VIA PadLock SDK

API & Programming Guide

VIA Technologies, INC.
9/1/2004

Trademarks

- Windows® 98/Me/2000/XP are trademarks or registered trademarks of Microsoft Corporation.
- All other brands, product names, company names, trademarks and service marks are the property of their respective companies.

Contact Us

VIA Technologies, INC.
525-535 Chung-Cheng Road, Hsin-Tien, Taipei 231, Taiwan
Phone：886-2-2218-5452
FAX: 886-2-8667-1804

Revision History

2004/09/01	Content Update and Finalization	Jenny Chien
2004/08/23	Content Update	Danial Miao
2004/08/20	Source Code and Document consistency Check	Jenny Chien
2004/07/07	Document Format	Jenny Chien
2004/06/16	Content Update	Danial Miao
2004/06/14	Content Update and document format	Jenny Chien
2004/05/26	Initial Release	Bruce Zheng

Table of Contents

1 Introduction

This Application Programming Interface (API) and Programming guide is for users using the VIA PadLock SDK for VIA processors based on the C5XL/C5P core. The VIA PadLock ACE (Advanced Cryptography Engine), embedded in the C5P and C5XL Nehemiah processor core architectures, implements the cryptography functionality. The VIA PadLock RNG (Random Number Generator) function in VIA processors based on the C5XL/C5P Nehemiah core architecture implements the random number generating functionality.

The VIA PadLock SDK is comprised of four groups of APIs, which will facilitate users to build their security applications on VIA C5XL/C5P Nehemiah based processors and will help them make full use of the many advanced features such as the VIA PadLock RNG and VIA PadLock ACE to enhance the overall performance of their applications.

For more details of the VIA PadLock RNG hardware, please refer to *VIA Nehemiah Random Number Generator Programming Guide*.

For more details of the VIA PadLock ACE hardware, please refer to *VIA Nehemiah Advanced Cryptography Engine Programming Guide*.

2 System Requirement

2.1 Hardware requirement

A processor based on the VIA C5P or C5XL Nehemiah cores must be used when using VIA PadLock SDK

2.2 Software requirement

Currently, the VIA PadLock SDK supports two software platforms: Microsoft Windows and Linux. These operating systems are required to implement the VIA PadLock SDK. The relevant development tools chain such as GNU Compiler Collection GNU Make and Microsofte Visual C++ are also needed.

3 Installation & Usage

3.1 Installation

3.1.1 Linux System

For Linux system, GNU Make and GCC should be installed on the system. Follow the steps below to install the VIA PadLock SDK:

1. Uncompress the tar ball in the working directory, for example /home/Jenny

   ```
   [Jenny@localhost Jenny]# tar xzvf padlock-sdk-linux-1.0.0-src.tar.gz
   ```

2. Enter the directory.

   ```
   [Jenny@localhost Jenny]# cd padlock-sdk-linux-1.0.0-src
   ```

3. Install the VIA PadLock SDK

   ```
   [Jenny@localhost padlock-sdk-linux-1.0.0-src]# make install
   ```

3.1.2 Windows System

For Windows system, to install the VIA PadLock SDK library, following the steps below:

1. Unzip package Padlock-SDK-Win-Library-1.0.0.zip.
2. Set proper lib and directory configuration and include “padlock.h” in your own application then use the APIs in the VIA PadLock SDK. Refer to the section "How to use the VIA PadLock SDK" for more information.

3.2 How to use the VIA PadLock SDK

3.2.1 Linux System

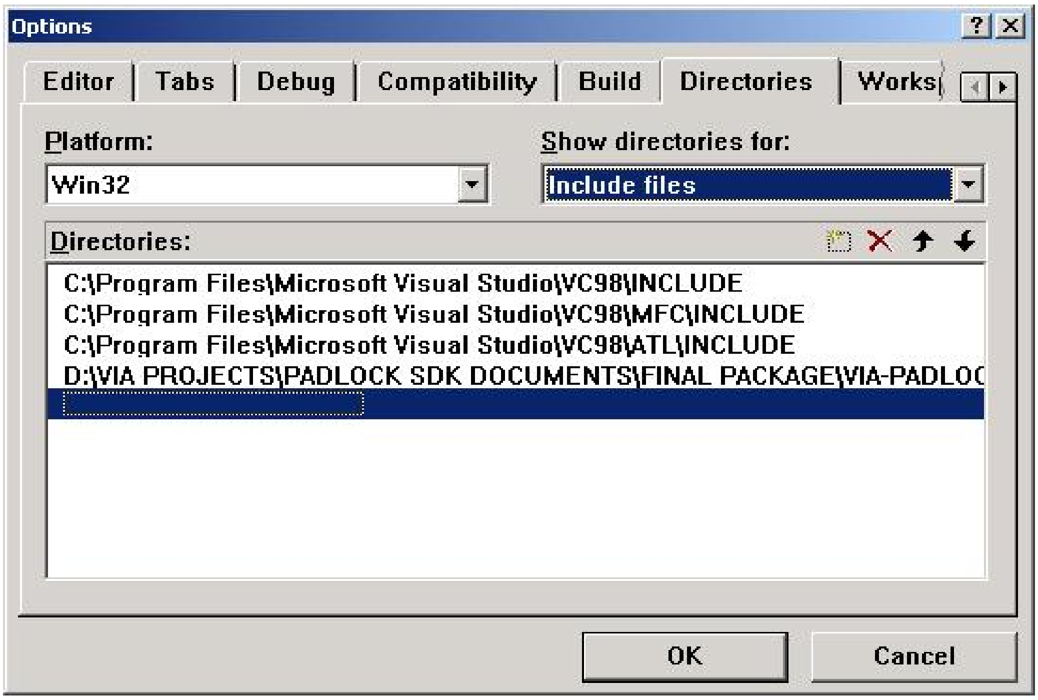
To make full use of the VIA PadLock SDK in Linux, include "`padlock.h`" in your application, For more details, please refer to "`padlock_demo.c`" file in the demo program folder "`padlock-sdk-linux-1.0.0-src/demo`".

You can also modify codes of the VIA PadLock SDK under "`padlock-sdk-linux-1.0.0-src/include`" and "`padlock-sdk-linux-1.0.0-src/src`", then under the "`padlock-sdk-linux-1.0.0-src/`" and run "`make install`", you will then be able to use your own VIA PadLock SDK in your application.

3.2.2 Windows System

For Windows system, follow the steps below to use the VIA PadLock SDK:

1. Unzip Padlock-SDK-Win-Library-1.0.0.zip. There should be three files in this directory: padlock.h, VIA_Padlock_SDK_Library.dll, and VIA_Padlock_SDK_Library.lib.
2. Create your own application project in Visual C++ in which you include “padlock.h”.
3. Click “Tools” -> “Options …” set the correct directory configurations.
   For include directory:



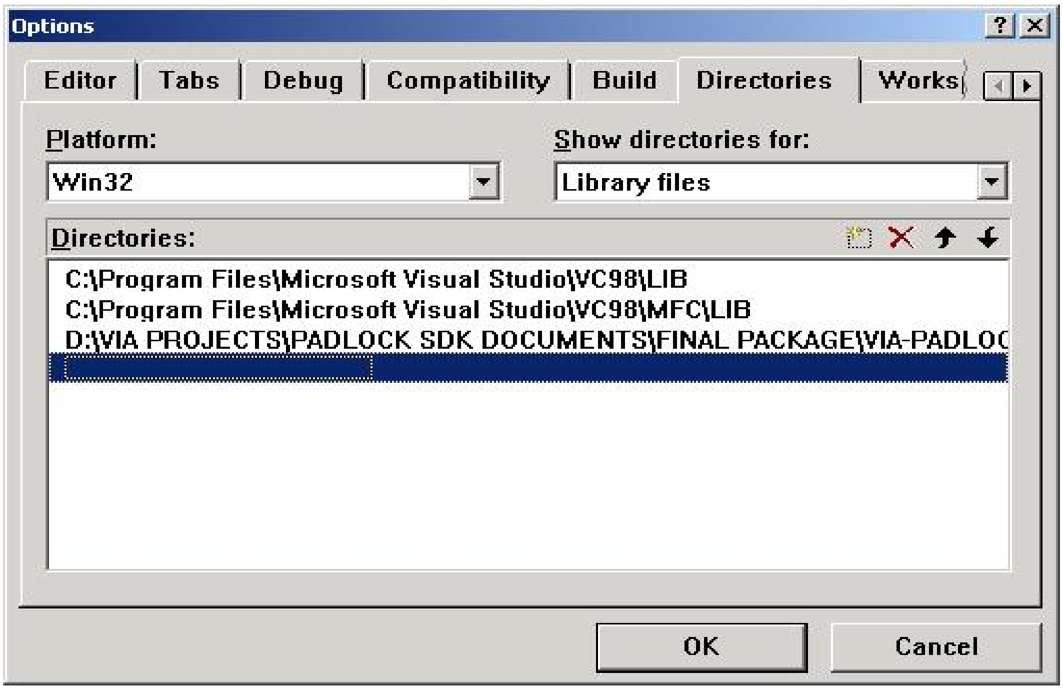
Platform: Win32

Show directories for: Include files

Directories:

```
C:\Program Files\Microsoft Visual Studio\VC98\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\MFC\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\ATL\INCLUDE
D:\VIA PROJECTS\PADLOCK SDK DOCUMENTS\FINAL PACKAGE\VIA-PADLOC
```

For lib directory:



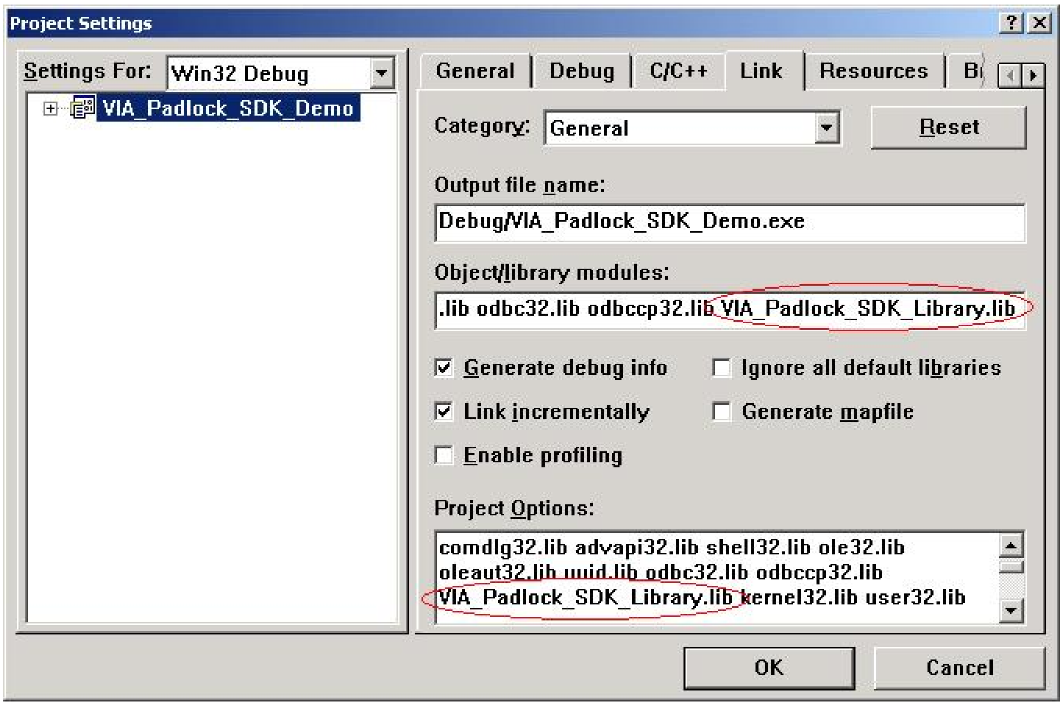
Platform: Win32

Show directories for: Library files

Directories:

```
C:\Program Files\Microsoft Visual Studio\VC98\LIB
C:\Program Files\Microsoft Visual Studio\VC98\MFC\LIB
D:\VIA PROJECTS\PADLOCK SDK DOCUMENTS\FINAL PACKAGE\VIA-PADLOC
```

4. Click “Project” -> “Setting …” set the correct lib configuration.



Settings For: Win32 Debug

Category: General

Output file name: Debug/VIA_Padlock_SDK_Demo.exe

Object/library modules: .lib odbc32.lib odbccp32.lib VIA_Padlock_SDK_Library.lib

☑ Generate debug info ☐ Ignore all default libraries

☑ Link incrementally ☐ Generate mapfile

☐ Enable profiling

Project Options:

```
comdlg32.lib advapi32.lib shell32.lib ole32.lib
oleaut32.lib uuid.lib odbc32.lib odbccp32.lib
VIA_Padlock_SDK_Library.lib kernel32.lib user32.lib
```

5. You will now be able to use all APIs in the VIA PadLock SDK.
   For more details, please refer to VIA_Padlock_SDK_Demo project in package PadlockSDK-Win-Develop-1.0.0.

You can also modify codes of the VIA PadLock SDK in VIA_Padlock_SDK_Library project in the PadlockSDK-Win-Develop-1.0.0 package. Rebuild the SDK library and copy the relevant three files: padlock.h VIA_Padlock_SDK_Library.dll and VIA_Padlock_SDK_Library.lib to VIA_Padlock_SDK_Library directory and use your own VIA Padlock SDK.

4 VIA PadLock SDK API

4.1 Type Definition

4.1.1 struct ace_aes_context

Description: parameter type used in the VIA PadLock SDK ACE normal & plain APIs

4.1.2 struct aligned_memory_context

Description: parameter type used in the VIA PadLock SDK ACE aligned memory APIs

4.1.3 RNG_RESULT

```
typedef enum _RNG_Result
{
      RNG_SUCCEEDED,
      RNG_FAILED
} RNG_RESULT;
```

Description: The return value of random number generating functions.

RNG_SUCCEEDED -- RNG operation succeeded
RNG_FAILED -- RNG operation failed

4.1.4 KEY_LENGTH

```
typedef enum _KEY_LENGTH
{
   KEY_128BITS,
   KEY_192BITS,
   KEY_256BITS
} KEY_LENGTH;
```

Description: The key length type of VIA ACE AES Algorithm.

KEY_128BITS -- 128 bits key
KEY_192BITS -- 192 bits key
KEY_256BITS -- 256 bits key

4.1.5 ACE_AES_MODE

```
typedef enum _ACE_AES_MODE
{
    ACE_AES_ECB,
    ACE_AES_CBC,
    ACE_AES_CFB128,
    ACE_AES_OFB128
} ACE_AES_MODE;
```

Description: The cipher mode type of VIA PadLock ACE AES algorithm.

ACE_AES_ECB -- ECB (Electronic Code Book) Mode
ACE_AES_CBC -- CBC (Cipher Block Chaining) Mode
ACE_AES_CFB128 -- 128 bits CFB (Cipher Feed Back) Mode
ACE_AES_OFB128 -- 128 bits OFB (Output Feed Back) Mode

4.1.6 AES_RESULT

```
typedef enum _AES_Result{
    AES_SUCCEEDED,
    AES_FAILED,
    AES_ADDRESS_NOT_ALIGNED,
    AES_NOT_BLOCKED,
    AES_KEY_NOT_SUPPORTED,
    AES_MODE_NOT_SUPPORTED
} AES_RESULT;
```

Description: The return value of the VIA PadLock ACE APIs.

AES_ADDRESS_NOT_ALIGNED -- the addresses of plain text and cipher text and initialization vector must be aligned by 16 bytes, which is demanded by the VIA PadLock ACE hardware
AES_NOT_BLOCKED -- the length of plaintext and cipher text must be multiples of 16 bytes, which is demanded by the VIA PadLock ACE hardware
AES_KEY_NOT_SUPPORTED -- invalid key length
AES_MODE_NOT_SUPPORTED -- invalid cipher mode

4.2 VIA PadLock SDK API

The VIA PadLock SDK API can be sorted into the following groups:

- RNG APIs
- ACE Normal & plain APIs
- ACE Aligned memory APIs
- ACE Fast & simple APIs

4.2.1 VIA PadLock SDK RNG APIs

int padlock_rng_available(void);

Description:

To test whether the VIA PadLock RNG hardware is available

Return:

1 -- if the VIA PadLock RNG hardware is available in VIA C5 based processor
0 -- if the VIA PadLock RNG hardware isn't available

Parameters:

none

RNG_RESULT padlock_rng_rand( );

Syntax:

RNG_RESULT padlock_rng_rand(unsigned char *rand, int rand_len);

Description:

Generate a random number using the VIA PadLock RNG hardware in VIA C5P based processor

Return:

RNG_SUCCEEDED -- random number generation operation succeeded
RNG_FAILED -- random number generation operation failed

Parameters:

rand,	the address to store the random number to be generated.
rand_len,.	the demanded length of the random number to be generated

The above APIs are illustrated in the demo program "VIA_Padlock_SDK_Demo,c" file as below:

```
// Demo Program for VIA Padlock SDK RNG APIs
#include “padlock.h”
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <string.h>
```

```
int main(void)
{
    int rng_available;
    unsigned char *random_num1[1024] = {0,};
    unsigned char *random_num2[1024] = {0,};
    … …
    rng_available = padlock_rng_available();
    if(!rng_available)
    {
        printf("VIA RNG hardware isn't available!\n");
        return -1;
    }

    padlock_rng_rand(random_num1, 1024);
    printf("VIA RNG generates random number 1 :\n");
    dump_rand(random_num1, 1024);

    padlock_rng_rand(random_num2, 1024);
    printf("VIA RNG generates random number 2 :\n");
    dump_rand(random_num2, 1024);

    if((!strcmp(random_num1, random_num2)))
       printf("\nVIA RNG random number generation function test failed!\n");
    return 0;
}
```

4.2.2 VIA PadLock SDK ACE normal & plain APIs

int padlock_ace_available(void);

Description:

To test whether VIA Padlock ACE hardware is available

Return:

1 -- if the VIA PadLock ACE hardware is available

0 – if the VIA PadLock ACE hardware is not is available

Parameters:

None

struct ace_aes_context *padlock_aes_begin(void);

Description:

To create an ACE AES context for later use

Return:

A pointer pointing to an ACE AES context

Parameters:

None

NOTE:

If it fails to create an ACE AES context the program will exit.

AES_RESULT padlock_aes_setkey( );

Syntax:

```
AES_RESULT padlock_aes_setkey(    struct ace_aes_context *ctx,
                                     const unsigned char *key,
                                  KEY_LENGTH key_len);
```

Description:

To set cipher key for later ACE AES cryptography operations

Return:

See AES_RESULT

Parameters:

ctx -- ACE AES Context pointer

key -- primary key data

key_len -- key length See KEY_LENGTH

AES_RESULT padlock_aes_setmodeiv( );

Syntax:

```
AES_RESULT padlock_aes_setmodeiv(    struct ace_aes_context *ctx,
                                   ACE_AES_MODE mode,
                                              unsigned char *iv);
```

Description:

To set cipher mode and initialization vector for later ACE AES cryptography operations

Return:

See AES_RESULT

Parameters:

ctx -- ACE AES Context pointer
mode -- AES cryptography mode See ACE_AES_MODE
iv -- the initialization vector for AES cryptography

NOTE:

1. The ECB mode cryptography doesn't need initialization vector. Assign iv with NULL,
2. Data stored in iv will be updated after later encryption or decryption. Reserve the original value of iv before encryption or decryption. For details, refer to the demo program below.

AES_RESULT padlock_aes_encrypt( );

Syntax:

```
AES_RESULT padlock_aes_encrypt(   struct ace_aes_context *ctx,
                                  unsigned char * plaintxt,
                                  unsigned char * ciphertxt,
                                  int nbytes);
```

Description:

To encrypt data stored in plaintxt and store the encryption result in cipertxt by the VIA PadLock ACE hardware

Return:

See AES_RESULT

Parameters:

ctx -- ACE AES Context pointer
plaintxt -- address where data to be encrypted is stored
ciphertxt -- address where result of encryption will be stored
nbytes -- total number of data to be encrypted in byte

NOTE:

1. The VIA PadLock ACE hardware can only process data whose length in byte is multiples of 16 bytes, please make sure nbytes is multiples of 16 or it will return with AES_NOT_BLOCKED.
2. The VIA PadLock ACE hardware can process data stored in 16 bytes aligned address directly. Though if they are not aligned with 16 bytes, you can still get the correct result but the efficiency of this situation is far lower than that of the situation of aligned addresses. It is highly recommended that plaintext and ciphertxt are aligned with 16 bytes.

AES_RESULT padlock_aes_decrypt();

Syntax:

```
AES_RESULT padlock_aes_decrypt(struct ace_aes_context *ctx,
                               unsigned char * ciphertxt,
                               unsigned char * plaintxt,
                               int nbytes);
```

Description:

To decrypt data stored in cipertxt and store the decryption result in plaintxt by VIA PadLock ACE hardware

Return:

See AES_RESULT

Parameters:

ctx	-- ACE AES context pointer
ciphertxt	-- address where data to be decrypted is stored
plaintxt	-- address where result of decryption will be stored
nbytes	-- total number of data to be decrypted in byte

NOTE:

1. The VIA PadLock ACE hardware can only process data whose length in byte is multiples of 16 bytes. Make sure nbytes is multiples of 16 or it will return with AES_NOT_BLOCKED
2. The VIA PadLock ACE hardware can process data stored in 16 bytes aligned address directly, though if they are not aligned with 16 bytes, you can still get the correct result but the efficiency of this situation is far lower than the situation of aligned addresses. It is highly recommended that plaintext and ciphertxt are aligned with 16 bytes.

```
// Demo Program for VIA Padlock SDK ACE normal & plain APIs
#include “padlock.h”
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <string.h>

// Standard Testing Vector from NIST for CBC mode 192bits key AES Cryptography
static int
cbc_aes192_key[6] =        {    0xf7b0738e, 0x52640eda, 0x2bf310c8, 0xe5799080,
                                0xd2eaf862, 0x7b6b2c52};
… …

static int
cbc_aes_iv[4] =            {    0x03020100, 0x07060504, 0x0b0a0908, 0x0f0e0d0c };
static int
cbc_aes_plain[16] =        {    0xe2bec16b, 0x969f402e, 0x117e3de9, 0x2a179373, \
                                0x578a2dae, 0x9cac031e, 0xac6fb79e, 0x518eaf45, \
                                0x461cc830, 0x11e45ca3, 0x19c1fbe5, 0xef520a1a, \
                                0x45249ff6, 0x179b4fdf, 0x7b412bad, 0x10376ce6 };
… …

static int
cbc_aes192_cipher[16] = {       0xb21d024f, 0x3d63bc43, 0x3a187871, 0xe871a09f, \
                                0xa9add9b4, 0xf4ed7dad, 0x7638e7e5, 0x5a14693f, \
                                0x20241b57, 0xe07afb12, 0xacbaa97f, 0xe002f13d, \
                                0x79e2b008, 0x81885988, 0xe6a920d9, 0xcd15564f };
static unsigned char scratch[64] = {0,};
static int
test_padlock_aes(ACE_AES_MODE mode, KEY_LENGTH key_len, int nbytes)
{
        struct ace_aes_context *ctx;
        char cipher_mode[10]= {0,};
        char cipher_key[10] = {0,};

        unsigned char *p_key;
        unsigned char *p_plain;
        unsigned char *p_cipher;
        unsigned char temp_iv[16]={0,};
        unsigned char orig_iv[16]={0,};
        AES_RESULT res;

… …

         // create the ace aes cipher context
        ctx = padlock_aes_begin();

        // set cipher key
        res = padlock_aes_setkey( ctx, p_key, key_len);
        if(res != AES_SUCCEEDED)
                … …

        // because the iv will be updated by the plain aes encryption and decryption API
        // here we reserve it in the orig_iv
        memcpy(temp_iv, orig_iv, 16);

        // set cipher mode and iv
        res = padlock_aes_setmodeiv( ctx, mode, temp_iv);
        if(res != AES_SUCCEEDED)
                … …

        // call ACE plain aes encryption API
        res = padlock_aes_encrypt( ctx, p_plain, scratch, nbytes);
        if(res != AES_SUCCEEDED)
                … …
        … …
        // verify the result of encryption
        if (memcmp (scratch, p_cipher, nbytes))
                 printf("%s mode plain AES-%s encryption test
                                  failed.\n",cipher_mode,cipher_key);
         else
                printf("%s mode plain AES-%s encryption test ok\n",cipher_mode,cipher_key);
```

```
    // because the temp_iv has been updated we have to restore orig_iv to it here
    memcpy(temp_iv, orig_iv, 16);
    // reset iv for decrytion
    res = padlock_aes_setmodeiv( ctx, mode, temp_iv);
    if(res != AES_SUCCEEDED)
        … …
    // call ACE plain aes decryption API
    res = padlock_aes_decrypt( ctx, p_cipher, scratch, nbytes);
    if(res != AES_SUCCEEDED)
        … …
    … …
    // verify the result of decryption
    if (memcmp (scratch, p_plain, nbytes))
         printf("%s mode plain AES-%s decryption test
                failed.\n",cipher_mode,cipher_key);
    else
        printf("%s mode plain AES-%s decryption test ok\n",cipher_mode,cipher_key);

    // destroy the ace aes cipher context
    padlock_aes_close( ctx);
    … …
}
```

4.2.3 VIA PadLock SDK ACE aligned memory APIs

The VIA PadLock ACE hardware can process data with 16 bytes aligned addresses (key iv plain text and cipher text etc), and it is more efficient. This API is provided to facilitate user to enhance the performance of their applications. There are optional APIs.

struct aligned_memory_context *padlock_aligned_malloc(int size);

Description:

Allocate 16 bytes aligned memory, which will be used as cipher buffer

Return:

A pointer pointing to an aligned memory context.

Parameters:

size -- the size of aligned memory .

NOTES:

The memory allocated by this function must be freed by the function

padlock_aligned_mfree ().

void padlock_aligned_mfree(struct aligned_memory_context *aligned_mctx);

Syntax:

void padlock_aligned_mfree(struct aligned_memory_context *aligned_mctx);

Description:

Free a memory block allocated by padlock_aligned_malloc()

Return:

none.

Parameter:

aligned_mctx -- the address of aligned memory context to be freed.

NOTES:

If it fails to allocate aligned memory context, the program will exit.

AES_RESULT padlock_aligned_memcpy_to();

Syntax:

```
AES_RESULT   padlock_aligned_memcpy_to(
                        struct aligned_memory_context *aligned_mctx,
                        unsigned char *src,
                        int nbytes);
```

Description:

Copy data to be encrypted or decrypted into the aligned memory context

Return:

See AES_RESULT

AES_SUCCEEDED – copy succeeded

AES_FAILED -- nbytes is larger than the size of aligned memory context or nbytes equals zero

AES_NOT_BLOCKED – because the VIA PadLock ACE hardware can only process data with 16 bytes multiple sized data, it is invalid to copy odd bytes data to aligned memory context

Parameter:

aligned_mctx -- the address of aligned memory context to be freed.

src -- the address of data to be processed

nbytes -- the total number of data to be processed in byte

AES_RESULT padlock_aligned_memcpy_from();

Syntax:

```
ALIGNED_MEMORY_RESULT     padlock_aligned_memcpy_from(
                          struct aligned_memory_context *aligned_mctx,
                          unsigned char *dst,
                          int nbytes);
```

Description:

Copy the result of encryption or decryption from the aligned memory context

Return:

See AES_RESULT

AES_SUCCEEDED -- copy succeeded

AES_FAILED -- nbytes is larger than the size of aligned memory context

Parameter:

aligned_mctx -- the address of aligned memory context to be freed.

dst -- the address of result of cryptography to be stored

nbytes -- the total number of data to be moved in byte

AES_RESULT padlock_aes_aligned_encrypt();

Syntax:

```
AES_RESULT padlock_aes_aligned_encrypt(   unsigned char *key,
                                          KEY_LENGTH key_len,
                                          ACE_AES_MODE mode,
                                         struct aligned_memory_context
*buf_aligned_mctx,
                                         unsigned char *iv);
```

Description:

To encrypt data stored in aligned memory context and store the encryption result back to the aligned memory context by the VIA PadLock ACE hardware.

Return:

See AES_RESULT

Parameters:

key -- primary key data

key_len -- length of key See KEY_LENGTH

mode -- cryptography mode See ACE_AES_MODE

buf_aligned_mctx -- the aligned memory context used as cipher buffer

iv -- initialization vector

NOTES:

1. The total number of data to be encrypted or decrypted in byte is nbytes in function

padlock_aligned_memcpy_to()

2. Data stored in iv will be updated after later encryption or decryption, please reserve the original value of iv before encryption or decryption. For details, please refer to the following demo program.

AES_RESULT padlock_aes_aligned_decrypt();

Syntax:

AES_RESULT padlock_aes_aligned_decrypt(unsigned char *key,
KEY_LENGTH key_len,
ACE_AES_MODE mode,
struct aligned_memory_context
***buf_aligned_mctx,**
unsigned char *iv);

Description:

To decrypted data stored in aligned memory context and store the decryption result back to aligned memory context by the VIA PadLock ACE hardware.

Return:

See AES_RESULT

Parameters:

key	-- primary key data
key_len	-- length of key See KEY_LENGTH
mode	-- cryptography mode See ACE_AES_MODE
buf_aligned_mctx	-- the aligned memory context used as cipher buffer
iv	-- initialization vector

NOTES:

1. The total number of data to be encrypted or decrypted in bytes is nbytes in function padlock_aligned_memcpy_to()
2. Data stored in iv will be updated after later encryption or decryption, please reserve the original value of iv before encryption or decryption. For details, please refer to the following demo program.

```
// Demo Program for VIA Padlock SDK ACE aligned memory APIs
// Here we used the above-mentioned standard testing vectors and global variables
static int
test_padlock_aligned_aes(ACE_AES_MODE mode, KEY_LENGTH key_len, int nbytes)
{
    struct aligned_memory_context *buf_aligned_mctx;

    char cipher_mode[10]= {0,};
    char cipher_key[10] = {0,};
```

```
unsigned char *p_key;
unsigned char *p_plain;
unsigned char *p_cipher;
unsigned char temp_iv[16]={0,};
unsigned char orig_iv[16]={0,};
AES_RESULT res;

… …
// create address aligned buffer context
buf_aligned_mctx = padlock_aligned_malloc(nbytes);

// copy data to be encrypted to aligned buffer
res = padlock_aligned_memcpy_to(buf_aligned_mctx, p_plain, nbytes);
if(res != AES_SUCCEEDED)
    … …
// Because the iv will be updated by the plain aes encryption and decryption API
// here we reserve it in the orig_iv
memcpy(temp_iv, (unsigned char *)cbc_aes_iv, 16);

// call ACE aligned aes encryption API
res = padlock_aes_aligned_encrypt(p_key, key_len, mode,
                                  buf_aligned_mctx, temp_iv);
if(res != AES_SUCCEEDED)
    return -1;
… …
// copy the encrypted result from aligned buffer to scratch
res = padlock_aligned_memcpy_from(buf_aligned_mctx, scratch, nbytes);
if(res != AES_SUCCEEDED)
    … …
// Verify the result of encryption
if (memcmp (scratch, p_cipher, nbytes))
    printf("%s mode aligned AES-%s encryption test
            failed.\n",cipher_mode,cipher_key);
else
    printf("%s mode aligned AES-%s encryption test ok\n",cipher_mode,cipher_key);

// copy data to be decrypted to aligned buffer
res = padlock_aligned_memcpy_to(buf_aligned_mctx, p_cipher, nbytes);
if(res != AES_SUCCEEDED)
    … …
// because the temp_iv has been updated   we have to restore orig_iv to it here
memcpy(temp_iv, orig_iv, 16);

// call ACE aligned aes decryption API
```

```
    res = padlock_aes_aligned_decrypt(p_key, key_len, mode,
                                      buf_aligned_mctx, temp_iv);
    if(res != AES_SUCCEEDED)
        … …
    … …
    // copy the decrypted result from aligned buffer to scratch
    res = padlock_aligned_memcpy_from(buf_aligned_mctx, scratch, nbytes);
    if(res != AES_SUCCEEDED)
        … …

    // verify the result of decryption
    if (memcmp (scratch, p_plain, nbytes))
        … …
    // destroy the aligned buffer context
    padlock_aligned_mfree( buf_aligned_mctx);
    … …
}
```

4.2.4 VIA PadLock SDK ACE fast & simple APIs

Though the efficiency of the VIA PadLock SDK ACE aligned memory APIs is satisfactory, it is too lengthy to use so many APIs. In order to simplify it, we provide an optional simple APIs

AES_RESULT padlock_aes_fast_encrypt( );

Syntax:

```
AES_RESULT padlock_aes_fast_encrypt( unsigned char *key, KEY_LENGTH key_len,
                                     ACE_AES_MODE mode,
                                     unsigned char *aligned_plaintxt,
                                     unsigned char *aligned_ciphertxt,
                                     int nbytes,
                                     unsigned char *iv);
```

Description:

To encrypted data stored in aligned_plaintxt and store the encryption result back to aligned_ciphertxt by the VIA PadLock ACE hardware.

Return:

See AES_RESULT

Parameters:

key -- primary key data
key_len -- length of key See KEY_LENGTH
mode -- cryptography mode See ACE_AES_MODE
aligned_plaintxt -- the aligned memory used to store data to be encrypted
aligned_ciphertxt -- the aligned memory used to store the encryption result
nbytes -- the total number of data to be encrypted in byte
iv -- initialization vector

NOTES:

1. The VIA PadLock ACE hardware can only process data whose length in bytes is multiples of 16 bytes, and make sure nbytes is multiples of 16 or it will return with AES_NOT_BLOCKED
2. If neither aligned_plaintxt or aligned_ciphertxt is16 bytes aligned, the API will do nothing and return AES_ADDRESS_NOT_ALIGNED
3. Data stored in iv will be updated after the encryption or decryption, reserve the original value of iv before encryption or decryption. For more details, refer to the demo program below.

AES_RESULT padlock_aes_fast_decrypt( );

Syntax:

```
AES_RESULT padlock_aes_fast_decrypt( unsigned char *key, KEY_LENGTH key_len,
                                     ACE_AES_MODE mode,
                                     unsigned char *aligned_ciphertxt,
                                     unsigned char *aligned_plaintxt,
                                     int nbytes,
                                     unsigned char *iv);
```

Description:

To decrypt data stored in aligned_ciphertxt and store the decryption result back to aligned_plaintxt by VIA ACE hardware

Return:

See AES_RESULT

Parameters:

key -- primary key data
key_len -- length of key See KEY_LENGTH
mode -- cryptography mode See ACE_AES_MODE
aligned_ciphertxt -- the aligned memory used to store data to be decrypted
aligned_plaintxt -- the aligned memory used to store the decryption result
nbytes -- the total number of data to be decrypted in byte
iv -- initialization vector

NOTES:

1. Because the VIA PadLock ACE hardware can only process data whose length in bytes is multiples of 16 bytes, please make sure nbytes is multiples of 16, or it will return with AES_NOT_BLOCKED
2. If either aligned_plaintxt or aligned_ciphertxt are not 16 bytes aligned in fact, API will do nothing and return AES_ADDRESS_NOT_ALIGNED
3. Data stored in iv will be updated after later encryption or decryption, please reserve the original value of iv before encryption or decryption. For details, please refer to the following demo program.

```
// Demo Program for VIA Padlock SDK ACE fast & simple APIs
// Here we used the above-mentioned standard testing vectors and global variables
static int
test_padlock_fast_aes(ACE_AES_MODE mode, KEY_LENGTH key_len, int nbytes)
{
    unsigned char *p_temp_buf;
    unsigned char *p_aligned_buf;

    char cipher_mode[10]= {0,};
    char cipher_key[10] = {0,};

    unsigned char *p_key;
    unsigned char *p_plain;
    unsigned char *p_cipher;
    unsigned char temp_iv[16]={0,};
    unsigned char orig_iv[16]={0,};

    AES_RESULT res;


    … …
    // malloc temporary buffer for later use
    p_temp_buf = (unsigned char *)malloc((nbytes + 16));
    if(p_temp_buf == NULL)
        … …
```

```
// get address aligned buffer from temporary buffer
p_aligned_buf = (unsigned char *)((((unsigned long)p_temp_buf) + 15 )&(~15UL));

// copy the data to be encrypted to aligned buffer
memcpy(p_aligned_buf, p_plain,nbytes);

// because the temp_iv will be updated
// we have to reserve it to orig_iv and copy it to temp_iv here
memcpy(temp_iv, orig_iv, 16);

// call ACE fast aes encryption API
res = padlock_aes_fast_encrypt(p_key, key_len, mode,
                                    p_aligned_buf, p_aligned_buf, nbytes, temp_iv);
if(res != AES_SUCCEEDED)
   … …
// copy the encryption result to scratch
memcpy(scratch, p_aligned_buf, nbytes);
… …
// verify the result of encryption
if (memcmp (scratch, p_cipher, nbytes))
     printf("%s mode fast AES-%s encryption test failed.\n",cipher_mode,cipher_key);
 else
    printf("%s mode fast AES-%s encryption test ok\n",cipher_mode,cipher_key);

// copy the data to be decrypted to aligned buffer
memcpy(p_aligned_buf, p_cipher, nbytes);

// because the temp_iv has been updated we have to restore orig_iv to it here
memcpy(temp_iv, orig_iv, 16);

//call ACE fast aes decryption API
res = padlock_aes_fast_decrypt(p_key, key_len, mode,
                                     p_aligned_buf, p_aligned_buf, nbytes, temp_iv);
if(res != AES_SUCCEEDED)
    … …
// copy the decryption result to scratch
memcpy(scratch, p_aligned_buf, nbytes);
… …

// verify the result of decryption
if (memcmp (scratch, p_plain, nbytes))
     printf("%s mode fast AES-%s decryption test failed.\n",cipher_mode,cipher_key);
 else
    printf("%s mode fast AES-%s decryption test ok\n",cipher_mode,cipher_key);
    // free temporary buffer
    free( p_temp_buf);
    … …
}
```