JN516x/7x AES Coprocessor API Reference Manual

JN-RM-2013
Revision 2.0
31-Jan-2017

Contents

Preface

This manual describes the Application Programming Interface (API) to the AES Coprocessor on the NXP JN516x and JN517x wireless microcontrollers. API functions are described that can be used to set up, control and respond to events generated by the AES block.

It is assumed that the reader has knowledge of the AES encryption algorithm.

Organisation

This manual consists of four chapters, as follows:

- Chapter 1 introduces the AES Coprocessor API.
- Chapter 2 describes the AES Block Cipher Mode functions – a set of high-level functions that can be used to perform encryption/decryption on the AES Coprocessor.
- Chapter 3 describes the AES Coprocessor low-level functions.
- Chapter 4 details the structures used by the AES Coprocessor API.

Conventions

Files, folders, functions and parameter types are represented in **bold** type.

Function parameters are represented in *italics* type.

Code fragments are represented in the Courier typeface.

Acronyms and Abbreviations

ACL	Access Control List
AES	Advanced Encryption Standard
API	Application Programming Interface
CCM	Counter with CBC-MAC
ECB	Electronic Code Book
PIB	PAN Information Base

Support Resources

To access online support resources for the JN516x and JN517x devices, visit the Wireless Connectivity area of the NXP web site:

www.nxp.com/products/interface-and-connectivity/wireless-connectivity

Trademarks

1 Introduction to the AES Coprocessor API

The AES Coprocessor API provides a thin layer above the registers used to control the security engine of the JN516x and JN517x wireless microcontrollers – the AES Coprocessor. The API allows several register accesses to be encapsulated into one function call, making this on-chip peripheral easier to use, without a detailed knowledge of its operation.

The functions provided by the AES Coprocessor API are defined in the header file **AHI_AES.h**.

1.1 Basic Mechanism of API

The AES Coprocessor API facilitates hardware acceleration for encoding and decoding data blocks for use by applications or by higher stack layers such as ZigBee. This encryption/decryption is additional to IEEE 802.15.4 security, which is performed during frame transmission and reception under the control of the IEEE 802.15.4 stack using the same on-chip hardware accelerator.

By using hardware acceleration, it is possible to greatly increase security encode and decode performance. The API provides a mechanism for an application to pass in blocks of data for encoding or decoding, and for the resulting data to be placed in a second memory block. It is possible for both blocks to be at the same location, for improved memory use.

The security information required includes a 128-bit key, security level and any data that is used to initialise the security engine but which is not part of the encoded data. The security engine is operated independently of the stack security, and hence does not use the ACL entries in the PIB.

1.2 Categories of API Functions

The AES Coprocessor API contains two types of function:

- **AES Block Cipher Mode functions**: These functions fully configure the AES Coprocessor into a specified AES Cipher mode. You simply provide the configuration data, and the API will configure the Coprocessor and return a result. These functions are described in Chapter 2.
- **AES Coprocessor low-level functions**: The API also includes a number of functions that provide low-level access to the AES Coprocessor registers. These functions are described in Chapter 3.

Note: The AES Coprocessor function calls are blocking, i.e. they only return when the encode or decode operation has completed.

Note: The low-level functions, described in Chapter 3, should not normally be needed. The high-level functions, described in Chapter 2, should provide sufficient access to the AES Coprocessor for most users.

2 AES Block Cipher Mode Functions

This chapter details the AES Block Cipher Mode functions which provide high-level interaction with the AES Coprocessor.

The functions are listed below along with their page references.

Important: For JN517x devices, there is an endian consideration when mapping data into the `tsReg128` and `tuAES_Block` structures used in these functions. If data is written directly into the structure then the called function will work as expected. If the data is being transferred from elsewhere using `memcpy` then the data needs to be endian-corrected for each element of the structure before the function is called.

Note: For details of the `tsReg128` and `tuAES_Block` structures, refer to Chapter 4.

bACI_ECBencodeStripe

```
PUBLIC bool_t bACI_ECBencodeStripe(
                                   tsReg128 *psKeyData,
                                   bool_t bLoadKey,
                                   tsReg128 *psInputData,
                                   tsReg128 *psOutputData);
```

Description

This function performs AES ECB encryption on a single 128-bit stripe, using the AES Coprocessor. The function returns when the encode has completed.

The API imposes no buffer alignment requirements on the user. It is the responsibility of the user to generate any zero-padding needed for input data with lengths that are not a multiple of 128 bits.

For JN517x devices, refer to the note about endianness on page 5.

Parameters

*psKeyData	Pre-allocated pointer to structure containing 128-bit key data
bLoadKey	Specifies whether a new key is to be loaded (TRUE for new key, FALSE otherwise)
*psInputData	Pre-allocated pointer to structure of input data
*psOutputData	Pre-allocated pointer to structure of output data

Returns

TRUE – The function has completed successfully.

FALSE – The function has not completed. The AES Coprocessor is busy or there has been an input parameter error.

bAES_CCMstar

```
PUBLIC bool_t bAES_CCMstar(
                    tsReg128 *psKeyData,
                    bool_t bLoadKey,
                    uint8 u8AESmode,
                    uint8 u8M,
                    uint8 u8alength,
                    int8 u8mlength,
                    tsReg128 *psNonce,
                    uint8 *pau8authenticationData,
                    uint8 *pau8inputData,
                    uint8 *pau8outputData,
                    uint8 *pau8checksumData,
                    bool_t *pbChecksumVerify);
```

Description

This function performs CCM* AES encryption/decryption with checksum generation/verification, using the AES Coprocessor. It is a software implementation of the AES CCM* hardware.

Note that:

- In encode mode (CCM), the function will return a checksum in the *au8checksumData* buffer upon completion.
- In decode mode (CCM_D), the function expects the checksum to be the last *M* bytes of the *au8inputData* buffer, and so *mlength* = (data length + *M*) bytes.

The function returns when the encryption/decryption has completed or there has been an input parameter error or the AES Coprocessor is busy.

The API imposes no buffer alignment requirements on the user, and also generates any stripe zero-padding needed for input data (on data lengths that are not a multiple of 128 bits).

For JN517x devices, refer to the note about endianness on page 5.

Parameters

**psKeyData*	Pre-allocated pointer to structure containing 128-bit key data
bLoadKey	Specifies whether a new key is to be loaded (TRUE for new key, FALSE otherwise)
u8AESmode	Required CCM* mode of operation of the AES Coprocessor. The supported modes are: IEEE CTR (XCV_REG_AES_SET_MODE_CTR) CCM Encode (XCV_REG_AES_SET_MODE_CCM) CCM Decode (XCV_REG_AES_SET_MODE_CCM_D)
u8M	Required number of checksum bytes. Supported values are 0, 2, 4, 8, 16 and 32
u8alength	Length of authentication data, in bytes
u8mlength	Length of input data, in bytes
**psNonce*	Pre-allocated pointer to structure containing 128-bit nonce data
**pau8authenticationData*	Pre-allocated pointer to byte array of authentication data

**pau8inputData*	Pre-allocated pointer to byte array of input data
**pau8outputData*	Pre-allocated pointer to byte array of output data
**pau8checksumData*	Pre-allocated pointer to byte array of checksum data. In CCM decode mode (CCM_D), this value can be NULL
**pbChecksumVerify*	Pre-allocated pointer to boolean which in CCM decode mode (CCM_D) stores the result of the checksum verification operation. Can be NULL in other modes (CCM and CTR)

Returns

TRUE – The function has completed successfully.

FALSE – The function has not completed. The AES Coprocessor is busy or there has been an input parameter error.

vACI_OptimisedCCMstar

```
PUBLIC void vACI_OptimisedCCMstar(
                                  bool_t bEncrypt,
                                  uint8 u8M,
                                  uint8 u8alength,
                                  int8 u8mlength,
                                  tuAES_Block *psNonce,
                                  uint8 *pau8authenticationData,
                                  uint8 *pau8Data,
                                  uint8 *pau8checksumData,
                                  bool_t *pbChecksumVerify);
```

Description

This function performs CCM* AES encryption/decryption, like the function **bAES_CCMstar()**, but is optimised for speed and memory.

The function assumes that the input and output buffers are word-aligned. It also assumes that the encryption/decryption key has already been loaded.

The first byte of the nonce is used to carry the CCM* flags data and should be left unpopulated by the calling function (this is different from **bAES_CCMstar()**).

Note that:

- In encode mode (CCM), the function will return a checksum in the *au8checksumData* buffer upon completion.
- In decode mode (CCM_D), the function expects the checksum to be the last *M* bytes of the *au8inputData* buffer, and so *mlength* = (data length + *M*) bytes.

The function returns when the encryption/decryption has completed or the AES Coprocessor is busy or there has been an input parameter error.

For JN517x devices, refer to the note about endianness on page 5.

Parameters

bEncrypt	Specifies whether encryption or decryption is required (TRUE - encryption, FALSE - decryption)
u8M	Required number of checksum bytes. Supported values are 0, 2, 4, 8, 16 and 32
u8alength	Length of authentication data, in bytes
u8mlength	Length of input data, in bytes
**psNonce*	Pre-allocated pointer to structure containing 128-bit nonce data
**pau8authenticationData*	Pre-allocated pointer to byte array of authentication data (must be word-aligned)
**pau8Data*	Pre-allocated pointer to byte array for input and output data (must be word-aligned)
**pau8checksumData*	Pre-allocated pointer to byte array of checksum data (must be word-aligned). In CCM decode mode (CCM_D), this value can be NULL
**pbChecksumVerify*	Pre-allocated pointer to boolean which in CCM decode mode (CCM_D) stores the result of the checksum verification operation. Can be NULL in other modes (CCM and CTR)

Returns

None

bACI_CCMstar

```
PUBLIC bool_t bACI_CCMstar(
                           tuAES_Block *psKeyData,
                           bool_t bLoadKey,
                           uint8 u8AESmode,
                           uint8 u8M,
                           uint8 u8alength,
                           int8 u8mlength,
                           tuAES_Block *psNonce,
                           uint8 *pau8authenticationData,
                           uint8 *pau8inputData,
                           uint8 *pau8outputData,
                           uint8 *pau8checksumData,
                           bool_t *pbChecksumVerify);
```

Description

This function replicates the AES CCM* hardware functionality that was found on earlier JN51xx wireless microcontrollers (pre-JN516x) and is provided for backward compatibility. It performs CCM* AES encryption/decryption with checksum generation/verification, using the AES Coprocessor. It can perform any operation in the CCM and CCM* encryption suites.

Note that:

- In encode mode (CCM), the function will return a checksum in the *au8checksumData* buffer upon completion.
- In decode mode (CCM_D), the function expects the checksum to be the last *M* bytes of the *au8inputData* buffer, and so *mlength* = (data length + *M*) bytes.

The function returns when the encode has completed or the AES Coprocessor is busy or there has been an input parameter error.

The API imposes no buffer alignment requirements on the user, and also generates any stripe zero-padding needed for input data (on data lengths that are not a multiple of 128 bits).

For JN517x devices, refer to the note about endianness on page 5.

Parameters

**psKeyData*	Pre-allocated pointer to structure containing 128-bit key data
bLoadKey	Specifies whether a new key is to be loaded (TRUE for new key, FALSE otherwise)
u8AESmode	Required CCM* mode of operation of the AES Coprocessor. The supported modes are: IEEE CTR (XCV_REG_AES_SET_MODE_CTR) CCM Encode (XCV_REG_AES_SET_MODE_CCM) CCM Decode (XCV_REG_AES_SET_MODE_CCM_D)
u8M	Required number of checksum bytes. Supported values are 0, 2, 4, 8, 16 and 32
u8alength	Length of authentication data, in bytes
u8mlength	Length of input data, in bytes
**psNonce*	Pre-allocated pointer to structure containing 128-bit nonce data

**pau8authenticationData*	Pre-allocated pointer to byte array of authentication data
**pau8inputData*	Pre-allocated pointer to byte array of input data
**pau8outputData*	Pre-allocated pointer to byte array of output data
**pau8checksumData*	Pre-allocated pointer to byte array of checksum data. In CCM decode mode (CCM_D), this value can be NULL
**pbChecksumVerify*	Pre-allocated pointer to boolean which in CCM decode mode (CCM_D) stores the result of the checksum verification operation. Can be NULL in other modes (CCM and CTR)

Returns

TRUE – The function has completed successfully.

FALSE – The function has not completed. The AES Coprocessor is busy or there has been an input parameter error.

3 AES Coprocessor Low-level Functions

This chapter details the low-level functions of the AES Coprocessor API. These functions allow direct access to the registers of the AES Coprocessor on the JN516x/7x wireless microcontroller.

Note: The low-level functions, described in this chapter, should not normally be needed. The high-level functions, described in Chapter 2, should provide sufficient access to the AES Coprocessor for most users.

The functions are listed below along with their page references.

vACI_WriteKey

PUBLIC void vACI_WriteKey(tsReg128 ***psKeyData*);**

Description

This function loads the 128-bit AES key into the AES Coprocessor core.

Parameters

psKeyData	Pre-allocated pointer to structure containing 128-bit key data

Returns

None

vACI_CmdWaitBusy

```
PUBLIC void vACI_CmdWaitBusy(void);
```

Description

This function is a blocking function that is used to wait for the AES Coprocessor to complete execution of the current command.

Parameters

None

Returns

None

bACI_isBusy

```
PUBLIC bool_t bACI_isBusy(void);
```

Description

This function is used to determine whether the AES Coprocessor is busy processing a command from the API or the underlying hardware.

Parameters

None

Returns

TRUE – The Coprocessor is busy

FALSE – The Coprocessor is not busy and may accept a command from API

4 Structures

The AES Coprocessor API uses the structures described below.

Important: For JN517x devices, there is an endian consideration when mapping data into the `tsReg128` and `tuAES_Block` structures described below. If data is written directly into the structure then the data can be used 'as is'. If the data is being transferred from elsewhere using `memcpy` then the data needs to be endian-corrected for each element of the structure before it is used.

4.1 tsReg128

This is a 128-bit data and configuration data structure:

```
typedef struct
{
   uint32 u32register0;
   uint32 u32register1;
   uint32 u32register2;
   uint32 u32register3;
} tsReg128;
```

where:

- `u32register0` contains the top 32 bits of the 128-bit data stripe
- `u32register1` contains the next-to-top 32 bits of the data stripe
- `u32register2` contains the next-to-bottom 32 bits of the data stripe
- `u32register3` contains the bottom 32 bits of the data stripe

4.2 tuAES_Block

This is a security block definition structure:

```
typedef union
{
    uint8    au8[AES_BLOCK_SIZE];
    uint32   au32[AES_BLOCK_SIZE / 4];
} tuAES_Block;
```

where:

- `au8[]` is an array containing the AES data block as bytes
- `au32[]` is an array containing the AES data block as 32-bit words

Revision History

Version	Date	Description
1.0	30-Mar-2006	First release
1.1	18-Sep-2006	Put in new template and re-worked
1.2	20-Sep-2006	Streamlined API description
1.3	06-Dec-2007	Added JN5139 and changed header filename
2.0	31-Jan-2017	Updated for the JN516x and JN517x families of devices, and incorporated relevant low-level functions (previously documented in JN-RM-2028)

Important Notice

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the *Terms and conditions of commercial sale* of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

NXP Semiconductors

For the contact details of your local NXP office or distributor, refer to:

www.nxp.com

