Using AES Encryption and Decryption with Stellaris® Microcontrollers

Application Note

AN01251-03

Copyright

Texas Instruments
108 Wild Basin, Suite 350
Austin, TX 78746
http://www.ti.com/stellaris

Table of Contents

Introduction

The Advanced Encryption Standard (AES) is a publicly defined encryption[1] standard used by the U.S. government. It is a strong encryption method with reasonable performance and size, replacing both the Data Encryption Standard (DES) and the Triple Data Encryption Standard (3DES), the latter of which is a set of various incompatible forms of running DES three times to get more than 56 bits of key[2]. AES is fast in both hardware and software, is fairly easy to implement, and requires little memory.

Note: This application note assumes that you have at least a basic to moderate understanding of data encryption and decryption[3] and associated standards. Encryption-related terms are defined in footnotes on the page in which the term first occurs. There are also many sources of information on cryptography, and this application note is not intended to replace those.

- **For applications that need public key exchanging encryption,** see *Evaluating Matrix SSL for Stellaris® Microcontrollers Application Note (AN01244)* available at the www.ti.com/stellaris web site. SSL requires far more code and data and is slower than direct AES, but is important when keys are to be exchanged openly.
- **For applications that can use pre-arranged keys, such as setup during manufacturing or configuration,** this application note explains how to get the smallest footprint solution with good speed and a strong-level of protection.
- **For applications that can set up keys safely across the communications platform, such as during installation or when off a public network,** this application note explains how to avoid risk in the setup process.

The encryption package described in this application note is available with full source code in the StellarisWare® software distribution and is based on a BSD-style open source license. See the LICENSE.txt file on the StellarisWare® CD in the `third_party/aes` directory.

Encryption Method Summary

This application note describes the following four encryption methods used with AES:

- Electronic code book (ECB)
- Cipher block chaining (CBC) or chained encryption
- Cipher feedback (CFB)
- Counter (CTR)

These methods are described in more detail below to help you select the appropriate method(s).

1. *Encryption* is the process used to convert plaintext into a form that is not readable without knowledge of the rules and key to decrypt. AES encrypts blocks of 128 bits (16 bytes) at a time.
2. A *key* is the value string used for both encryption and decryption. AES is symmetric, so the same key is used for both, although the expansion of the key is not. Key sizes are normally 128-bits (16 bytes, but may be 192 or 256 bytes as well).
3. *Decryption* is the process used to convert from an encrypted form back to plaintext, using a key.

Electronic Code Book (ECB)[1]

ECB is the basic AES encryption and decryption method. Each block of 128 bits (4 words) is encrypted or decrypted with the key. The result is the same size and is standalone. That is, you may start six 128-bit frames into the encrypted data and decrypt the block without any problems.

Cipher Block Chaining (CBC)

CBC or chained encryption is a method used on top of ECB. Each plaintext[2] block is XORed with the encrypted previous block before being encrypted. This has three interesting properties:

- No block can be decrypted without decrypting all of the preceding ones. That is, you cannot jump to the sixth block and decrypt it. Likewise, the encryption must start from the first block and follow through to the end. Ordering is implied (strictly speaking, the physical order in memory does not have to be followed, but both the encryption and decryption must follow the same order).
- An initialization vector (see "Selecting an Initialization Vector" on page 10) must be used for the first block to prime the pump (what you XOR with the plaintext of the first block). This may be random, may be session specific, or may be used over and over (the strength is in the key).
- The resultant data loses all resemblance to the original plaintext and appears to be noise (which is good).

Cipher Feedback (CFB)

CFB is also a method used on top of ECB. Each plaintext block is XORed with the encrypted results of the previous block and that is the result block. The first block is XORed with the encrypted initialization vector. This is similar to CBC, and has the same properties.

In addition, this method is of particular interest because CFB decryption uses ECB encryption rather than a separate ECB decryption. This means a smaller code and data footprint and is often faster (since decryption key expansion takes longer).

Counter (CTR)

CTR is a method which XORs the plaintext block with an encryption of a counter and an initialization vector-like value. The value of this method is that you may randomly decrypt (and of course encrypt) any block, as long as you know its position (count), The count is mixed with an upper part initialization vector to make it stronger (else the count would leave too many MSb 0s). The counter does not have to be a strict counter, and you can merge the initialization vector into the upper part, leaving only enough bits for the counter or XORed to the counter. Because there are no standards for either rule, you can only use this when both sides pre-arrange how it will be done.

1. A *code book* uses a series of code values to use as replacements. In traditional code book models, a message is converted by mapping from alphabet letters to numbers or other letters, according to a pattern in the code book. The modern electronic code books map from numbers (usually bytes or larger) to other numbers of the same size using a key. In this case, the code book is known to all, but the key forms the order in which the mappings are done using a series of rounds (conversions), forming a polynomial expression.
2. *Plaintext* is the original text/values to be encrypted; the original message.

Like CFB, this method is of particular interest because CFB decryption uses ECB encryption rather than a separate ECB decryption. This means a smaller code and data footprint and is often faster (since decryption key expansion takes longer).

Summary

Generally, embedded applications select which method to use in advance, so there is no reason to carry unused code. All methods other than ECB require use of an initialization vector (initialization vector), which is discussed in “Selecting an Initialization Vector” on page 10.

ECB and CBC require both an AES encryption and decryption, which includes key expansion and rounds processing. CFB and CTR only require AES encryption.

Source Code Summary and Configuration

The encryption source code package allows you to configure the source so that you have only the features you need. You can customize the following options:

- **Encryption-only, decryption-only, or both**

  Some applications only send secure data; others may only receive; some block forms only require AES encryption. This allows removal of the unused portion of data.

- **Single table versus four tables for rounds[1]**

  Four tables, which are rotated variants of each other, are typically used to achieve best performance. One set of four tables is required for encryption and one set of four tables is required for decryption. Each table is 1 KB, so a total of 8 KB is required for both encryption and decryption.

  A single table can be used instead, thereby significantly reducing flash use with only a slight decrease in performance due to the required shifts. Again, one 1 KB table is required for encryption and one 1 KB table for decryption, for a total of 2 KB. So, 8 KB of flash is needed when using four tables, versus one table and both encrypting and decrypting. The performance of the use of one table is summarized in Table 3 on page 15.

- **Choice of number of encryption methods (ECB, CBC, CFB, or CTR)**

  ECB encrypts each block individually (and is the basis of the other three encryption methods). This means it can be used for random access decryption (a particular block), but is weaker than the other three as some plaintext information may leak through by pattern. This is most striking with pictures and patterned data since it is an in-place transformation. Further, substitution and replay attacks become much easier.

1. *Rounds* are the processing done between the data, the key, and a fixed encoding table.This is usually a set of XORs, using the data and key to index into the table. Each round uses the result to feed into the next round (index and XOR), so multiple rounds are used to strengthen the process. A set of these rounds (passes) creates a polynomial set, which can then be decrypted by reversing the process (XORing the other way). Even someone knowing a particular block’s plaintext cannot determine the key, since many keys could have produced the same one-block conversion. This in turn prevents use in decrypting the rest.

The main objection to ECB is that the results bear some resemblance to the original plaintext. That is, attributes of the data are not completely reduced to a noise-like pattern. Further, since each block is independent, substitution injection and replay attacks (see “Types of Attacks” on page 13 for more information on different attack types) are possible (meaning one block can be replaced by a “man-in-the-middle” and/or older blocks replayed (also see “Verification”).

CBC or chained encryption mixes the results from one block into the next, making substitution almost impossible and destroying pattern exposure.

Cipher feedback (CFB) is similar to CBC, but it reverses the merging process.

Counter (CTR) allows for random access by using a block count, instead of the results from a previous block.

ECB is the fastest and smallest algorithm, but only by a bit (see Table 1 on page 15 and Table 3 on page 15 for more information). Selecting which method you use is usually based on your application: how it processes the data, and how prone the plaintext is to attack or pattern. Unless the data is always 128 bits or less, CBC is the most common choice for streamed data.

- **Key encoding or pre-expanded**

The normal model is to take the key and expand it for the encryption or decryption process (separate process for each) just before use. An alternative is to pre-expand the key at manufacturing or configuration time for one or all endpoints of the connection. This saves some flash, but more importantly, this saves up to 68 words (272 bytes) of SRAM and expansion time, and it makes it harder to find the key. A small Windows application is provided, aes_gen_key.exe, which generates the expanded key as data or code, depending on command- line switches.

If pre-expanded is used, it is possible to have the expanded key encoded as instructions (code). This does not save RAM, but it makes the key far more secure. By making the key into pure code (versus data in flash), the Stellaris OTP feature can be used to make the code execute only (no read). This means that the encoded key cannot be read from flash; it can only be loaded into RAM during encrypt/decrypt execution, after which it is scrubbed.

The data length of a pre-expanded key is 44 words for 128-bit keys, 54 words for 192-bit keys, and 68 words for 256-bit keys. If the pre-expanded key is generated as data, then this space is allocated in flash, and the run-time key-expanding function is not needed. If the pre-expanded key is not used, then the expanded key is not using up space in flash, but instead the run-time key-expanding function is, and furthermore, the space for the expanded key now resides in RAM. Typically, RAM is a more scarce resource than flash in which case it is better to use the pre-expanded key as data if possible. The instruction-based pre-expanded keys use approximately two to four times as much flash and require as much RAM as run-time expansion. This means that the instruction-based approach, while more secure, does not save flash or RAM space.

- **Key size**

AES supports 128-, 192-, and 256-bit keys. This affects the number of key rounds processed and the buffer size (if the key is not pre-expanded). It is possible to allow run-time selection of the key size, but less code and data is used if only one key size is used. Only one size may be selected if a pre-expanded key is used.

In general, with any encryption, key size relates directly to strength. A small key is easier to crack (for example, an 8-bit key could be cracked in seconds, a 56-bit key could take hours, a 256-bit key could take years). Of course, the strength of the encryption algorithm relates to the strength, and AES is quite strong.

The major weakness is when a chunk of a message's plaintext is known externally, making the code much easier to crack. If the longest predictable string fully occupies one block, which in AES is always 128-bits and there are a set of them (to reduce possible key transforms), initialization vector (initialization vector) schemes are critical, even if using CTR. To avoid this, consider the key size and style of encryption to the longest predictable string and ensure it never fully occupies one block.

The source code defaults are set for most common use, which is 128-bit keys encoded at run-time using EBC and supporting both encryption and decryption. This offers reasonable safety and minimal size, and requires no initialization vector.

Adding AES Encryption to Your Application

Note: See the AES examples that are available in the StellarisWare® software distribution for the following evaluation boards: EK-LM3S1968, EK-LM3S2965 (revc), EK-LM3S3748, EK-LM3S6965 (revc), EK-LM3S8962, EK-LM3S9B90, EK-LM3S9B92, as well as the DK-LM3S9B96 development board.

To add AES encryption to your application:

1. Set the configuration options using the aes_config_opts.h file. Default is both encryption and decryption, but you can select just one. Unless the highest performance is required, use the single- table model.

2. Select which method to use (see "Encryption Method Summary" on page 5). CBC is safer than EBC, but it is a bit slower and needs an initialization vector (see "Selecting an Initialization Vector" on page 10).

3. Decide whether to store the original key (or keys) or the expanded version, based on performance and space (especially RAM) requirements.

4. Set the keys for your application:

   a If using key encodings, call either the `aes_setkey_enc()` function or the `aes_setkey_dec()` function for encryption or decryption. The process requires holding up to 68 words of RAM for each direction. and results in the encoded key in memory with a size of 176, 216, or 272 bytes, depending on key size. Table 2 on page 15 shows the time required for encoding.

   b If using prebuilt keys in flash as data, call either the `AESExpandedEncryptKeyData()` function or the `AESExpandedDecryptKeyData()` function, as generated by the `aes_gen_key.exe` tool.

   c If using pre-expanded keys as instructions, call the `AESLoadExpandedEncryptKey()` function or the `AESLoadExpandedDecryptKey()` function as generated by the

`aes_gen_key.exe` tool. The process requires the same amount of RAM as Option 4a for key encodings.

5. If using CBC, CFB, or CTR, you must have an initialization vector. This is the first value to process (or used in every block for CTR) and should be a nonce (only used once). Although often referred to as random, it may more reasonably be an encoding of time and some pre-arranged data such as size of block. The initialization vector is usually passed in the message itself, which may be safe enough. For more details, see “Selecting an Initialization Vector” on page 10.

6. To encrypt, process 128-bit (4 words) blocks of plaintext using the encoded key, the plaintext, and the initialization vector if CBC, CFB, or CTR. For plaintext that is not a multiple of 128 bits, it is up to the application to consider how to determine length on the receiving side. A common technique is to pass the full length in the first block, so partial messages can be easily detected (when some are lost).

7. The process for decryption is the same as for encryption, except that the incoming data is always an even multiple of 128 bits and the initialization vector likely comes with the data.

Selecting an Initialization Vector

When using CBC, CFB, and CTR, it is important to decide how and whether the initialization vector will be known to both sides. Generally, initialization vectors should not be repeated because they allow replay attacks (man-in-the-middle resends that previous packet to unlock something or gain some other value).

The typical model is to use date and time and/or randomness to construct the initialization vector. Randomness is a hard thing to produce in any computer system, especially an embedded one. Use of time to pick the seed (or for the initialization vector itself) has a weakness in that time in an embedded system can be manipulated from the outside. For example, if the time is a count from reset, an attack can involve resetting the device (or power up) to control the time precisely. Similarly, when an RTC relies on a backup battery. If time is from the “network,” the risk is that this injects a fake time to ensure precise control of the device’s sense of time.

Use of floating pins and other techniques often are prone to attack as well (if these are known). Generally, the choice of required strategy depends on how serious the threat.

For most systems, use of time is good enough. Randomness is not optimal for embedded systems, since these will not want to check whether a value was used before, and pseudo-random tables take a lot of space.

Time may be passed in the message, and then the receiver verifies the time is valid within some plus-or-minus value (relative to sending time; for example, within a couple seconds). Any initialization vector with a time that is not close enough is rejected. This prevents basic replay attacks within the limits of time control described above.

If stronger provisions are needed, there are two easy ways to do this in embedded systems:

- Have each side record into flash an agreed count after acceptance of messages. This means that the initialization vector contains a number that changes after each message and does not repeat for 2^{32} messages. This prevents replay attacks, unless an attacker can rewrite the flash to reset counters. It is crucial that acceptance of the message be known, else the counters can get out of

sync causing all messages to be discarded. The normal model is that acceptance is passed in the next return, so the previous count is only accepted for NAKs (did not get a return/reply), otherwise it must be the next count; since a NAK only causes a resend, there is no replay advantage (although malicious attacks can use a fake NAK to get the two sides out of sync, so careful retraining may be needed for safety).

- Have the first message send its time (A sends a time number to B), but let the second message (B to A) use the variability in reception (delay between send and receive) to control the next time/ initialization vector down to nanoseconds. This only works with unclocked communications (for example, ENET, CAN, and UART, but not SPI or I^2C). This means using a first exchange set to get a variable time factor. The variable factor is passed in a reply (B to A). That is used in the initialization vector of the next message from A. This way, even if a fake variable value is injected in a message to B, the message is rejected since B knows what it used (i.e., tricking A does not help). The risk is that if the variability range is small enough, enough retries allow return to previously used delay number, allowing replay attacks. To prevent this, the variable delay is normally updated by one or both sides (in messages, with results showing in the initialization vector). This cannot be used when reception loss is an issue.

Although it is acceptable to pass the initialization vector in the message, common methods to add a bit of strength is to pass the next initialization vector in the previous message, or to pre-arrange parts of the initialization vector based on IDs, count of messages, etc. This makes it harder for man-in-the-middle attacks to generalize source for replay attacks (for example, makes it look like a message came from another component). Any scheme that relies on history needs to take care if reception loss is a problem. If replay attacks are not an issue (see “Verification), then prearranged initialization vectors can be used.

An example of this scheme is provided in the `aes_expanded_key` example application in the aes_iv.c file.

Verification

It is important to remember that decryption cannot tell if the incoming data is wrong. It simply converts incoming data into some other form. Therefore, it is often important to verify the correctness of the plaintext emitted. This may be done using a number of techniques, in part based on what causes of error are to be considered. The three most common error forms are:

- **Message bit errors.** Errors in the transferred data will translate to one or more errors in the decrypted result. For chained methods, they will usually translate to two or more errors. If the communication link is subject to such errors, provisions for rejection or correction must be made.

- **Injection and substitution errors.** Especially when messages consist of an authentication portion followed by a command/request, attacks may involve trial and error at modifying part of a message. Other attacks may only be intended to be disruptive, such as causing a security system to ignore a seemingly broken remote sensor.

- **Missing portion errors.** If some of the message is missing, especially off the end, it is important to know that. A common technique with any of the chained systems is to put the initialization vector at the end. If no initialization vector at the end, the whole message is junk. This is less likely to be useful in embedded systems for longer messages as the whole message has to be stored before it can be decrypted (versus serial decryption and process).

Depending on type of errors expected, the most common verification techniques include:

- ECC to catch and correct up to the number of bit errors expected.
- CRC to catch errors within some degree of accuracy.
- Markers/tags and length information to understand the basic layout and confirm blocks are generally valid. For example, `0x55,somedata_4,0xAA`; if the `0x55` and `0xAA` are not seen, then there is a problem. Coupled with local check-sum or CRC adds local confidence (and improves accuracy).
- Redundancy (often using NOT or XOR style) to catch bit errors and bit attacks.

General Considerations

An encryption method is only as good as its weakest link. Using a strong system with strong keys is of no value if the keys are left unsecured. General provisions must be considered to make this system actually secure (to the level needed):

- If the flash of the device can be read back, then the key or keys can be read back.
  - Disable of debug is the first step, as it removes direct access.
  - Use of keys as code and use of the Stellaris OTP mechanism to make executable/no-read can remove direct exposure.
- If new flash images are to be loaded (field upgrade), then this presents two weaknesses: someone can read the flash content to find the keys or someone can modify the image to leak information or remove protection.
  - The most common protection for the first is to OTP the keys and encryption code into flash, so that the downloaded image does not contain it.
  - The most common protection for the second is to have the OTP code perform some sort of CRC or MD5 style hash on the code image. If the CRC/hash is long enough, this cannot be easily worked out.
- If too many people know the key (for example, customers, system integrators, different vendors), the source of leaks goes up quickly.
  - The most common technique is to apply a key at configuration or install time. The source of the key may be “random” or pulled from a key server (off the Internet).
  - When there is a master device and the rest are registered (in some safe way) to it, another approach is to have the master define the key automatically. This can be based against a laser-inscribed number or other detail known only to the master.
  - Worse case is to move to a key exchange model, as used by TLS/SSL.
- If the key has to be exchanged across an open system, then that exposes it to snooping.

  - The most common method is to use a technique like TLS/SSL. That is basically a bootstrap model using public key approaches to doing the exchange. That method adds a lot of code and data requirements.
  - Another approach has starting (seed) keys (or key) known only to both sides, which are then used to pass the new key or keys in encrypted from. The problem is, of course, the “known only to both sides” clause. This method is often used to create “session keys” (keys whose lives are only as long as a session of communications, but sometimes more broadly to a one-time registration), so that the risk is reduced only to the initial exchange. To make the system a bit safer, it is often the case that the seed key is modified by knowledge of the specifics of the endpoint; for example, the type of device being connected; this limits man-in-the-middle attacks to ones where the attacker would know the same details.
  - A similar seed-to-session model is used while the network is off the WAN. This model is often used, for example, to register security remotes during install. So, even though standard networking was used for the session setup, only private session keys are used on the open network.

- If using ECB, it is important to understand substitution attacks. These involve replacing some parts of a message while leaving others alone. The most common example is when authentication is at the start and the command is at the end. The attack replaces the command with another (used in some previous context).

  The use of CBC or other serialization model prevents this.

- One often forgotten consideration is closure. If your message traffic involves open something, do some actions, close the thing—then attacks can involve intercepting and removing the “close” message. This can allow privileges or access that was not considered.
  - Time-outs are the most common method, but these run the risk of opening a time period for attacks (for example, replay attacks).
  - Avoiding this approach, if possible, is optimal. Self-contained messages (or meta-messages) are preferred for this reason. This also avoids the risk of message loss due to errors on the line.
  - Use of an open that indicates what comes next can be used to avoid some replay attacks, although this still opens a time window.

- Most other considerations are covered in “Selecting an Initialization Vector” on page 10.”

Types of Attacks

Attacks are attempts by a person or program to find the key, determine the original plaintext, or inject messages into the system. There are multiple ways to attack encrypted code. You should be familiar with the types of attacks when you select your encryption method. The types of attacks are described in detail below:

- **closure** or **session attacks**

  There are no standard names for this style, but the method involves intercepting and removing an “end” or “close” message. This can allow insertion or *replay attacks* to gain access otherwise not

possible. For example, if a session is opened and sets privilege, followed by a set of privileged commands and then close, then removal of the close would keep the privilege level active.

- **injection attacks**

  An attempt by a person or program to insert their own messages into the stream. Normally this would be use to gain information, gain authorization, or damage the system. See also *replay attacks*.

- **known pattern attacks**

  Using a fixed pattern in each message gives the attacker something to work against. This is particularly problematic when at the start or end of the message. Block size, whether you carry data forward (for example, *CBC or chained encryption*), and position knowledge all can contribute to how easy or hard it is. For some things, like processor instructions, there is little that can be done (for example, a return instruction is usually a fixed instruction pattern no matter where it is located, so it is easy to find and target).

- **man-in-the-middle**

  A common attack form is to intercept messages between A and B (traditionally, people's names are used, but A and B work fine) by being able to access the communications route (physical cable/radio or virtual routing). This permits interception (to allow decoding attempts), injection (sending messages appearing to come from A to B, or B to A), and disruption.

- **removal attacks**

  A common attack form is to remove certain messages which are recognized. If you sent a message to say to lock a door, and that message looks the same every time, then the attack is a matter of intercepting and removing the message from the stream. It is important to make sure Messages have context (for example, time) and/or vary enough to ensure that this is not possible. The attacker does not have to know the *key*s or anything about the *plaintext* if they can associate the message with the action.

- **replay attacks**

  An injection method of sending an older message again to gain advantage. So, the attacker will copy messages and then inject them later to cause the same behavior. For example, if a message is (validly) sent to unlock access, then replaying that at later time can gain the same unlock. Since the receiver sees a valid message, it will perform the task. The attacker never needs to know the *key*.

- **substitution attacks**

  These types of attacks replace one or more blocks, but the leave the rest of the message alone. This can work against ECB, because many times the first part contains the authentication information, and the latter part contains the request. So, if you have `block0=authentication`, `block1=message`, someone can intercept a new two-block message and substitute an older message (say to grant to new rights) to gain new privileges. This is why the stream-oriented *encryption* methods are used.

Sizes

The sizes of the AES package when compiled with gcc (-O3 –Os) are shown in Table 1. This shows some sizes, but not all combinations. Note that many variants may seem surprising. It needs to be understood that the Encode table and its S-box are 1024+256 bytes. The Decode table and its S-box are also 1024+256. The decode-only case is larger than expected, because it needs the encode S-box.

The sizes given for pre-expanded keys do not include the size of the encoded key (in code or data); those sizes are shown in Table 2.

Table 1. Representative Sizes of Code and Read-Only Memory

Build Rules	Code (flash)	Table (flash)
Encode+Decode, 1 Table, 128 bits, ECB only, Encode keys at runtime. DEFAULT	2672/0xA70	2600/0xA28
Encode+Decode, 1 Table, all sizes, ECB only, Encode keys at runtime.	2984/0xBA8	2600/0xA28
Encode+Decode, 1 Table, 128 bits, Pre-encoded keys as data.	1900/0x76C	2560/0xA00
Encode+Decode, 1 Table, 128 bits, Pre-encoded keys as code.	1900/0x76C	2560/0xA00
Encode only, 1 Table, 128, ECB only, Encode keys at runtime.	1344/0x540	1320/0x528
Decode only, 1 Table, 128, ECB only, Encode keys at runtime.	1584/0x630	1576/0x628

Table 2. Size of Pre-Expanded Keys (one for encode, one for decode)

Key Size	Expanded Data Key	Expanded Code Key
128	176	356 to 704[a]
192	216	436 to 864[a]
256	272	548 to 1088[a]

a. The size range depends on the number of repeated values.

Performance

The performance numbers shown in Table 3 are for 128-bit keys using one table (see DEFAULT in Table 1) using the same compiler and switches used for the sizes section. These show the time (in cycles) taken to prepare keys, encrypt a 128-bit block, and decrypt a 128-bit block. The times are based on passing in an array, thus representing the time taken to unpack the data from an array and repack on completion.

To understand performance, it is important to realize that AES encryption and decryption is bound by random table loads, XORs, key loads, byte extracts, and rotates. A 128-bit key processing a 128-bit block requires 10 rounds or processing, where each round has 16 table loads, 20 XORs, 4 accesses to expanded key data, plus byte extracts from the block data (to use as table indexes) and rotations.

Table 3. Cycle Times for Four Main Operations

Operation	Cycles	Cycles/Byte
Encrypt key encode	546[a]	

Table 3. Cycle Times for Four Main Operations (Continued)

Operation	Cycles	Cycles/Byte
Decrypt key encode	2387[b]	
Encrypt a 128-bit block	1329	83
Decrypt a 128-bit block	1347	84

a. This is for encoding at runtime. If pre-expanded as data, cost is 0. If pre-expanded as code, cost is approximately 160 cycles.
b. Decrypt keys are formed by first creating an encryption key. If both are to be left in memory at the same time, then this is the cost of both keys.

References

The following documents and source code are available for download at www.ti.com/stellaris:

- StellarisWare® Driver Library, Order number SW-DRL
- *StellarisWare® Driver Library User's Guide*, publication number SW-DRL-UG
- *Evaluating Matrix SSL for Stellaris Microcontrollers Application Note*, Publication Number AN01244