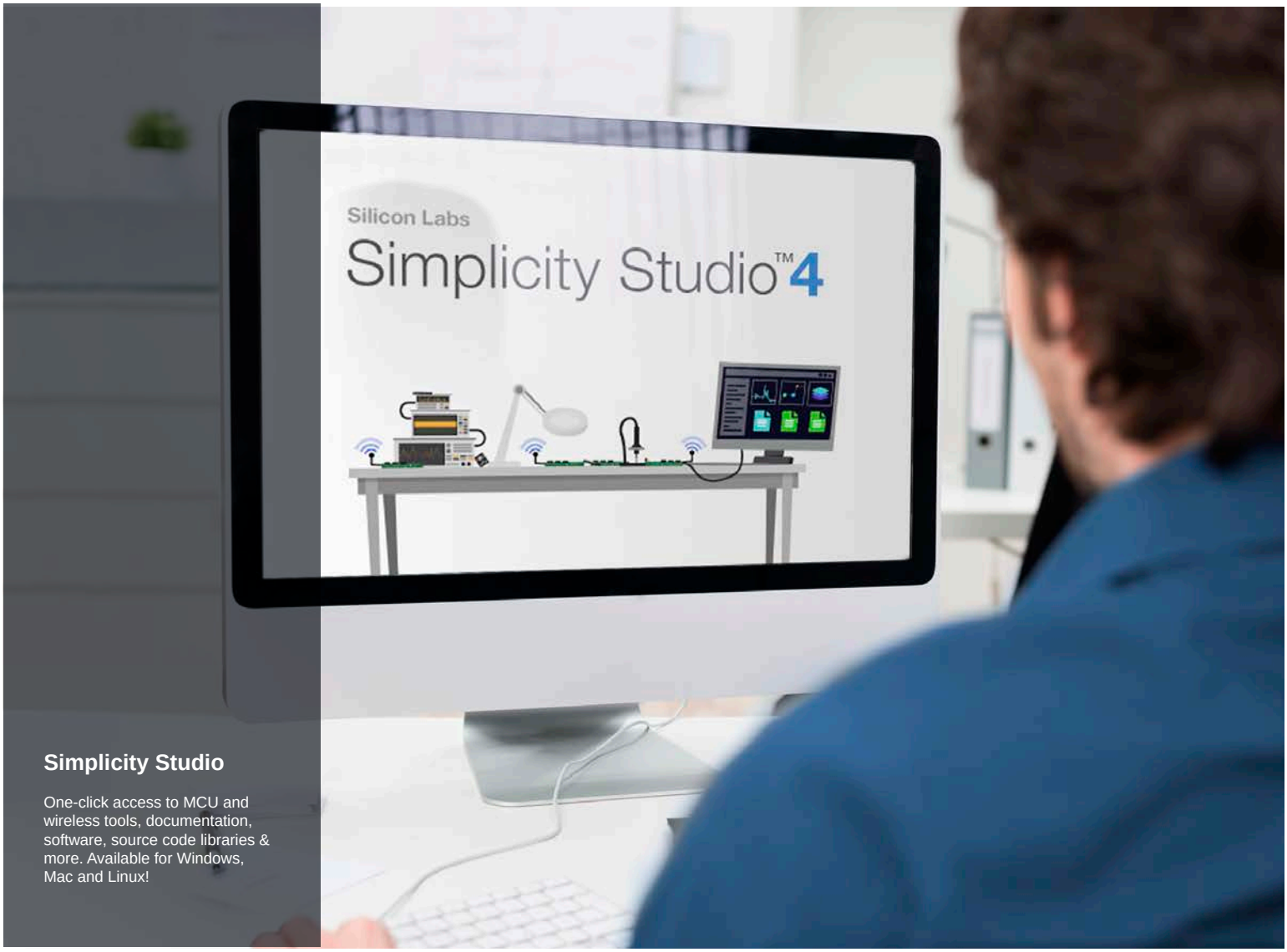
3.3.2. 192-Bit Cipher Key

```
Input Data     : 0x00112233445566778899AABBCCDDEEFF
Cipher Key     : 0x000102030405060708090A0B0C0D0E0F1011121314151617
Encrypted Data : 0xDDA97CA4864CDFE06EAF70A0EC0D7191
```

3.3.3. 256-Bit Cipher Key

```
Input Data     : 0x00112233445566778899AABBCCDDEEFF
Cipher Key     :
0x000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F
Encrypted Data : 0x8EA2B7CA516745BFEAFC49904B496089
```

The proper execution of the encrypt and decrypt routines can be confirmed in the firmware by setting a breakpoint after the `InvCipher()` function, which is called in F326_AES_Main.c. Add the variables, `EncryptedData` and `PlaintextData`, to the IDE watch window and confirm that their values are the same as the ones listed above. `PlaintextData` should be equivalent to Input Data.



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