PROGRAMMING GUIDE

VIA PadLock SDK

Version 3.10
December 29, 2008

Office:

VIA Technologies Incorporated
Taiwan Office:
1st Floor, No. 531
Chung-Cheng Road, Hsin-Tien
Taipei, Taiwan ROC
TEL: 886-2-2218-5452
FAX: 886-2-2218-5453
WWW: http://www.via.com.tw

REVISION HISTORY

Version	Date	Revision
2.02	4/27/07	Initial public release Content based on RD source document released on 4/24/2007 Updated format
2.10	7/1/08	Format updated Grammar and spelling updated Logo updated
3.10	20/11/08	Add support for x64 platform Add partial hash API Add note that buffer of source data need more bytes than the length.

TABLE OF CONTENTS

1 INTRODUCTION

This API and Programming guide is for users using the VIA PadLock SDK in VIA C5XL/C5P/C5Q/C5J/Nano series processors. The Advanced Cryptography Engine (ACE) in VIA C5P series processors or Advanced Cryptography Engine version 2 (ACE2) in VIA C5Q/C5J/Nano series processors implements the cryptographic functionality. The Random Number Generator (RNG) function in VIA C5XL/C5P/C5Q/C5J/Nano series processors implements the random number generating functionality. The PadLock Hash Engine (PHE) in VIA C5J/Nano series processors implements both SHA-1 algorithm and SHA-256 algorithm. The PadLock Montgomery Multiplier (PMM) in VIA C5Q/C5J series processors implements the Montgomery Multiplication algorithm.

The VIA PadLock SDK is comprised of four groups of APIs, which facilitates the building of security applications for use with VIA C5XL/C5P/C5Q/C5J/Nano series processors. It helps users to make full use of many advanced features in VIA C5 series processors, such as RNG, ACE or ACE2, PHE, and PMM in order to enhance the overall performance of their applications.

For more details regarding the VIA PadLock Security Suite, please refer to the *VIA PadLock Programming Guide*.

2 SYSTEM REQUIREMENTS

2.1 Hardware Requirements

VIA C5XL/C5P/C5Q/C5J/Nano series processors must be used when using the VIA PadLock SDK.

2.2 Software Requirements

The VIA PadLock SDK supports two software platforms: Microsoft Windows (including Windows CE 5.0) and Linux. Relevant development tools for the supported software platforms, such as the GNU Compiler Collection (GCC), GNU Make, Nasm, Microsoft Visual C++, and Microsoft eMbedded Visual C++, are also needed.

3 INSTALLATION AND USAGE

3.1 Installing VIA PadLock SDK

3.1.1 RSA Library

RSA sample code is provided:

1. To demonstrate how to integrate the VIA PadLock SDK Library.
2. To provide an RSA library with VIA PMM hardware acceleration.

VIA modified the original RSA library in the following two ways:

1. Replaced the ModExp function with PadLock SDK APIs.
2. Replaced the RNG function with PadLock SDK APIs.

Users can find the source code in **`rsa_lib\include\`** and **`rsa_lib\src\`** folders.

3.1.2 Linux System

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

1. Enter the directory
```
[user@localhost user]# cd sdk/project/VIA_Padlock_SDK_Library_Linux/
If running on 64-bit Linux
[user@localhost user]# cd sdk/project/VIA_Padlock_SDK_Library_Linux64/
```
2. Install the PadLock SDK
```
[user@localhost VIA_Padlock_SDK_Library_linux]# make install
If running on 64-bit Linux
[user@localhost VIA_Padlock_SDK_Library_linux64]# make install
```

3.1.3 Windows System

For Windows systems, to install VIA PadLock SDK library, set the proper library and directory configurations and include **`padlock.h`** in the 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

For Linux systems, it is very easy to use the VIA PadLock SDK. Users only need to include the **padlock.h** file in the application to make full use of the VIA PadLock SDK. For more details, please refer to the source code files in the demo program folder **demo**.

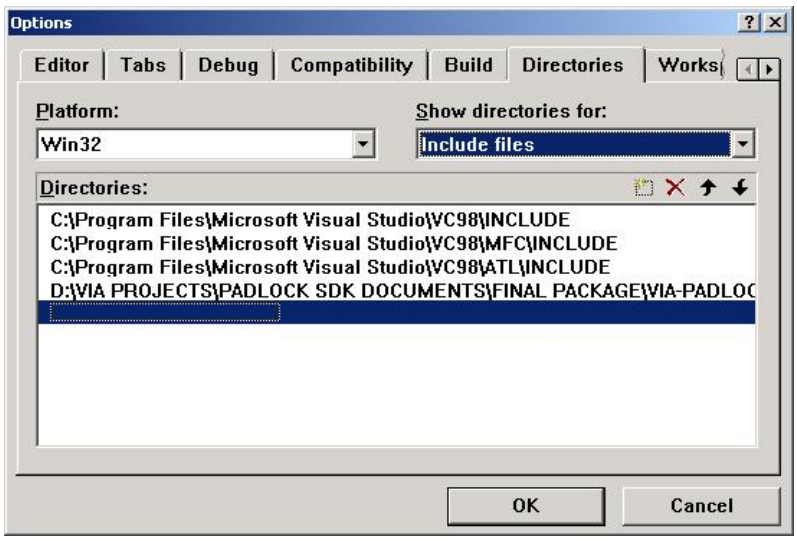
Users can find the **libvia_padlock.so** (PadLock SDK Library 3.10) and **padlock.h** files in the **release\linux** (or **release\linux64**) folder. Before using the library, be sure **libvia_padlock.so** is in the **user\lib** and **lib** folders.

Users can also customize the code in the VIA PadLock SDK in the following directories: **sdk\include** and **sdk\src**. Then change directories to the **sdk\project\VIA_Padlock_SDK_Library_Linux** (or **sdk\project\VIA_Padlock_SDK_Library_Linux64**)folder and run `make install`. Users can then use custom versions of VIA PadLock SDK in the application.

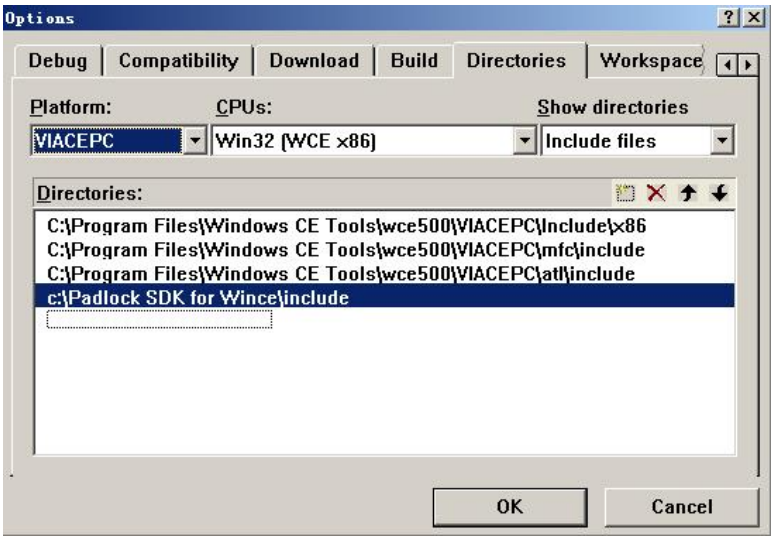
3.2.2 Windows System

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

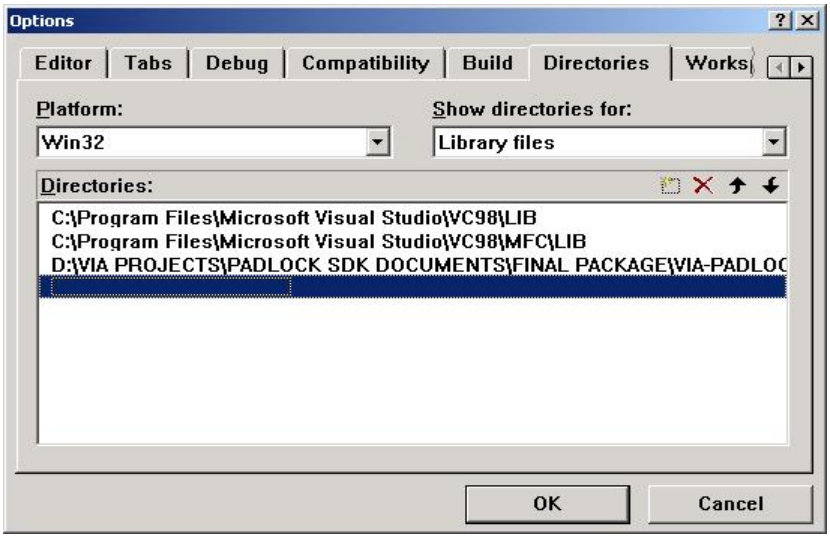
1. In **release\windows** (or **release\windows64, release\wince**) there should be three files in this directory: **padlock.h**, **VIA_Padlock_SDK_Library.dll**, and **VIA_Padlock_SDK_Library.lib**.
2. Create an application project in Visual C++ (or eMbedded Visual C++) and include **padlock.h**.
3. Click **Tools → Options...** to set the correct directory configurations.
    a. To add the **include** directory in Visual C++:



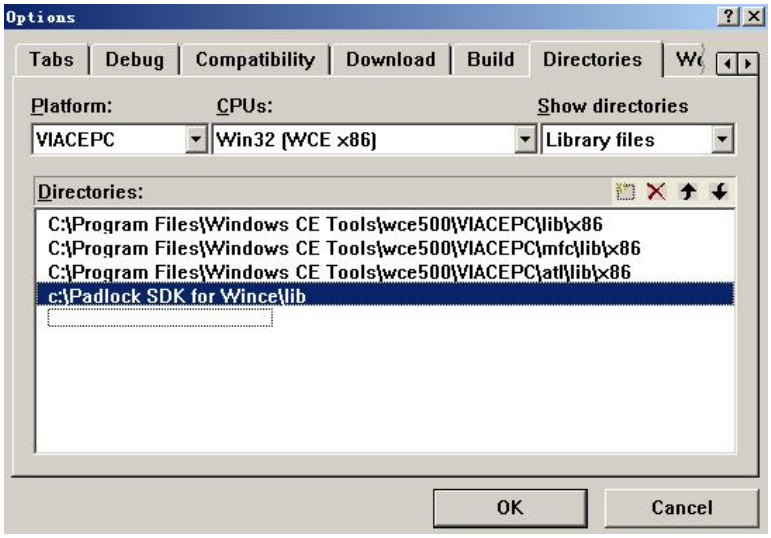
b. To add the **`include`** directory in eMbedded Visual C++:



c. To add the **`lib`** directory in Visual C++:

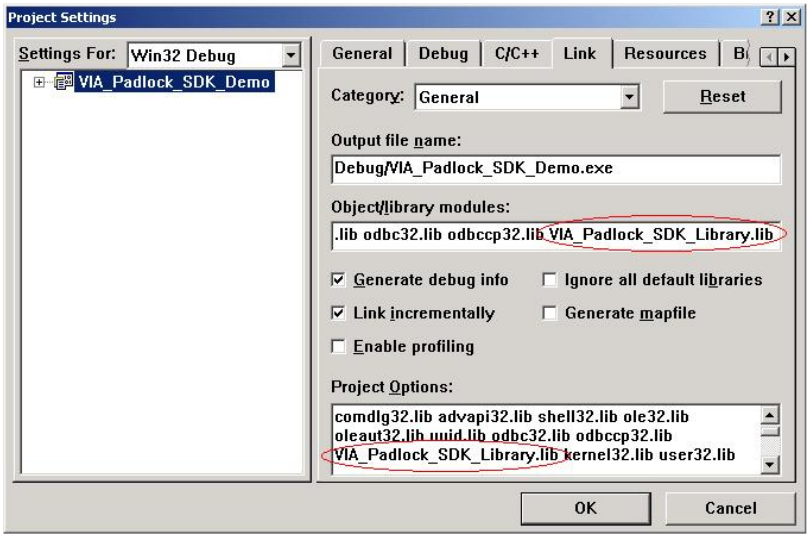


d. To add the **lib** directory in eMbedded Visual C++:

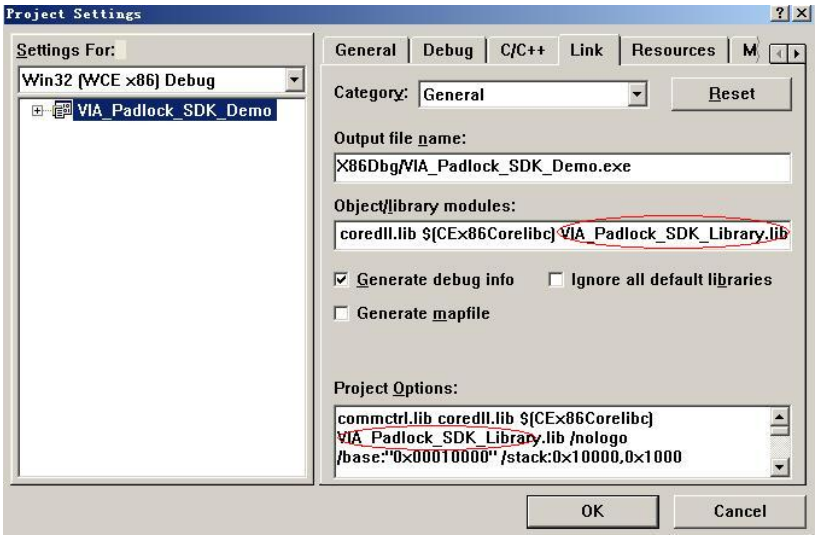


4. Click **Project → Setting...** to set the correct **lib** configuration.

a. In Visual C++:



b. In eMbedded Visual C++:



5. For WinCE system, users must download the **VIA_Padlock_SDK_Library.dll** to the WinCE system directory **\Windows**.
6. All APIs in the VIA PadLock SDK are ready for use. For more details, please refer to VIA_Padlock_SDK_Demo project in the **demo** folder.

Users can also customize the source code in the VIA PadLock SDK in the VIA_Padlock_SDK_Library project in the **sdk** folder. Then rebuild the SDK library and copy the relevant three files: **padlock.h**, **VIA_Padlock_SDK_Library.dll**, and **VIA_Padlock_SDK_Library.lib** to the VIA_Padlock_SDK_Library directory to use a custom version of the VIA PadLock SDK.

4 VIA PADLOCK SDK API

4.1 Type Definition

4.1.1 uint_32 type

Description:
`uint_32` is a data type used in the VIA PadLock SDK PMM APIs.

```
#ifndef uint_16
    typedef unsigned short uint_16;
#endif
#ifndef uint_32
    typedef unsigned int uint_32;
#endif
```

4.1.2 struct ace_aes_context

Description:
`ace_aes_context` is a parameter used in the VIA PadLock SDK ACE Normal & Plain APIs.

4.1.3 struct aligned_memory_context

Description:
`aligned_memory_context` is a parameter used in the VIA PadLock SDK ACE aligned memory APIs.

4.1.4 struct phe_sha_context

Description:
`phe_sha_context` is a parameter used in the VIA PadLock SDK PHE partial hash APIs.

4.1.5 RNG_RESULT

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

Description:
`RNG_RESULT` is the return value of the random number generating functions.

Parameter:
`RNG_SUCCEEDED` – RNG operation succeeded
`RNG_FAILED` – RNG operation failed

4.1.6 KEY_LENGTH

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

Description:

KEY_LENGTH is the key length type of the VIA ACE AES algorithm.

Parameter:

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

4.1.7 ACE_AES_MODE

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

Description:

ACE_AES_MODE is the cipher mode type of the VIA ACE AES algorithm.

Parameter:

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

4.1.8 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:

AES_RESULT is the return value of the VIA PadLock ACE APIs.

Parameter:

AES_ADDRESS_NOT_ALIGNED – the addresses of plaintxt, ciphertxt, or iv must be aligned by 16 bytes, which is required by the VIA ACE hardware.
AES_NOT_BLOCKED – the length of plaintxt or ciphertxt must be multiples of 16 bytes, which is required by the VIA ACE hardware.
AES_KEY_NOT_SUPPORTED – invalid key length
AES_MODE_NOT_SUPPORTED – invalid cipher mode

4.1.9 SHA_RESULT

```
typedef enum _SHA_RESULT{
    SHA_SUCCEEDED,
    SHA_FAILED
}SHA_RESULT;
```

Description:
SHA_RESULT is the return value of the VIA PadLock PHE operation.

Parameter:
SHA_SUCCEEDED – SHA operation succeeded
SHA_FAILED – SHA operation failed

4.2 Introduction to 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
- PHE APIs
- PMM APIs

4.2.1 VIA PadLock SDK RNG APIs

```
int padlock_rng_available( void);
```

Description:
padlock_rng_available is used to test whether the VIA RNG hardware is available.

Parameter:
none

Return Value:
1 – if VIA RNG hardware is available in VIA C5 series processors
0 – if the RNG hardware is not available

RNG_RESULT padlock_rng_rand( );

Syntax:

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

Description:

padlock_rng_rand generates a random number using the VIA RNG hardware in VIA C5 series processors.

Parameter:

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

Return Value:

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

The APIs mentioned in section 4.2.1 are illustrated in the demo program **VIA_Padlock_SDK_Demo_RNG,c** file shown 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 and Plain APIs

4.2.2.1 //PADLOCK.C'S API

`int padlock_ace_available( void);`

Description:
`padlock_ace_available` tests whether the VIA ACE hardware is available.

Parameter:
none

Return Value:
`1` – if VIA ACE hardware is available
`0` – if VIA ACE hardware is not available

4.2.2.2 //PADLOCK_AES.C'S API

`struct ace_aes_context *padlock_aes_begin(void);`

Description:
`ace_aes_context *padlock_aes_begin` creates an ACE AES context for later use.

Parameter:
none

Return Value:
A pointer pointing to an ACE AES context

`void padlock_aes_close();`

Syntax:

```
void padlock_aes_close( struct ace_aes_context *ctx );
```

Description:
`padlock_aes_close` destroys an ACE AES context.

Parameter:
`ctx` – ACE AES context pointer

Return Value:
none

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:

padlock_aes_setkey sets the cipher key for ACE AES cryptography operations.

Parameter:

ctx – ACE AES context pointer
key – primary key data
key_len – key length (See KEY_LENGTH)

Return Value:

See AES_RESULT

AES_RESULT padlock_aes_setmodeiv( );

Syntax:

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

Description:

padlock_aes_setmodeiv sets the cipher mode and initialization vector for ACE AES cryptography operations.

Parameter:

ctx – ACE AES context pointer
mode – AES cryptography mode (See ACE_AES_MODE)
iv – the initialization vector for AES cryptography

Return Value:

See AES_RESULT

Note:
1. The ECB mode cryptography does not need an initialization vector. Assign iv with NULL.
2. Data stored in iv will be updated after 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:

padlock_aes_encrypt encrypts data stored in plaintxt and stores the result in cipertxt using the VIA ACE hardware.

Parameter:

ctx – ACE AES context pointer

plaintxt – address where data to be encrypted is stored. User should allocate an additional 64 bytes if nbytes is larger than 16k

ciphertxt – address where result of encryption will be stored. User should allocate an additional 64 bytes if nbytes is larger than 16k

nbytes – total number of data to be encrypted in bytes

Return Value:

See AES_RESULT

Note:

1. VIA ACE hardware can only process data whose length in byte is multiples of 16 bytes, please make sure nbytes is in multiples of 16 or it will return with AES_NOT_BLOCKED.
2. VIA 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 plaintxt 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:

`padlock_aes_decrypt` decrypts data stored in `ciphertxt` and stores the result in `plaintxt` using the VIA ACE hardware.

Parameter:

`ctx` – ACE AES context pointer

`plaintxt` – address where data to be decrypted is stored. User should allocate an additional 64 bytes if nbytes is larger than 16k

`ciphertxt` – address where result of decryption will be stored. User should allocate an additional 64 bytes if nbytes is larger than 16k

`nbytes` – total number of data to be decrypted in bytes

Return Value:

See AES_RESULT

> **Note:**
> 1. VIA ACE hardware can only process data whose length in byte is in multiples of 16 bytes. Make sure `nbytes` is multiples of 16 or it will return with `AES_NOT_BLOCKED`.
> 2. VIA 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 `plaintxt` 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 ACE hardware can process data efficiently with 16 bytes of aligned addresses (`key`, `iv`, `plaintxt`, and `ciphertxt`, etc). This API facilitates the enhancement of application performance. Other optional APIs are also available.

The performance of aligned memory APIs may be lower than the Normal & Plain APIs with 16 bytes of aligned `plaintxt` and `ciphertxt`.

`Struct aligned_memory_context *padlock_aligned_malloc(int size);`

Description:
`aligned_memory_context *padlock_aligned_malloc` allocates 16 bytes of aligned memory to be used as a cipher buffer.

Parameter:
`size` – the size of the aligned memory

Return Value:
A pointer pointing to an aligned memory context

> **Note:**
> 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:
`padlock_aligned_mfree` frees a memory block allocated by `padlock_aligned_malloc()`.

Parameter:
`aligned_mctx` – the address of aligned memory context to be freed

Return Value:
none

> **Note:**
> 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:

padlock_aligned_memcpy copies data to be encrypted or decrypted into the aligned memory context.

Parameter:

aligned_mctx – the destination address of aligned memory context to copy to
src – the address of data to be processed
nbytes – the total number of data to be processed in bytes

Return Value:

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 – the VIA ACE hardware can only process data in multiples of 16 bytes; all other sizes of data are considered invalid

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:

padlock_aligned_memcpy_from copies the result of encryption or decryption from the aligned memory context.

Parameter:

aligned_mctx – the source address of aligned memory context to copy from
dst – the address of result of cryptography to be stored
nbytes – the total number of data to be moved in bytes

Return Value:

See AES_RESULT
AES_SUCCEEDED – copy succeeded
AES_FAILED – nbytes is larger than the size of aligned memory context

AES_RESULT padlock_aligned_memcpy_from();

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:

padlock_aligned_memcpy_from encrypts data stored in the aligned memory context and stores the result in the aligned memory context using the VIA ACE hardware.

Parameter:

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

Return Value:

See AES_RESULT

Note:
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:

padlock_aes_aligned_decrypt decrypts data stored in the aligned memory context and stores the result in the aligned memory context using the VIA ACE hardware.

Parameter:

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

Return Value:

See AES_RESULT

> **Note:**
> 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.

```
// 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 and Simple APIs

Though the efficiency of the VIA PadLock SDK ACE aligned memory APIs is satisfactory, it may be a bit difficult to use so many APIs. In order to simplify it, optional simple APIs are provided.

The performance of Fast & Simple APIs may be lower than the Normal & Plain APIs with 16 bytes of aligned `plaintxt` and `ciphertxt`.

`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:

`padlock_aes_fast_encrypt` encrypts data stored in `aligned_plaintxt` and stores the result in the `aligned_ciphertxt` using the VIA ACE hardware.

Parameter:

`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. User should allocate an additional 64 bytes if nbytes is larger than 16k
`aligned_ciphertxt` – the aligned memory used to store the encryption result. User should allocate an additional 64 bytes if nbytes is larger than 16k
`nbytes` – the total number of data to be encrypted in bytes
`iv` – initialization vector

Return Value:

See `AES_RESULT`

> **Note:**
> 1. The VIA ACE hardware can only process data in multiples of 16 bytes, otherwise it will return with `AES_NOT_BLOCKED`.
> 2. If neither `aligned_plaintxt` nor `aligned_ciphertxt` is 16 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:

padlock_aes_fast_decrypt decrypts data stored in aligned_ciphertxt and stores the result in the aligned_plaintxt using the VIA ACE hardware.

Parameter:

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 decrypted. User should allocate an additional 64 bytes if nbytes is larger than 16k
aligned_ciphertxt – the aligned memory used to store the decryption result. User should allocate an additional 64 bytes if nbytes is larger than 16k
nbytes – the total number of data to be decrypted in bytes
iv – initialization vector

Return Value:

See AES_RESULT

Note:
1. Because VIA ACE hardware can only process data in multiples of 16 bytes, otherwise it will return with AES_NOT_BLOCKED.
2. If either aligned_plaintxt or aligned_ciphertxt are not 16 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 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);
    … …
}
```

4.2.5 VIA PadLock SDK PHE APIs

int padlock_phe_available();

Syntax:

```
int padlock_phe_available();
```

Description:
padlock_phe_available tests whether the VIA PHE hardware is available.

Parameter:
none

Return Value:
1 – if VIA PHE hardware is available
0 – if VIA PHE hardware is unavailable

SHA_RESULT padlock_phe_sha1();

Syntax:

```
SHA_RESULT padlock_phe_sha1(unsigned char *src, int nbytes,unsigned char* dst);
```

Description:
padlock_phe_sha1 produces a 160-bit SHA-1 hash of the source using the VIA PHE hardware.

Parameter:
src – address of source string to be hashed. User should allocate an additional 64 bytes if nbytes is larger than 16k
nbytes – length of source string (in bytes)
dst – address of digest buffer to store the result

Return Value:
SHA_SUCCEEDED – PHE operation succeeded
SHA_FAILED – PHE operation failed

SHA_RESULT padlock_phe_sha256();

Syntax:

```
SHA_RESULT padlock_phe_sha256(unsigned char *src, int nbytes,unsigned char* dst);
```

Description:
`padlock_phe_sha256` produces a 256-bit SHA-256 hash of the source using the VIA PHE hardware.

Parameter:
`src` – address of source string to be hashed. User should allocate an additional 64 bytes if nbytes is larger than 16k
`nbytes` – length of source string (in bytes)
`dst` – address of digest buffer to store the result

Return Value:
`SHA_SUCCEEDED` – succeeded
`SHA_FAILED` – failed

int padlock_phe_partial_available();

Syntax:

```
int padlock_phe_partialavailable();
```

Description:
`padlock_phe_partial_available` tests whether the VIA PHE partial hash is available.

Parameter:
none

Return Value:
`1` – if VIA PHE partial hash is available
`0` – if VIA PHE partial hash is unavailable

struct phe_sha_context* padlock_phe_partial_sha1_init();

Syntax:

```
struct phe_sha_context* padlock_phe_partial_sha1_init();
```

Description:
`padlock_phe_partial_sha1_init` creates a SHA context.

Parameter:
none

Return Value:
A pointer pointing to a SHA context.

SHA_RESULT padlock_phe_partial_sha1_update();

Syntax:

```
SHA_RESULT padlock_phe_partial_sha1_update(struct phe_sha_context* ctx, unsigned char *src, int
nbytes);
```

Description:

padlock_phe_partial_sha1_update hashes the current block of data by SHA1.

Parameter:

ctx – current context
src – address of source string to be hashed
nbytes – length of source string (in bytes), it must be multiples of 64 bytes.

Return Value:

SHA_SUCCEEDED – succeeded
SHA_FAILED – failed

SHA_RESULT padlock_phe_partial_sha1_final();

Syntax:

```
SHA_RESULT padlock_phe_partial_sha1_final(struct phe_sha_context* ctx, unsigned char *src, int
nbytes, unsigned char* dst);
```

Description:

padlock_phe_partial_sha1_final hashes the last block of data by SHA1 and produces a 160-bit digest.

Parameter:

ctx – current context
src – address of source string to be hashed
nbytes – length of source string (in bytes)
dst – address of digest buffer

Return Value:

SHA_SUCCEEDED – succeeded
SHA_FAILED – failed

struct phe_sha_context* padlock_phe_partial_sha256_init();

Syntax:

```
struct phe_sha_context* padlock_phe_partial_sha256_init();
```

Description:

padlock_phe_partial_sha256_init creates a SHA context.

Parameter:

none

Return Value:

A pointer pointing to a SHA context.

SHA_RESULT padlock_phe_partial_sha256_update();

Syntax:

```
SHA_RESULT padlock_phe_partial_sha256_update(struct phe_sha_context* ctx, unsigned char *src, int
nbytes);
```

Description:

padlock_phe_partial_sha256_update hashes the current block of data by SHA256.

Parameter:

ctx – current context
src – address of source string to be hashed
nbytes – length of source string (in bytes) , it must be multiples of 64 bytes.

Return Value:

SHA_SUCCEEDED – succeeded
SHA_FAILED – failed

SHA_RESULT padlock_phe_partial_sha256_final();

Syntax:

```
SHA_RESULT padlock_phe_partial_sha256_final(struct phe_sha_context* ctx, unsigned char *src, int
nbytes, unsigned char* dst);
```

Description:

padlock_phe_partial_sha256_final hashes the last block of data by SHA256 and produces a 256-bit digest.

Parameter:

ctx – current context
src – address of source string to be hashed
nbytes – length of source string (in bytes)
dst – address of digest buffer

Return Value:

SHA_SUCCEEDED – succeeded
SHA_FAILED – failed

```
// Demo Program for VIA PadLock SDK SHA APIs
void sha256_test()
{
    int i, len;
        int phe_available;

    char hash[32];
    static char *str[]={
    "abc",
        "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"
        };
    phe_available = padlock_phe_available();
    if(!phe_available)
    {
        printf("VIA PHE hardware isn't available!\n");
        return;
    }
```

```
    for (i=0;i<2;++i)
    {
        len=strlen(str[i]);
        memset(hash,0,32);
        padlock_phe_sha256((unsigned char*)str[i], len, (unsigned char*)hash);
    }
}


void sha1_test()
{
    int i, len;
    char hash[20];
    static char *str[]={
    "abc",
        "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq"
        };

    for (i=0;i<2;++i)
    {
        len=strlen(str[i]);
        memset(hash,0,20);
        padlock_phe_sha1((unsigned char*)str[i], len, (unsigned char*)hash);
    }
}
void sha256_partial_test()
{
    int i,n,k;
    char hash[32];
    char* str[2]={
                        "abc",
                        "100 bytes
0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789"};
    char* compare[2]={
                        "ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad",
                        "b71f7fff653a759242a8e411fff0e407441a4a94f7833f91f9806c4468622d8e"
                        };
    struct phe_sha_context *ctx256 =  NULL;

    for(i=0;i<2;i++)
    {
        n = strlen(str[i]);

        ctx256 = padlock_phe_partial_sha256_init();
        if(ctx256==NULL)
            return;

        for(k=0;k<n/64;k++)
            padlock_phe_partial_sha256_update(ctx256,((unsigned char*)str[i])+k*64,64);

        padlock_phe_partial_sha256_final(ctx256,((unsigned char*)str[i])+k*64,n%64,(unsigned
char*)hash);

    }
}

void sha1_partial_test()
{
    int i,n,k;
    char hash[20];
    char* str[2]={
                        "abc",
                        "100 bytes
0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789"};
    char* compare[2]={
                                "a9993e364706816aba3e25717850c26c9cd0d89d",
                                "47cc0e6d3689acb6203aa96423cf7963eb25cc59"};
    struct phe_sha_context *ctx1 =    NULL;

    for(i=0;i<2;i++)
    {
        n = strlen(str[i]);

        ctx1 = padlock_phe_partial_sha1_init();
```

```
        if(ctx1==NULL)
            return;

        for(k=0;k<n/64;k++)
            padlock_phe_partial_sha1_update(ctx1,((unsigned char*)str[i])+k*64,64);

        padlock_phe_partial_sha1_final(ctx1,((unsigned char*)str[i])+k*64,n%64,(unsigned
char*)dst);

    }

}
```

4.2.6 VIA PadLock SDK PMM APIs

int padlock_pmm_available();

Syntax:

```
int padlock_pmm_available();
```

Description:

padlock_pmm_available tests whether the VIA Mont hardware is available.

Parameter:

none

Return Value:

1 – if VIA Mont hardware is available

0 – if VIA Mont hardware is unavailable

void padlock_pmm();

Syntax:

```
void padlock_pmm( uint_32* A,  uint_32* B,  uint_32* M,  uint_32* dst,  uint_32 ndigits);
```

Description:

padlock_pmm implements the Montgomery Multiplication algorithm.

Parameter:

A – pointer of big integer A

B – pointer of big integer B

M – pointer of big integer M, modulus

dst – pointer of big integer dst, result buffer

ndigits – the length of big integer A, B, M, dst, in uint_32

Return Value:

none

Note:

1. The length of big integer A, B, M, dst should be equal.
2. The length of big integer A, B, M, dst should be equal or larger than 256, and equal or less than 32768.
3. The length of big integer A, B, M, dst should be multiples of 128.
4. To calculate dst = A * B mod M with Montgomery Multiplier, additional software big integer functions should be provided(implemented) first to calculate the RM (Refer to the sample code to see how to calculate RM). The result of dst = A * B mod M can be get by the following calling sequence:

```
padlock_pmm (A,  RM,  M,  TempResult,  ndigits);
padlock_pmm(TempResult,  B,  M,  dst,  ndigits);
```

```
// Sample code of calculating RM
void padlock_montmul_get_rm( uint_32 *M, uint_32 nbits)
{
    /* calculate rm
        rm = temp * temp mod M, temp = 0x0001 0000 0000....0000,
        the length of temp = ((digits of M) + 1), or ((bytes of M) + 4).
    */

    uint_32 bytes = nbits / 8;
    uint_32 digits = length / 32;
    uint_32* rm = (uint32*)malloc(2 * bytes + 4);
    uint_32* tmp = (uint32*)malloc(bytes + 4);
    memset((unsigned char *)rm, 0, 2 * bytes + 4);
    memset((unsigned char *)tmp, 0, bytes + 4);
    tmp[digits] = 1;

    /* first, do rm = tmp * tmp,
    using big integer function: result = a * b
        Big_Int_Mult(uint_32 *a, uint_32 *b,  uint_32 *result, uint_32 digits))
    */
    Big_Int_Mult(rm, tmp, tmp, digits + 1);

    /* then, get rm = rm mod M
        using big integer function: result = A mod M
        Big_Int_Mod(uint_32* result, uint_32 *A, uint_32 *digitsA, uint32* M, uint32 digitsM))
    */
    Big_Int_Mod(rm, rm, 2 * digits + 1, M, digits);
}
```