

KEYLOK[®]
*Software Piracy
Prevention System*
**CODE ON DONGLE
MANUAL**



KEYLOK[®] FORTRESS

Copyright

©Copyright 2010- All Rights Reserved. This documentation and the accompanying software are copyrighted materials. Making unauthorized copies is prohibited by law.

Trademarks

MAI Digital Security owns a number of registered and unregistered Trademarks and Service Marks (the "Marks"). These Marks are extremely valuable to MAI Digital Security and shall not be used by you, or any other person, without MAI Digital Security's express written permission. The Marks include, but are not necessarily limited to the following: KEYLOK®; S-LOK™; COMPU-LOCK™; the KEYLOK® logo and the MAI Digital Security Logo.

S-LOK™ is a joint trademark of Microcomputer Applications, Inc. and A.S.M. Inc.

You shall not use any of the Trademarks or Service Marks of MAI Digital Security without the express written permission of such Trademark or Service Mark owner.

MAI Digital Security (KEYLOK®)

1025 W 7th Ave
Denver, CO 80204 USA
Phone: (720) 904-2252 or 1-800-4KEYLOK (1-800-453-9365)•
Fax: (303) 228-0285 or (720) 294-0275
Email: info@keylok.com • Web: www.keylok.com

Last updated 1/30/10

Contents

Introduction	7
General Product Overview	8
Fortress Overview	9
Why Does Code on The Dongle Matter For Software Protection?	9
How It Works?	9
Implementation Overview	10
Code Selection	10
Directory and File Structure	11
Development Process	11
KEXEC – Interfacing with Your Code on the Dongle	12
KEYLOK Fortress Functions	14
Flow Control	14
_exit	14
Input/Output	14
_set_response	14
File Operation	15
_open	15
_close	15
_read	15
_write	16
_create	16
_enable_exe	17
Mathematics	17
_addf	17
_add	18
_subf	18
_sub	18
_multf	18
_mult	19
_divf	19
_div	19
_sinf	19
_sin	20
_cosf	20
_cos	20
_tanf	20
_tan	21
_asinf	21
_asin	21
_acosf	21
_acos	21
_atanf	22

_atan.....	22
_atanf2f.....	22
_atan2.....	22
_sinhf.....	23
_sinh.....	23
_coshf.....	23
_cosh.....	23
_tanhf.....	24
_tanh.....	24
_ceilf.....	24
_ceil.....	24
_floorf.....	24
_floor.....	25
_absf.....	25
_abs.....	25
_fmodf.....	25
_fmod.....	26
_expf.....	26
_exp.....	26
_logf.....	26
_log.....	27
_log10f.....	27
_log10.....	27
_sqrtf.....	27
_sqrt.....	27
_powf.....	28
_pow.....	28
_modf.....	28
_frexp.....	29
_ldexp.....	29
_fdcmp.....	29
_dtof.....	29
_ftod.....	30
_dtol.....	30
_altod.....	30
_lltod.....	30
Cyptographic Algorithms	31
_tdes_enc.....	31
_tdes_dec.....	31
_des_enc.....	32
_des_dec.....	32
_sha1_init.....	33
_sha1_update.....	33
_sha1_final.....	34
_rsa_enc.....	34
_rsa_dec.....	35
_rsa_sign.....	35

_rsa_veri	36
_rand	36
Memory Operations	37
_mem_copy	37
_mem_move	37
_mem_set.....	37
_mempool_init.....	38
_malloc.....	38
_calloc	38
_realloc.....	39
_free	39
_invert.....	39
_mem_cmp	39
Time Functions	40
_set_timer	40
_start_timer	40
_stop_timer.....	40
_get_timer	41
Macro and Auxilliary Functions.....	41
_swap_u16.....	41
_swap_u32.....	41
LE16_TO_CC	41
LE32_TO_CC	42
CC_TO_LE16	42
CC_TO_LE32	42
BE16_TO_CC.....	42
BE32_TO_CC.....	42
CC_TO_BE16.....	43
CC_TO_BE32	43
_atod.....	43
DEFINE_AT.....	43
Device Info Functions.....	44
_get_gbdata	44
Distributing Your Application.....	45
Integrating the KEYLOK® API	45
KEYLOK® Utility Programs	45
Fortress Management Utility.....	45
Sample Code.....	45
KEYLOK® Sample Code.....	45
Appendix A Cryptographic Algorithms	48
A.1. Key Operating Functions	48
A.1.1. X_GenerateRsaKeys	48
A.1.2. R_GenerateRsaKeys	49
A.1.3. X_Pub2Cos	49

A.1.4. X_Pri2Cos	50
A.1.5. X_Cos2Pub	50
A.1.6. X_Cos2Pri	50
A.2. Cryptographic Algorithm Functions	51
A.2.1. RSAPublicEncrypt.....	51
A.2.2. RSAPrivateDecrypt.....	51
A.2.3. Sign	52
A.2.4. Verify	52
A.2.5. Digest	53
A.2.6. DES	53
A.2.7. TDES.....	54
A.3. Error Code Index.....	54

Introduction

KEYLOK® is proud to offer the most complete solution to software security in the marketplace. KEYLOK® pioneered dongle based software piracy prevention in 1980, and has continued to lead the industry throughout the history of software protection.

Our product offerings include:

KEYLOK® II: The KEYLOK® security system provides a very high degree of security with an economical hardware key. KEYLOK® devices are available for parallel, USB and serial ports on computers running DOS, WINDOWS 3.x / 9x / ME / NT / 2000 / XP / Vista / 7 / Server2003 / Server2008 / Server 2008 R2, LINUX, FreeBSD, QNX or Macintosh OS X. Serial port dongles can be used on any computer with an RS232 port and do not require a device driver. KEYLOK® devices have programmable memory, remote update, lease control algorithms, and networking capability for controlling multiple users with a single hardware lock.

FORTRESS: Backwards compatible to KEYLOK® II, the USB dongle also offers two distinct differences. Firstly, the dongle operates in HID mode: you do NOT need to install a device driver. And secondly, Fortress allows you to migrate functions from your code to execute only on the dongle providing you with an unparalleled security solution. Fortress also provides larger user memory (5,120 – 51,000 bytes) and provides an added level of tamper resistance, increasing the protection against piracy through reverse engineering.

S-LOK™: S-LOK™ is a special version of KEYLOK® II that is capable of “wrapping” a protected shell around a program executable without having to make source code modifications. S-LOK™ is currently compatible with only parallel port dongles.

Throughout this manual the KEYLOK® II product will be referred to generically as KEYLOK®. Various models are available to support USB, parallel and serial connections. Software interfaces to the security system and associated demonstration programs are available for most programming languages and development environments. A demonstration program prepared in the language that you are using makes implementation within your program quite simple. Robust protection algorithms defend against determined pirates attempting to bypass security.

General Product Overview

The KEYLOK® security system protects your software applications from piracy, thereby increasing your revenues associated with software sales. The security is transparent to your end-user once the hardware device is installed on the computer's USB, parallel or serial port. Unlimited backup copies of the program can be made to protect your clients with the knowledge that for each copy of your software sold only one copy can be in use at any one time. Your clients can install the software on multiple machines (e.g. at the office and at home) without having to go through difficult and time-consuming install/uninstall operations. Your clients can easily restore or reinstall copies of your software following catastrophic events such as hard disk failures. These advantages provide your clients with the features they desire and deserve while preserving your financial interests.

The KEYLOK® security system uses a number of sophisticated techniques for verification of hardware device presence. The KEYLOK® is also provided with 112 bytes (5,120 bytes for Fortress) of field programmable read/write memory. There are NO special programming adapters required to program the dongle memory.

When first attempting to communicate with the hardware device it is necessary to successfully complete an exchange of bytes between the host and the device that differs during each device use (i.e., using an active algorithm). If this sequence is properly executed the device will return a customer unique 32-bit identification code which you can use as confirmation that one of your hardware security devices is installed on the computer. If an improper data exchange occurs then the security system returns to the host a random 32-bit code in place of the proper identification code. Upon successful completion of the authentication sequence, the host computer then sends a 48-bit customer unique password to the device to authorize memory read operations. Memory write operations require the successful transmission of yet another 48-bit customer unique password to the device. The write password must be sent AFTER transmission of the proper READ authorization sequence. If the device is sent the incorrect read/write password then subsequent memory operations are ignored and random information is returned to the program. In summary, a total of 176 bits of customer unique codes must be known in order to alter the memory within the dongle.

Up to fifty-six (56) separate 16-bit counters (values of 0-65535) can be independently maintained within the device. Counters are particularly useful for controlling demonstration copies of software, as well as pay-per-use software (e.g. testing). Some clients use the counter as a means of controlling software use up until the time they have been paid for the software, and then provide their clients a special code that 'unlocks' the device for unlimited future use.

At the time of manufacturing each device is programmed with a unique device serial number, thus providing you the capability of identifying the specific device (and thus specific end-user) for customers requiring this level of control.

The security system includes algorithms for performing very secure remote memory modifications to the device. The remote update procedure involves the exchange of encrypted information between the software vendor and the end-user. Remote update can be used to perform field updates of the dongle to support your licensing and copy protection needs. See the Remote Update section of this manual for a more thorough explanation of the available options.

Expiration date functions are available on all devices to support leasing and rental licensing methods.

The KEYLOK® price-to-feature ratio is unparalleled in the industry.

Fortress Overview

The core of the Fortress device is the Smart Card chip. The design of this integrated circuit chip provides a secure computing platform, complete with processor, I/O controller, operating system, memory and storage. The security features are built into the chip's operating system which has been designed to provide the highest levels of security currently available and is certified by international standards. Based on these standards, ISO has formulated the international criteria EAL (Evaluation Assurance Level) for the evaluation of the security product. EAL is divided into 1-7 grades, the higher of the grade, the more secure it will be. The scientific and technical product with the highest grade of security in the world can only reach EAL5+ currently. The most advanced PHILIPS 16-bit smart card chip used within Fortress has reached the highest grade in the global hi-tech sector: EAL 5+.

The Fortress file system is organized similar to your typical microcomputer operating system. There is a root directory and sub directories. Each directory has its own level of security. The files stored on the device cannot be read directly. All access to the directory structure is provided by the card operating system. Three types of files are supported; executable files, data files and key files. The data files can only be accessed by the executable files loaded onto the device. Key files contain the RSA key for encryption purposes.

Fortress has built-in support for RSA, DES/TDES and SHA-1 cryptographic algorithms. All communication between the application and the device and all data stored on the device can be encrypted.

Why Does Code on The Dongle Matter For Software Protection?

Traditional software protection methodologies utilize dongles which provide a base level of security and encryption functions. Although effective, history has shown these methodologies will fail over time based upon the perseverance of the hacker. The typical solution is solely dependent upon the software engineer responsible for implementing software protection and his/her knowledge of the vendor solution and there is an obvious knowledge gap between this person and the professionals focused on decrypting protection.

The Smart Card solution is a complete paradigm shift. Firstly, international standards certify the security features of the device which was never available before from any of the dongle vendors. These features are something you can depend upon. Secondly, you now have the ability to transfer executable code onto the device and have it run in a completely secure environment.

The separation of code and hardware certification is a new and unique solution for software protection.

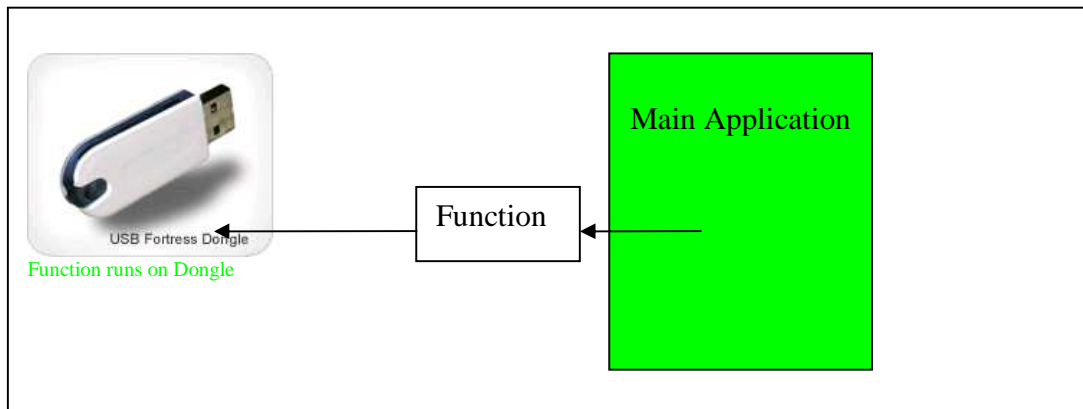
How It Works?

KEYLOK® Fortress is a USB key housing a smart card chip and memory. This driverless device can plug into any USB port and is recognized by the host operating system without any supplemental software (e.g. hardware unique device driver) being installed.

KEYLOK® Fortress is a complete computing platform that runs in parallel to the main computer. It allows algorithms and functions of an application to be stored and then run on the USB key without ever being loaded into the main computer memory. All data is exchanged through the USB port. This structural flexibility provides an extremely secure environment and unlimited possibilities for the protection of an application or data.

Implementation Overview

A portion of the application to be protected is selected and migrated to the KEYLOK® Fortress smart card chip. The function or algorithm selected should be one or more of the key pieces of the application, rendering the application inoperable if the KEYLOK® Fortress device is not present on the host computer. This function will run only on the Fortress dongle and is never loaded into main computer memory. The main application makes calls to the external function and data is passed back and forth between the protected application and the dongle.



When the main application hits the function, it passes data to the device, the function runs on the device and passes back data to the main application. The function on the dongle is never loaded into memory of the computer on which the main application is running.

The functions which can be migrated to the device must be C code compiled with a special compiler targeting the dongle CPU's internal instruction set. Many built-in file/math/memory operations, cryptographic algorithms, and time functions are supported.

Code Selection

What code should be transferred to the device? To maximize the security protection of your application, you should select code which is:

1. Unique and specific to your application. This provides you with a more secure solution.
2. Vital to the operation of your application. Without it, your application does not run. Hackers will not be able to use your application by bypassing these key functions.

And you should avoid code which is:

1. Based upon a popular algorithm. Hackers understand the functionality of most popular algorithms and can simulate them.
2. Processor intensive. The processor on the smart card is slower than most computers and you want to avoid any bottlenecks.

Directory and File Structure

As stated above, Fortress supports a typical PC operating system like directory and file structure. Each directory is secured by a Developer PIN. The Developer PIN provides you with the ability to write files to the directory. A randomly generated Developer PIN is created for you. You have the ability to change this PIN in one of two ways. You can provide KEYLOK[®] with your new PIN and it will be programmed into the dongle or you can use the Fortress Management Utility to change it once you receive your programmed dongle. Each directory is also secured by a read/execute PIN. This User PIN provides you with the ability to execute the binary files which have been loaded onto the Fortress dongle. A randomly generated User PIN is created for you. You have the ability to change this PIN in the same manner as the Developer PIN.

Three types of files can be migrated to the Fortress dongle. All of the files are named by KEYLOK[®] and are programmed onto the dongle during manufacturing. Binary executable files contain your key functions or code that are called by the KEXEC function. Data files are for use by the binary executable files loaded onto the dongle to perform the necessary functions. These data files are only accessible by your executable files. Key files are used by the executable files for encryption/decryption functions.

You can have as many files as is necessary to accomplish your goals. You are limited by the 51,000 bytes of memory available on the dongle less the amount of user memory you need. For example, if you need 5,120 bytes of user memory, then you have 45,880 bytes for your files.

Developer PIN

The Developer PIN is necessary to add or replace files on the dongle and to change the Developer PIN. You must treat this Developer PIN as key confidential company information. It should only be known to a select group of people and should not be part of your distributed application. You will need to use the Fortress Management Utility to make changes to the files or User PIN.

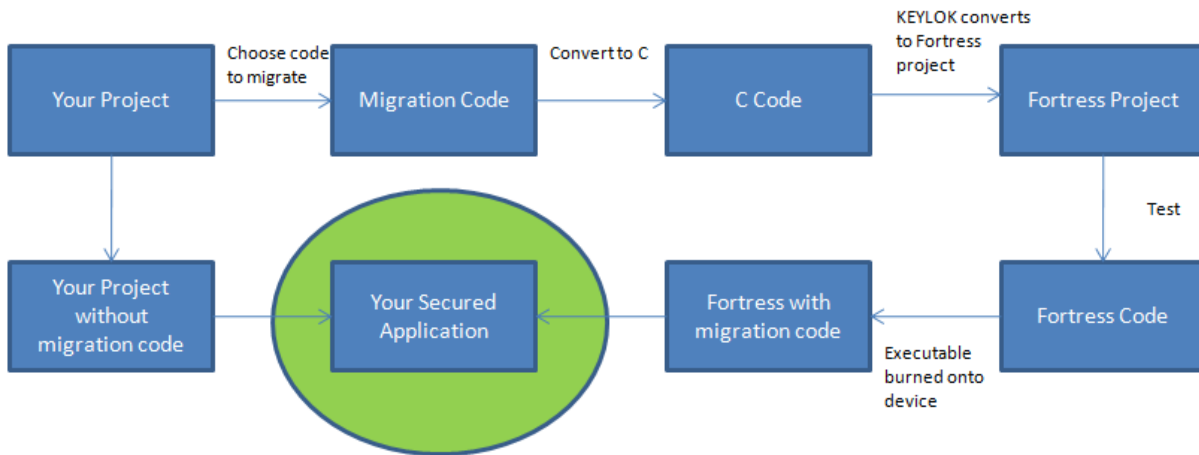
User PIN

The User PIN is used by the KEXEC function which is the interface between your application and the code residing on the Fortress dongle. The User PIN is part of your distributed application.

Development Process

You will develop and debug your application ignoring the device and software protection methodologies. Once your application development is complete, identify important code as noted above. The selected code to be ported to the dongle must be converted to C if it is not already in that format. KEYLOK[®] will work with you to get the code converted into C, if necessary, and then will create the compiled version, test it and then load it onto the Fortress dongle. You will modify your code to make the calls to the migrated code. You will be provided

with your unique device which when combined with your application provides you with the most secure solution available.



KEEXEC – Interfacing with Your Code on the Dongle

KEEXEC is the function you call from within your code to access and execute the migrated code on the Fortress dongle.

The first argument is a pointer to a string containing the name of the folder in the directory structure that contains your code and optionally a data file(s) that will be used by your code. The name of this folder is assigned by KEYLOK[®] personnel for your company.

The second argument is a pointer to a string containing the name of the program that contains your company unique code. One or more programs can be stored within the dongle to execute your application unique algorithms. The program file name(s) are assigned by KEYLOK[®] personnel.

The third argument is a pointer to a string that contains an eight character (64 bit) password that must be known to execute programs stored in your company folder on the dongle. This value can be changed from the default value assigned by KEYLOK[®] personnel to whatever value you wish.

The fourth argument is a pointer to a buffer that is used for passing arguments to/from your company unique functions.

The fifth argument is the size of the buffer in bytes (unsigned short) containing IN/OUT arguments (maximum of 250 bytes).

KEXEC (LPSTR ExeDir, LPSTR ExeFile, LPSTR UserPIN, LPSTR Buffer, USHORT BufferSize)	
Prerequisite	Successful Check for KEYLOK®
ExeDir	Directory name on the dongle
ExeFile	File name of the executable on the dongle
UserPIN	Security PIN which allows access to execute code on dongle
Buffer	Data buffer for passing data between the application and the dongle
BufferSize	Size of data buffer in bytes

KEYLOK® Fortress Functions

In this chapter we describe the KEYLOK® Fortress Functions supported by the Smart Card OS.

The functions can be categorized as follows:

Category	Description
Flow Control	Used to terminate your function
Input/Output	Used to pass data between your application and the functions on the dongle
File Operations	Used to operate internal files of Fortress, i.e. data file.
Mathematics	Support simple/double precision float calculations. Provides all the common mathematic functions defined in math.h of standard C language.
Cryptographic Algorithms	Provide SHA1, RSA, DES/TDES Cryptographic algorithms.
Memory Operations	Basic memory operation support.
Time	Timer and real clock functions.
Macro and Auxiliary	Used for type conversions.
Device Info	Used to get information about the device.

Flow Control

`_exit`

Exit Fortress EXF

```
void _exit(void)
```

Parameters:

None

Return Values:

None

Remarks:

Must use this function to terminate Fortress hardware program (EXF) and invoking this function anywhere will result in the termination of the EXF.

Input/Output

`_set_response`

Set the data to be transmitted back to PC by Fortress EXF.

```
BYTE _set_response(
```

```
BYTE bLen,
```

```
BYTE* pbBuff);
```

Parameters:

bLen [in] Length of returned data.

pbBuff [in] Buffer address.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

The length of returned data cannot exceed that of the communication buffer of 250 bytes.

File Operation

This group of functions is used to operate the internal file system of the Fortress device

_open

Select and open the file specified

```
BYTE _open(  
    WORD wFid,  
    HANDLE* pHandle);
```

Parameters:

wFid [in] File ID
pHandle [out] Handle to the file opened.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

Fortress internal program (EXF) can open up to 3 data files concurrently. If you want to open more files, unwanted file handles must be closed using _close(). You must check the Return value to see whether or not the file open operation succeeds.

_close

Close the file specified

```
BYTE _close(HANDLE handle);
```

Parameters:

handle [out] Handle to the file to be closed.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None.

_read

Read data from a opened file. Reading privilege must be satisfied according to the file security attributes.

```
BYTE _read(  
    HANDLE handle,  
    WORD wOffset,  
    BYTE bLength,  
    BYTE* pbBuff);
```

Parameters:

handle [in] Handle to the file opened.
wOffset [in] Reading offset.
bLength [in] Length of the data to be read (1 - 247 bytes).
pbBuff [out] Pointer to data buffer, used to store the data read.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

To read the content of the specified file, the reading privilege must be satisfied according to the file security attributes:

- Data file, Public Key file can be read by EXFs of the same directory freely.
- If the security attributes of an EXF has been set to "unreadable/unwritable", then it cannot be read by other EXFs. Otherwise, it can be read by another EXF of the same type if they also reside in the same directory.

- The Private key file can NEVER be read.

_write

Write data to a file already opened. The writing privilege must be satisfied according to the file security attributes.

```
BYTE _write(
    HANDLE handle,
    WORD wOffset,
    BYTE bLength,
    BYTE* pbBuff);
```

Parameters:

handle [in] Handle to the file opened.
wOffset [in] Writing offset.
bLength [in] Length of the data to be written (1 – 247 bytes).
pbBuff [in] Pointer to the data buffer, used to store the data to be written.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

To write data to the specified file, the writing privilege must be satisfied according to the file security attributes:

- Data file, Public Key file can be written by EXFs of the same directory freely.
- If the security attributes of an EXF has been set to "unreadable/unwriteable, then it cannot be written by other EXFs. Otherwise, it can be written by another EXF of the same type if they also reside in the same directory.
- The Private key file can be written by EXFs of the same directory.

_create

Create a new file and inherit the security attributes from it's parents file.

```
BYTE _create(
    WORD wFid,
    WORD wSize,
    BYTE bFileType,
    BYTE* bFlag,
    HANDLE* pHandle);
```

Parameters:

wFid [in] ID of the file to be created.
wSize [in] Size of the file.
bFileType [in] Possible values:
 SES_FILE_TYPE_EXE – executable file
 SES_FILE_TYPE_EXE_DATA – Data file
 SES_FILE_TYPE_RSA_PUB – RSA Public file
 SES_FILE_TYPE_RSA_SEC – RSA Private file
bFlag [in] Possible values:
 CREATE_OPEN_ALWAYS
 Open the file if it already exists, create a new file otherwise.
 CREATE_FILE_NEW
 Create and open the file, return error if it already exists.
 CREATE_OPEN_EXISTING
 Open an existing file.
pHandle [out] Handle to the file if it's opened successfully.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

Because this function is designed specifically for remote operations, files created by it can only be accessed internally in Fortress. In other words, one cannot write to a file created by this function even if he has the developer level PIN.

Files created by this function will inherit security attributes from its parent file. So if you want to create a usable new file, the parent file must have its security attribute set to "READABLE/WRITEABLE". Otherwise a useless "dead file" will be created.

Executable (EXF) files created by this function must be enabled using `_enable_exe()` before it can be executed.

`_enable_exe`

Enable EXFs created by `_create()`.

```
BYTE _enable_exe(WORD wFid);
```

Parameters:

wFid [in] ID of the EXF to be enabled.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

Only for EXFs created by `_create()`.

Mathematics

This category includes all the commonly used mathematic functions.

If you want only single precision calculation results, these functions are not absolutely necessary. `Math.h` can be included and those functions can be used. However, using single precision float functions can greatly boost calculation speeds.

Conversely, double precision float calculations must use these functions due to the fact the compiler cannot effectively support double type. Double precision float in Fortress is represented using a structure, defined as `DOUBLE_T`, and thus you cannot assign a value to it simply by using `"="`. There are three common ways to assign a double precision float:

1. By using a memory copy.
2. Using a function to convert different formats among single precision, integer and double precision
3. Using the auxiliary function which can convert a string to a double precision float.

`_addf`

Single precision float addition

```
float _addf(  
    float x,  
    float y);
```

Parameters:

x [in] augends

y [in] addend

Return Values:

Sum of two single precision float numbers.

Remarks:

None

_add

Double precision float addition

```
BYTE _add(  
    DOUBLE_T *presult,  
    DOUBLE_T *px,  
    DOUBLE_T *py);
```

Parameters:

Result [out] Pointer to the result
px [in] Pointer to the augends
py [in] Pointer to the addend

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_subf

Single precision float subtraction

```
float _subf(  
    float x,  
    float y);
```

Parameters:

x [in] minuend
y [in] subtrahend

Return Values:

Subtraction of two single precision float numbers.

Remarks:

None

_sub

Double precision float subtraction

```
BYTE _sub(  
    DOUBLE_T *presult,  
    DOUBLE_T *px,  
    DOUBLE_T *py);
```

Parameters:

Result [out] Pointer to the result
px [in] Pointer to the minuend
py [in] Pointer to the subtrahend

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_multf

Single precision float multiplication

```
float _multf(  
    float x,  
    float y);
```

Parameters:

x [in] multiplicand
y [in] multiplier

Return Values:

Product of two single precision float numbers.

Remarks:

None

_mult

Double precision float multiplication

```
BYTE _mult(  
    DOUBLE_T *presult,  
    DOUBLE_T *px,  
    DOUBLE_T *py);
```

Parameters:

Result [out] Pointer to the result
px [in] Pointer to the multiplicand
py [in] Pointer to the multiplier

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_divf

Single precision float division

```
float _divf(  
    float x,  
    float y);
```

Parameters:

x [in] dividend
y [in] divisor

Return Values:

Quotient of two single precision float numbers.

Remarks:

None

_div

Double precision float division

```
BYTE _div(  
    DOUBLE_T *presult,  
    DOUBLE_T *px,  
    DOUBLE_T *py);
```

Parameters:

Result [out] Pointer to the result
px [in] Pointer to the dividend
py [in] Pointer to the divisor

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_sinf

Sine of a single precision float

```
float _sinf(float x);
```

Parameters:

x [in] radian

Return Values:

Sin of the radian.

Remarks:

None

_sin

Sine of a double precision float

```
BYTE _sin(  
    DOUBLE_T *presult,  
    DOUBLE_T *px);
```

Parameters:

Result [out] Pointer to the result

px [in] Pointer to the radian

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_cosf

Cosine of a single precision float

```
float _cosf(float x);
```

Parameters:

x [in] radian

Return Values:

Cosine of the radian.

Remarks:

None

_cos

Cosine of a double precision float

```
BYTE _cos(  
    DOUBLE_T *presult,  
    DOUBLE_T *px);
```

Parameters:

Result [out] Pointer to the result

px [in] Pointer to the radian

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_tanf

Tangent of a single precision float

```
float _tanf(float x);
```

Parameters:

x [in] radian

Return Values:

Tangent of the radian.

Remarks:

None

_tan

Tangent of a double precision float

```
BYTE _tan(  
    DOUBLE_T *presult,  
    DOUBLE_T *px);
```

Parameters:

Result [out] Pointer to the result
px [in] Pointer to the radian

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_asinf

Asin of a single precision float

```
float _asinf(float x);
```

Parameters:

x [in] Sine value

Return Values:

Asin of the float.

Remarks:

None

_asin

Asin of a double precision float

```
BYTE _asin(  
    DOUBLE_T *presult,  
    DOUBLE_T *px);
```

Parameters:

Result [out] Pointer to the result
px [in] Pointer to the sine value

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_acosf

Acos of a single precision float

```
float _acosf(float x);
```

Parameters:

x [in] cosine value

Return Values:

Acos of the float.

Remarks:

None

_acos

Acos of a double precision float

```
BYTE _acos(  
    DOUBLE_T *presult,  
    DOUBLE_T *px);
```

Parameters:

Result [out] Pointer to the result
px [in] Pointer to the cosine value.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_atanf

Cotangent of a single precision float

```
float _atanf(float x);
```

Parameters:

x [in] Tangent value.

Return Values:

Cotangent of the float.

Remarks:

None

_atan

Cotangent of a double precision float

```
BYTE _atan(  
    DOUBLE_T *presult,  
    DOUBLE_T *px);
```

Parameters:

Result [out] Pointer to the result

px [in] Pointer to the tangent value

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_atanf2f

Cotangent of two single precision floats quotient

```
float _atan2f(  
    float x,  
    float y);
```

Parameters:

x [in] dividend.

Y [in] divisor

Return Values:

Cotangent of two float's quotient.

Remarks:

None

_atan2

Cotangent of two double precision floats

```
BYTE _atan2(  
    DOUBLE_T *presult,  
    DOUBLE_T *px  
    DOUBLE_T *py);
```

Parameters:

Result [out] Pointer to the result

px [in] Pointer to the dividend

py [in] Pointer to the divisor.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_sinhf

Sinh of a single precision float

```
float _sinhf(float x);
```

Parameters:

x [in] Radian.

Return Values:

Sinh of the float.

Remarks:

None

_sinh

Sinh of a double precision float

```
BYTE _sinh(  
    DOUBLE_T *presult,  
    DOUBLE_T *px);
```

Parameters:

Result [out] Pointer to the result

px [in] Pointer to the radian

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_coshf

Cosh of a single precision float

```
float _coshf(float x);
```

Parameters:

x [in] Radian.

Return Values:

Cosh of the float.

Remarks:

None

_cosh

Cosh of a double precision float

```
BYTE _cosh(  
    DOUBLE_T *presult,  
    DOUBLE_T *px);
```

Parameters:

Result [out] Pointer to the result

px [in] Pointer to the radian

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

`_tanhf`

Tanh of a single precision float

```
float _tanhf(float x);
```

Parameters:

x [in] Radian.

Return Values:

Tanh of the float.

Remarks:

None

`_tanh`

Tanh of a double precision float

```
BYTE _tanh(  
    DOUBLE_T *presult,  
    DOUBLE_T *px);
```

Parameters:

Result [out] Pointer to the result

px [in] Pointer to the radian

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

`_ceilf`

Calculates the ceiling of a single precision float

```
float _ceilf(float x);
```

Parameters:

x [in] Floating point value.

Return Values:

This function returns a float value representing the smallest integer that is great than or equal to x.

Remarks:

None

`_ceil`

Calculates the ceiling of a double precision float

```
BYTE _ceil(  
    DOUBLE_T *presult,  
    DOUBLE_T *px);
```

Parameters:

Result [out] Pointer to the result

px [in] Pointer to the float parameter

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

`_floorf`

Calculates the floor of a single precision float

```
float _floorf(float x);
```

Parameters:

x [in] Floating point value.

Return Values:

This function returns a float value representing the largest integer that is less than or equal to x.

Remarks:

None

_floor

Calculates the floor of a double precision float

```
BYTE _floor(  
    DOUBLE_T *presult,  
    DOUBLE_T *px);
```

Parameters:

Result [out] Pointer to the result
px [in] Pointer to the float parameter

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_absf

Calculates the absolute value of a single precision float

```
float _absf(float x);
```

Parameters:

x [in] Floating point value.

Return Values:

Absolute value of x.

Remarks:

None

_abs

Calculates the absolute value of a double precision float

```
BYTE _abs(  
    DOUBLE_T *presult,  
    DOUBLE_T *px);
```

Parameters:

Result [out] Pointer to the result
px [in] Pointer to the float parameter

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_fmodf

Calculates the single precision float remainder.

```
float _fmodf(  
    float x,  
    float y);
```

Parameters:

x [in] Dividend.
Y [in] Divisor

Return Values:

The floating point remainder of x/y.

Remarks:

The `_fmodf` function calculates the floating point remainder f of x/y such that $x=i*y+f$, where i is an integer, f has the same sign as x , and the absolute value of f is less than the absolute value of y .

`_fmod`

Calculates the double precision float remainder.

```
BYTE _fmod(  
    DOUBLE_T *presult,  
    DOUBLE_T *px,  
    DOUBLE_T *py);
```

Parameters:

Result [out] Pointer to the result
x [in] Pointer to the dividend.
Y [in] Pointer to the divisor

Return Values:

Return `KF_SUCCESS` if succeed or corresponding error code otherwise.

Remarks:

The `_fmod` function calculates the floating point remainder f of x/y such that $x=i*y+f$, where i is an integer, f has the same sign as x , and the absolute value of f is less than the absolute value of y .

`_expf`

Calculates the exponential value of a single precision float

```
float _expf(float x);
```

Parameters:

x [in] exponent.

Return Values:

This function returns the exponential value of the floating point parameter x , if successful.

Remarks:

None

`_exp`

Calculates the exponential value of a double precision float

```
BYTE _exp(  
    DOUBLE_T *presult,  
    DOUBLE_T *px);
```

Parameters:

Result [out] Pointer to the result
px [in] Pointer to the exponent

Return Values:

Return `KF_SUCCESS` if succeed or corresponding error code otherwise.

Remarks:

None

`_logf`

Calculates logarithms of a single precision float

```
float _logf(float x);
```

Parameters:

x [in] float parameter.

Return Values:

Logarithm of the single precision float x .

Remarks:

None

_log

Calculates the logarithms of a double precision float

```
BYTE _log(  
    DOUBLE_T *presult,  
    DOUBLE_T *px);
```

Parameters:

Result [out] Pointer to the result
px [in] Pointer to the float parameter

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_log10f

Calculates the base 10 logarithm of a specified single precision float

```
float _log10f(float x);
```

Parameters:

x [in] float parameter.

Return Values:

The base 10 logarithm of the single precision float x.

Remarks:

None

_log10

Calculates the base 10 logarithm of a specified double precision float

```
BYTE _log10(  
    DOUBLE_T *presult,  
    DOUBLE_T *px);
```

Parameters:

Result [out] Pointer to the result
px [in] Pointer to the float parameter

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_sqrtf

Calculates the square root of a single precision float

```
float _sqrtf(float x);
```

Parameters:

x [in] float parameter.

Return Values:

The square root of the single precision float x.

Remarks:

None

_sqrt

Calculates the square root of a double precision float

```
BYTE _sqrt(  

```

```
DOUBLE_T *presult,  
DOUBLE_T *px);
```

Parameters:

Result [out] Pointer to the result
px [in] Pointer to the float parameter

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_powf

Calculates x raised to the power of y where both x and y are single precision float

```
float _powf(  
float x,  
float y);
```

Parameters:

x [in] Base.
Y [in] Exponent

Return Values:

The value of x^y .

Remarks:

None

_pow

Calculates x raised to the power of y where both x and y are of double precision float

```
BYTE _pow(  
DOUBLE_T *presult,  
DOUBLE_T *px,  
DOUBLE_T *py);
```

Parameters:

Result [out] Pointer to the result
px [in] Pointer to the base
py [in] Pointer to the exponent

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_modf

Divide a double precision float to integral part and fractional part

```
BYTE _modf(  
DOUBLE_T *presult,  
DOUBLE_T *px,  
DOUBLE_T *intptr);
```

Parameters:

Result [out] Pointer to the fractional part
px [in] Pointer to the double to be divided
intptr [out] Pointer to the integral part

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_frexp

Divide a double precision float to the product of a fraction and power of 2.

```
BYTE _frexp(  
    DOUBLE_T *presult,  
    DOUBLE_T *px,  
    int *expptr);
```

Parameters:

Result [out] Pointer to the fractional part
px [in] Pointer to the double to be divided
expptr [out] Pointer to the exponent

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This function calculates presult, expptr such that:

$$*px = *presult \times 2^{(*expptr)}$$

$$\text{Where } 1 > *presult \geq 0.5$$

_ldexp

Calculate the power of a double precision float and 2.

```
BYTE _ldexp(  
    DOUBLE_T *presult,  
    DOUBLE_T *px,  
    int exp);
```

Parameters:

Result [out] Pointer to the result
px [in] Pointer to the float parameter
exp [in] Exponent of 2

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This function calculates presult such that:

$$*presult = *px \times 2^{(exp)}$$

_fdcmp

Compare two double precision floats.

```
char _fdcmp(  
    DOUBLE_T *px,  
    DOUBLE_T *py);
```

Parameters:

px [in] Pointer to the first float parameter
py [in] Pointer to the second float parameter

Return Values:

Return comparison result:

$$1 \ *px > *py$$

$$0 \ *px = *py$$

$$-1 \ *px < *py.$$

Remarks:

None

_dtof

Type conversion from double precision to single precision float.

```
BYTE _dtof(  

```

```
float *presult,  
DOUBLE_T *px);
```

Parameters:

result [out] Pointer to the single precision float
px [in] Pointer to the double precision float parameter

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_ftod

Type conversion from single precision to double precision float.

```
BYTE _ftod (  
DOUBLE_T *presult,  
float *px);
```

Parameters:

result [out] Pointer to the double precision float
px [in] Pointer to the single precision float parameter

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_dtol

Type conversion from double precision to 32-bit signed integer (long).

```
BYTE _dtol(  
long *presult,  
DOUBLE_T *px);
```

Parameters:

result [out] Pointer to the long integer
px [in] Pointer to the double precision float parameter

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_altod

Type conversion from 32-bit signed integer (long) to double precision float.

```
BYTE _altod (  
DOUBLE_T *presult,  
long *px);
```

Parameters:

result [out] Pointer to the result
px [in] Pointer to the 32-bit long signed integer

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

_lftod

Type conversion from 32-bit unsigned integer (long) to double precision float.

```
BYTE _lftod (  
DOUBLE_T *presult,  
long *px);
```

```
DOUBLE_T *presult,  
DWORD *px);
```

Parameters:

result [out] Pointer to the result
px [in] Pointer to the 32-bit long unsigned integer

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

None

Cryptographic Algorithms

KEYLOK[®] Fortress provides four categories of fundamental cryptographic algorithms: asymmetric cipher algorithm (RSA), symmetric cipher algorithm (DES/TDES), hash algorithm (SHA1) and real random number generator. These cryptographic algorithms can be used in protecting your software.

This manual does not provide details of the supported cryptographic algorithm functions. It is the responsibility of the software vendor to determine which, if any, algorithms will work best with their application. The detailed information is widely available on the internet.

All of the internal cryptographic algorithms have corresponding software versions. For example, you can use software-version functions to encrypt data, and then use corresponding functions of the internal hardware to decrypt them. For more detail about software version cryptographic algorithms, please refer to the "Appendix A: Cryptographic Algorithms Reference".

_tdes_enc

TDES Encryption, ECB Mode

```
BYTE _tdes_enc(  
    BYTE *pbKey,  
    BYTE bLen,  
    BYTE *pbData);
```

Parameters:

pbKey [in] 16 octets DES key
bLen [in] Length of the data to be encrypted 1-248
pbData [in,out] Plain text to be encrypted when input, cipher text when output.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This function performs triple-DES encryption in ECB mode. Try to make the length of the input plain text be multiple of eight, or try to add some padding to meet this requirement. Otherwise, the function will do some automatic padding to the plain text which will make the decrypted result contain some padding data which could yield undesirable results.

The length of the cipher text is always a multiple of 8 and it may be longer than the plain text. Because the plain text and the cipher text use the same buffer, the former will be overwritten by the latter. So if you need to use the plain text after calling this function, copy it to another memory block. Ensure the buffer is large enough for both the plain text and the resulting cipher text.

_tdes_dec

TDES decryption in ECB Mode. Paired with `_tdes_enc()`

```
BYTE _tDES_dec(  
    BYTE *pbKey,  
    BYTE bLen,  
    BYTE *pbData);
```

Parameters:

pbKey [in] 16 octets DES key
bLen [in] Length of the data to be decrypted 8-248. Must be multiple of 8.
pbData [in,out] Cipher text to be decrypted when input, plain text when output.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This function performs triple-DES decryption in ECB mode.

The length of the cipher text is always a multiple of 8 and it must be a multiple of 8 when input. Because the plain text and the cipher text use the same buffer, the latter will be overwritten by the former. So if you need to use the cipher text after calling this function, copy it to another memory block.

_des_enc

DES encryption in ECB Mode

```
BYTE _des_enc(  
    BYTE *pbKey,  
    BYTE bLen,  
    BYTE *pbData);
```

Parameters:

pbKey [in] 8 octets DES key
bLen [in] Length of the data to be encrypted 1-248
pbData [in,out] Plain text to be encrypted when input, cipher text when output.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This function performs DES encryption in ECB mode. This function uses a key of 8 octets or 56 bits which is not considered secure enough. Please use the `_tDES_enc()` for more secure encryption.

Try to make the length of the input plain text be multiple of eight, or try to add some padding to meet this requirement. Otherwise, the function will do some automatic padding to the plain text which will make the decrypted result contain some padding data which could yield undesirable results.

The length of the cipher text is always a multiple of 8 and it may be longer than the plain text. Because the plain text and the cipher text use the same buffer, the former will be overwritten by the latter. So if you need to use the plain text after calling this function, copy it to another memory block. Ensure the buffer is large enough for both the plain text and the resulting cipher text.

_des_dec

DES decryption in ECB Mode. Paired with `_des_enc()`

```
BYTE _des_dec(  
    BYTE *pbKey,  
    BYTE bLen,  
    BYTE *pbData);
```

Parameters:

pbKey [in] 8 octets DES key
bLen [in] Length of the data to be decrypted 8-248. Must be multiple of 8.
pbData [in,out] Cipher text to be decrypted when input, plain text when output.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This function performs DES decryption in ECB mode. This function uses a key of 8 octets or 56 bits which is not considered secure enough. Please use the `_tdes_dec()` for more secure encryption.

The length of the cipher text is always a multiple of 8 and it must be a multiple of 8 when input. Because the plain text and the cipher text use the same buffer, the latter will be overwritten by the former. So if you need to use the cipher text after calling this function, copy it to another memory block.

`_sha1_init`

SHA1 initialization. SHA1 is completed using a group of functions. Please refer to the Remarks section.

```
BYTE _sha1_init(HASH_CONTEXT *pContext);
```

Parameters:

pContext [in] HASH context

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This function initializes the HASH context.

A typical HASH procedure includes:

- Initialize context using `_sha1_init()`
- Add data, this can be achieved by calling `_sha1_update()` several times
- Get hash result using `_sha1_final()`, purge the hash context at the same time.

Function group `_sha1_xxx()` needs to maintain a buffer in the system, which will be used for the completion of the RSA signature verification.

The current version doesn't allow multiple HASH processes to run concurrently.

`_sha1_update`

Adding SHA1 data can be achieved using a calling sequence. For details please refer to the Remarks section of `_sha1_init`.

```
BYTE _sha1_update(  
    HASH_CONTEXT *pContext,  
    BYTE *pbData,  
    BYTE bLen);
```

Parameters:

pContext [in] HASH context

bLen [in] Length of the data to be added this time

pbData [in] Data to be added this time..

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

Use this function to add data to be hashed. A big data block can be divided into several smaller ones and added to the hash function by calling `_sha1_update()` continuously. You are limited to 8191 bytes for the total inputted data.

_sha1_final

Get the result of SHA1 HASH and then purge the context. SHA1 is achieved by a group of functions. Please refer to the Remarks section of `_sha1_init()`.

```
BYTE _sha1_final(  
    HASH_CONTEXT *pContext,  
    BYTE *pbResult);
```

Parameters:

`pContext` [in] HASH context
`pbResult` [out] 20 bytes hash result returned.

Return Values:

Return `KF_SUCCESS` if succeed or corresponding error code otherwise.

Remarks:

After calling this function, the system will also copy the result to a specific HASH buffer so as to be used in the signature verification process.

`pbResult` must have sufficient storage to contain the HASH result.

_rsa_enc

RSA encryption

```
BYTE _rsa_enc(  
    BYTE bMode,  
    WORD wFid,  
    BYTE bLen,  
    BYTE *pbData);
```

Parameters:

<code>bMode</code>	[in] Calculation Mode	
	<code>RSA_CALC_NORMAL</code>	Direct calculation without encoding
	<code>RSA_CALC_PKCS</code>	Encryption according to PKCS#1 standard
<code>wFid</code>	[in] Public file ID	
<code>bLen</code>	[in] Length of the data to be encrypted, different for the two modes.	
	<code>RSA_CALC_NORMAL</code>	128 bytes
	<code>RSA_CALC_PKCS</code>	1-117 bytes
<code>pbData</code>	[in,out] Plain text when input, 128 bytes cipher text when output.	

Return Values:

Return `KF_SUCCESS` if succeed or corresponding error code otherwise.

Remarks:

This function completes the RSA Public key encryption, and the length of the public key is 1024 bits. You have to specify the RSA Public key file for this function.

The result after encryption is always 128 bytes long. You have to allocate sufficient storage for `pbData`.

When using this function for data encryption, setting `bMode` to `RSA_CALC_PKCS` is highly recommended. If using `RSA_CALC_NORMAL`, try to make the first byte of the 128 bytes of data to be 0, otherwise, it may cause the inputted data to be greater than the modulus and this may lead to errors. In fact, when inputting data shorter than 128 bytes under `RSA_CALC_NORMAL` mode, the function will do some padding automatically by adding zeroes to the front. We do not recommend having data shorter than 128 bytes as the input. `RSA_CALC_NORMAL` is commonly used in constructing other algorithm protocols such as digital signatures.

_rsa_dec

RSA decryption

```
BYTE _rsa_dec(
    BYTE bMode,
    WORD wFid,
    BYTE bLen,
    BYTE *pbData);
```

Parameters:

bMode [in] Calculation Mode

RSA_CALC_NORMAL	Direct calculation without encoding
RSA_CALC_PKCS	Decryption according to PKCS#1 standard

wFid [in] Private file ID

bLen [in] Length of the data to be decrypted. Must be 128 bytes.

pbData [in,out] Cipher text when input, plain text when output.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This function completes the RSA Private key decryption, and the length of the private key is 1024 bits. You have to specify the RSA Public key file for this function.

When using this function for data decryption, setting bMode to RSA_CALC_NORMAL yields a result which is always 128 bytes. If using RSA_CALC_PKCS, the result is "data length" + "plain text". The first byte of pbData is the length of the plain text and the following bytes are the decrypted plain text.

You must use the same mode for encryption and decryption to get valid results.

_rsa_sign

RSA digital signature. There are some limitations to using this function.

```
BYTE _rsa_sign(
    BYTE bMode,
    WORD wFid,
    BYTE bLen,
    BYTE *pbData);
```

Parameters:

bMode [in] Calculation Mode

	RSA_CALC_NORMAL	Encrypt the HASH value directly using the private key
	RSA_CALC_HASH	Do the HASH operation first for the inputted data and then encrypt the result using the private key
	RSA_CALC_PKCS	Sign the HASH value according to PKCS#1 standard
	RSA_CALC_HASH RSA_CALC_PKCS	Do the HASH operation first and then sign the HASH result according to PKCS#1 standard.

wFid [in] Private file ID

bLen [in] The data to be signed. A different length for each mode.

	RSA_CALC_NORMAL	20 bytes
	RSA_CALC_HASH	1-128 bytes

	RSA_CALC_PKCS	20 bytes
	RSA_CALC_HASH RSA_CALC_PKCS	1-128 bytes

pbData [in,out] Plain text or HASH value when input, 128 octets signature when output.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This function completes the RSA Private key signing operation, and the length of the private key is 1024 bits and the signing result is 128 bytes.

Of the four modes above, RSA_CALC_PKCS is the standard digital signing scheme. This scheme provides compatibility with other signing systems. If the inputted plain text is relatively short, you can use RSA_CALC_HASH|RSA_CALC_PKCS to do both hashing and signing together. The other two modes are not recommended.

This function only supports the SHA1 HASH algorithm.

_rsa_veri

This function does RSA digital signature verification according to the HASH buffer and inputted signature.

```
BYTE _rsa_veri (
    BYTE bMode,
    WORD wFid,
    BYTE bLen,
    BYTE *pbData);
```

Parameters:

bMode [in] Calculation Mode

	RSA_CALC_NORMAL	Decrypt the signature directly and compare with the HASH buffer to verify.
	RSA_CALC_PKCS	According to the Hash buffer and signature, do the standard PKCS#1 verification

wFid [in] Public file ID

bLen [in] Length of the digital signature. Must be 128..

pbData [in] The signature.

Return Values:

If verified successfully, returns KF_SUCCESS, otherwise it returns the corresponding error code.

Remarks:

This function completes the RSA signature verification and the key length is 1024 bits.

This verification function does not provide hashing calculations of the plain text. You have to call the corresponding hashing function to do HASH first and then call this function. This function only takes HASH values from the system's HASH buffer and cannot take a directly inputted HASH value.

This function only supports the SHA1 HASH algorithm.

_rand

Hardware generated real random number

```
BYTE _rand (
    BYTE bLen,
```

```
BYTE *pbData);
```

Parameters:

bLen [in] Length of the random number to be generated. 1-255
pbData [out] Random number generated.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

Random number produced by this function is a hardware generated real number and needs no initialization.

Memory Operations

These functions provide basic memory operations such as copy and move.

`_mem_copy`

Copy data between memory buffers. Similar to memcpy() in standard C.

```
BYTE _mem_copy(  
void *pDest,  
void *pSrc,  
BYTE bLen);
```

Parameters:

pDest [in] Destination address
pSrc [in] Source address
bLen [in] Length of the data to be copied.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

Different from memcpy in standard C, `_mem_copy` returns error code rather than a pointer to the destination address.

`_mem_move`

Move data from one buffer to another. Similar to memcpy() in standard C.

```
BYTE _mem_move(  
void *pDest,  
void *pSrc,  
BYTE bLen);
```

Parameters:

pDest [in] Destination address
pSrc [in] Source address
bLen [in] Length of the data to be moved.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

Different from memmove in standard C, `_mem_move` returns error code rather than a pointer to the destination address.

`_mem_set`

Fill the buffer with the value specified, similar to memset() in standard C

```
BYTE _mem_set(  
void *pDest,
```

```
BYTE c,  
BYTE bLen);
```

Parameters:

pDest [in] Destination address
c [in] Value to be filled in
bLen [in] Length of the data to be filled.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

Different from memset in standard C, `_mem_set` returns error code rather than a pointer to the destination address.

`_mempool_init`

Initialize the starting address and size of memory pool, used for dynamic memory management. It must be called before any one of `_malloc()`, `_calloc()` or `_realloc()`.

```
BYTE _mempool_init(  
void *pStart,  
WORD wSize);
```

Parameters:

pStart [in] Starting address of memory pool, must be a pointer to XRAM
wSize [in] Size of the memory pool.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

The internal resources cannot be used as freely as in a PC. Try to use static memory instead of dynamic memory unless you are very familiar with memory structure and usage.

`_malloc`

Malloc a memory block in the memory pool set by `_mempool_init`.

```
Void* _malloc(WORD wSize);
```

Parameters:

wSize [in] Size of the memory to be malloced.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

Must call `_mempool_init()` first to set an available memory pool.

The internal resources cannot be used as freely as in a PC. Try to use static memory instead of dynamic memory unless you are very familiar with memory structure and usage.

`_calloc`

Allocate an array in the memory pool set by `_mempool_init()` with elements initialized to 0.

```
Void* _calloc(  
WORD wNobj,  
WORD wSize);
```

Parameters:

wNobj [in] Number of elements
wSize [in] Length in bytes of each element.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

Must call `_mempool_init()` first to set an available memory pool.

The internal resources cannot be used as freely as in a PC. Try to use static memory instead of dynamic memory unless you are very familiar with memory structure and usage.

_realloc

Reallocate memory blocks in the memory pool set by `_mempool_init()`.

```
Void* _realloc(  
    Void *pointer,  
    WORD wSize);
```

Parameters:

pointer [in] Pointer to previously allocated memory block
wSize [in] New size in bytes.

Return Values:

Return `KF_SUCCESS` if succeed or corresponding error code otherwise.

Remarks:

Must call `_mempool_init()` first to set an available memory pool.

The internal resources cannot be used as freely as in a PC. Try to use static memory instead of dynamic memory unless you are very familiar with memory structure and usage.

_free

Free allocated memory block

```
BYTE _free( void *pointer);
```

Parameters:

pointer [in] Pointer to the memory block to be freed

Return Values:

Return `KF_SUCCESS` if succeed or corresponding error code otherwise.

Remarks:

None

_invert

Invert the contents of the memory buffer

```
BYTE _invert(  
    Void *pvdata,  
    BYTE bLen);
```

Parameters:

pvdata [in] Pointer to the memory buffer
bLen [in] Length of the data to be inverted.

Return Values:

Return `KF_SUCCESS` if succeed or corresponding error code otherwise.

Remarks:

This function can invert the higher and lower part of a specified memory buffer.

_mem_cmp

Compare characters in two buffers

```
char _mem_cmp(  
    void *pDest,  
    void *pSrc,  
    BYTE length);
```

Parameters:

pDest [in] First buffer
pSrc [in] Second buffer

length [in] Number of characters.

Return Values:

Return values indicates the relationship between the buffers.

<0 pdest less than psrc

=0 pdest equals psrc

>0 pdest greater than psrc

Remarks:

The `_mem_cmp` function compares the first `bLen` bytes of `pdest` and `psrc` and returns a value indicating their relationship.

Time Functions

These functions provide access to the timer. The timer is a 64 bit counter of CPU. You can use the current reading of the counter and the frequency of the CPU to calculate the real elapsed time.

`_set_timer`

Set the timer's initial value and working mode.

```
BYTE _set_timer(  
    BYTE bMode,  
    DWORD *pdwCount);
```

Parameters:

bMode [in] Timer mode

0 - Non-cycle mode, the timer stops when the counter number overflows

1 - Cycle mode, the timer restarts when the counter number overflows

2 - Cycle reset mode, the timer restarts from initial value after the

counter number overflows

pdwCount [in] Pointer to the initial value.

Return Values:

Return `KF_SUCCESS` if succeed or corresponding error code otherwise.

Remarks:

The counter uses counting up mechanism.

`_start_timer`

Start the timer according to the specified working mode and initial value

```
BYTE _start_timer();
```

Parameters:

None

Return Values:

Return `KF_SUCCESS` if succeed or corresponding error code otherwise.

Remarks:

None

`_stop_timer`

Stop the timer.

```
BYTE _stop_timer();
```

Parameters:

None

Return Values:

Return `KF_SUCCESS` if succeed or corresponding error code otherwise.

Remarks:

None

`_get_timer`

Get current reading of the timer.

```
BYTE _get_timer(DWORD *pdwCount);
```

Parameters:

pdwCount [out] Address of the variable for storing the current reading of the timer.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

The timer uses counting up mechanism, and the counter increases itself by one for 64 CPU clock cycles. The CPU runs at a frequency of 16MHZ, so the time span represented by one counter unit is:

$$1*64/16000000 = 4\mu s$$

The counter number is stored in a DWORD variable, so the total time can be represented by the counter before it overflows is:

$$0xffffffff*4/1000000/3600 = 4.77 \text{ hours}$$

Macro and Auxilliary Functions

These functions provide some additional functionality

`_swap_u16`

This macro is used to invert a 2 byte memory buffer

```
_swap_u16(__x__);
```

Parameters:

__x__ [in] address of a 2 byte variable.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This macro is kept for compatibility purposes only. Use invert() instead.

`_swap_u32`

This macro is used to invert a 4 byte memory buffer

```
_swap_u32(__x__);
```

Parameters:

__x__ [in] address of a 4 byte variable.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This macro is kept for compatibility purposes only. Use invert() instead.

LE16_TO_CC

This macro is converts a 2 byte variable (i.e. short) to a byte sequence supported by the compiler.

```
LE16_TO_CC(__x__);
```

Parameters:

__x__ [in] address of a 2 byte variable.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This is dependent upon the compiler used.

LE32_TO_CC

This macro is converts a 4 byte variable (i.e. long) to a byte sequence supported by the compiler.

```
LE32_TO_CC(__x__);
```

Parameters:

`__x__` [in] address of a 4 byte variable.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This is dependent upon the compiler used.

CC_TO_LE16

This macro is converts a 2 byte compiler variable to a short.

```
CC_TO_LE16(__x__);
```

Parameters:

`__x__` [in] address of a 2 byte variable.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This is dependent upon the compiler used.

CC_TO_LE32

This macro is converts a 4 byte compiler variable to a long.

```
CC_TO_LE32(__x__);
```

Parameters:

`__x__` [in] address of a 4 byte variable.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This is dependent upon the compiler used.

BE16_TO_CC

This macro converts a 2 byte variable (i.e. short) to a byte sequence supported by the compiler.

```
BE16_TO_CC(__x__);
```

Parameters:

`__x__` [in] address of a 2 byte variable.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This is dependent upon the compiler used.

BE32_TO_CC

This macro is converts a 4 byte variable (i.e. long) to a byte sequence supported by the compiler.

```
BE32_TO_CC(__x__);
```

Parameters:

`__x__` [in] address of a 4 byte variable.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This is dependent upon the compiler used.

CC_TO_BE16

This macro is converts a 2 byte compiler variable to a short.

```
CC_TO_LE16(__x__);
```

Parameters:

`__x__` [in] address of a 2 byte variable.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This is dependent upon the compiler used.

CC_TO_BE32

This macro is converts a 4 byte compiler variable to a long.

```
CC_TO_LE32(__x__);
```

Parameters:

`__x__` [in] address of a 4 byte variable.

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This is dependent upon the compiler used.

_atod

Convert a sting to double precision float.

```
BYTE _atod(  
    DOUBLE_T *presult,  
    Char *pstr)
```

Parameters:

`presult` [out] Pointer to the double precision float structure

`pStr` [in] String representation of a float. (e.g. "3.1415926")

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

This function consumes a lot of system resources. Depending upon how highly defined the number needs to be, you could use other system functions to a assign a double float value as as `_ftod()`.

DEFINE_AT

Define a variable at a specified address according to the compiler type.

```
DEFINE_AT(TYPE, NAME, ADDRESS, MEMORY)
```

Parameters:

`TYPE` [in] Type of the variable to be defined.

`NAME` [in] Variable name.

`ADDRESS` [in] Address of the variable specified

`MEMORY` [in] Memory space.

RAM_EXT	xdata
RAM_INT_DE	data
RAM_INT_ID	idata
ROM	code memory

Return Values:

None

Remarks:

None

Device Info Functions

This function provides access to the device info.

`_get_gbdata`

Read globally unique serial number

```
BYTE _get_gbdata(  
    BYTE bFlag  
    BYTE* pbData  
    BYTE bLen);
```

Parameters:

bFlag [in] Flag. GLOBAL_SERIAL_NUMBER (get the 8 octets GUSN)
pbData [out] Buffer for holding data read.
bLen [in] Size of the buffer pbData

Return Values:

Return KF_SUCCESS if succeed or corresponding error code otherwise.

Remarks:

Parameter bLen is used to represent the size of pbData. It is not necessary for it to be big enough to hold all of the data. For example, if you specify bLen=5 when reading the hardware GUSN (8 octets), this function only returns the first 5 bytes of GUSN and does not return an error.

Distributing Your Application

Integrating the KEYLOK® API

The KEXEC function is contained within the KEYLOK® API. The references to our API are satisfied by linking your code to either a Windows DLL or a library file (depending upon your programming language and development environment).

KEYLOK® Utility Programs

Fortress Management Utility

FortressManagementUtility.exe is a utility unique to your company which allows you to change the Developer and User PIN for the directory containing your code on the dongle. It also provides you with the ability to replace files on the dongle. You will need the Developer PIN to make any modifications. You can use the GUI to make your changes or the changes can be file driven. If file driven, all of the changes are contained within *FortressConfig.dat* and then the utility can be run from the command prompt and integrated into your own programming utility.

Sample Code

KEYLOK® Sample Code

The Fortress demo dongle is preloaded with two executable files; an ascending bubble sort and a descending bubble sort. Below is the sample code for calling these sorts from a C program using a variable string provided by the user at runtime.

```
/* FUNCTION PROTOTYPES */

#ifdef __cplusplus
#ifdef KL_DLL
extern "C" unsigned long __stdcall KEXEC( LPSTR ExeDir, LPSTR ExeFile, LPSTR lpUsrPin, LPSTR inBuffer,
USHORT sizeBuffer );
#else
extern "C" unsigned long __cdecl KEXEC( LPSTR ExeDir, LPSTR ExeFile, LPSTR lpUsrPin, LPSTR inBuffer,
USHORT sizeBuffer );
#endif
#else
#endif
```

```

#ifndef KL_DLL
extern unsigned long __stdcall KEXEC( LPSTR ExeDir, LPSTR ExeFile, LPSTR IpUserPin, LPSTR inBuffer, USHORT
sizeBuffer );
#else
extern unsigned long __cdecl KEXEC( LPSTR ExeDir, LPSTR ExeFile, LPSTR IpUserPin, LPSTR inBuffer, USHORT
sizeBuffer );
#endif

#endif

/* Global Variables */

char ExeDir[] = {"\\7777"};
char UserPin[] = {"12345678"}; /* this PIN is for demo dongles only. A unique code will be assigned
to your company to be used with your company specific
Fortress dongles. */

/* KEXEC Call */

case IDM_CLIENTEXE:
DialogBox(hInst, /* Current instance */
"CLIENTEXE", /* resource to use */
hWnd, /* parent handle */
(DLGPROC)CallClientEXEDlgProc; /* dialog procedure Instance address */
if (Address != -1) {
strcpy(buf2, "Test of Client EXE inside dongle to do bubble sort:\nUnsorted data: ");
strcat(buf2, sortstring);
status = KEXEC( (LPSTR) ExeDir, (LPSTR) ExeFile, (LPSTR) UserPin, (LPSTR) sortstring,
(USHORT)strlen(sortstring) ); // Maximum buffer size is 250 bytes
switch( status ) {
case KEY_ERROR_NOERROR:
strcat(buf2, "\nSorted data: ");
strcat(buf2, sortstring);
break;
case KEY_ERROR_FORTRESS_NOAUTHENTICATION:
strcat(buf2, "\nERROR: Successful Authentication is a prerequisite condition.");
break;
case KEY_ERROR_FORTRESS_NOFOLDER:
strcat(buf2, "\nERROR: Client folder with desired program not found.");
break;
case KEY_ERROR_FORTRESS_WRONGPIN:
strcat(buf2, "\nERROR: Client folder with desired program will not authenticate.");
break;
default:
strcat(buf2, "\nERROR: Unrecognized error condition.");
break;
}
MessageBox(NULL, buf2, "KEXEC Status", MB_OK);
}
return 0;

/* User Interface Code */

BOOL FAR PASCAL CallClientEXEDlgProc(HWND hwnd,
UINT message,
WPARAM wParam,
LPARAM lParam)
{

static char buf[sizeof(sortstring)];

_strset_s(buf,sizeof(buf),0);

```

```

switch (message){
case WM_INITDIALOG:
    SetFocus(GetDlgItem(hwnd, IDC_RADIOASC));
    SendMessage(GetDlgItem(hwnd, IDC_RADIOASC), BM_SETCHECK, (WPARAM)BST_CHECKED,
(LPARAM)0);
    break;
case WM_COMMAND:
    switch (wParam){
    case IDOK:
        if(SendMessage(GetDlgItem(hwnd,IDC_RADIOASC), BM_GETCHECK, (WPARAM)0
,(LPARAM)0) == BST_CHECKED)
            strcpy(ExeFile,"c101");
        else
            strcpy(ExeFile,"c102");
        GetDlgItemText(hwnd, IDC_SORTSTRING, buf, sizeof(buf));
        strcpy(sortstring,buf);
        break;
    case IDCANCEL:
        break;
    default:
        return FALSE;
    }
    EndDialog(hwnd, wParam);
}
return FALSE;
}

```

Appendix A Cryptographic Algorithms

You have many cryptographic options using the KEYLOK® Fortress dongle. The dongles have built-in support DES/TDES, RSA and SHA1 algorithms. You can maximize the protection of your application by combining these on-board algorithms with those widely available in cryptographic libraries which can be linked into your application. For example, you can encrypt text during the running of your application and pass the text to the Fortress dongle for decrypting and further processing.

The algorithms in the KEYLOK® Fortress dongle are open and standard. The security relies on the confidential Key you use and is therefore unique to each customer. This strategy ensures maximum security for you.

KEYLOK® provides you with all of the software implementations of the algorithms mentioned above to make it convenient for your development. You can use these libraries directly or obtain other ones. The libraries and header files are available in C.

A.1. Key Operating Functions

A.1.1. X_GenerateRsaKeys

Generate a RSA key pair on the PC and convert it to the public/private key format supported by Fortress dongle.

```
int WINAPI X_GenerateRsaKeys(  
COS_RSA_PUBLIC_KEY *pCosPubKey,  
COS_RSA_PRIVATE_KEY *pCosPriKey);
```

Parameters:

pCosPubKey [out] Pointer to COS_RSA_PUBLIC_KEY struct which defines the format of RSA public key inside Fortress. For details, please refer to the Remarks section of "rsa_enc()" function above.

pCosPriKey [out] Pointer to COS_RSA_PRIVATE_KEY struct which defines the format of RSA private key inside Fortress. For details, please refer to the Remarks section of "rsa_enc()" function above.

Return values:

Return RE_SUCCESS if the function succeeds or corresponding error code otherwise.

Remarks:

If you want to perform the RSA calculation internally in the Fortress dongle, a RSA key pair of corresponding format must be provided. You can use this API to generate the two files which can then be loaded into the Fortress dongle.

A.1.2. R_GenerateRsaKeys

Generate a RSA key pair of PKCS#1 format on PC.

```
int WINAPI R_GenerateRsaKeys(
R_RSA_PUBLIC_KEY *pPubKey,
R_RSA_PRIVATE_KEY *pPriKey);
```

Parameters:

pPubKey [out] Pointer to R_RSA_PUBLIC_KEY struct, used to store the public key.

pPriKey [out] Pointer to R_RSA_PRIVATE_KEY struct, used to store the private key.

Return values:

Return RE_SUCCESS if the function succeeds or corresponding error code otherwise.

Remarks:

The RSA key pair generated by this function can be used in the software version cryptographic algorithms. It can also be of the format defined in KEYLOK[®] header files after some simple processing.

R_RSA_PUBLIC_KEY struct is defined as follows:

```
typedef struct {
    unsigned int bits;
    unsigned char modulus[MAX_RSA_MODULUS_LEN];
    unsigned char exponent[MAX_RSA_MODULUS_LEN];
} R_RSA_PUBLIC_KEY;
```

R_RSA_PRIVATE_KEY struct is defined as follows:

```
typedef struct {
    unsigned int bits;
    unsigned char modulus[MAX_RSA_MODULUS_LEN];
    unsigned char publicExponent[MAX_RSA_MODULUS_LEN];
    unsigned char exponent[MAX_RSA_MODULUS_LEN];
    unsigned char prime[2][MAX_RSA_PRIME];
    unsigned char primeExponent[2][MAX_RSA_PRIME_LEN];
    unsigned char coefficient[MAX_RSA_PRIME];
} R_RSA_PRIVATE_KEY;
```

Compared the two above structs with the structs of RSA key pair defined in header file sense4.h and you will find that the only difference between them lies on one struct member variable: unsigned int bits.

A.1.3. X_Pub2Cos

Convert a RSA public key of PKCS#1 format to the format supported in Fortress dongle.

```
int WINAPI X_Pub2Cos(
COS_RSA_PUBLIC_KEY *pCosPubKey,
R_RSA_PUBLIC_KEY *pPubKey);
```

Parameters:

pCosPubKey [out] Pointer to struct COS_RSA_PUBLIC_KEY, storing the public key of Fortress format.

pPubKey [in] Pointer to struct R_RSA_PUBLIC_KEY, storing the public key of PKCS#1 format.

Return values:

Return RE_SUCCESS if the function succeeds or corresponding error code otherwise.

Remarks:

None

A.1.4. X_Pri2Cos

Convert a RSA private key of PKCS#1 format to the format supported in the Fortress dongle.

```
int WINAPI X_Pri2Cos(  
COS_RSA_PRIVATE_KEY *pCosPriKey,  
R_RSA_PRIVATE_KEY *pPriKey);
```

Parameters:

pCosPriKey [out] Pointer to struct COS_RSA_PRIVATE_KEY, storing the private key of Fortress format.

pPriKey [in] Pointer to struct R_RSA_PRIVATE_KEY, storing the private key of PKCS#1 format.

Return values:

Return RE_SUCCESS if the function succeeds or corresponding error code otherwise.

Remarks:

None

A.1.5. X_Cos2Pub

Convert a RSA public key of the format supported in Fortress hardware to PKCS#1 format.

```
int WINAPI X_Cos2Pub(  
R_RSA_PUBLIC_KEY *pPubKey,  
COS_RSA_PUBLIC_KEY *pCosPubKey);
```

Parameters:

pPubKey [out] Pointer to struct R_RSA_PUBLIC_KEY, storing the public key of PKCS#1 format.

pCosPubKey [in] Pointer to struct COS_RSA_PUBLIC_KEY, storing the public key of Fortress format.

Return values:

Return RE_SUCCESS if the function succeeds or corresponding error code otherwise.

Remarks:

None

A.1.6. X_Cos2Pri

Convert a RSA private key of the format supported in Fortress hardware to PKCS#1 format.

```
int WINAPI X_Cos2Pri(  
R_RSA_PRIVATE_KEY *pPriKey,  
COS_RSA_PRIVATE_KEY *pCosPriKey);
```

Parameters:

pPriKey [out] Pointer to struct R_RSA_PRIVATE_KEY, storing the private key of PKCS#1 format.

pCosPriKey [in] Pointer to struct COS_RSA_PRIVATE_KEY, storing the private key of Fortress format.

Return values:

Return RE_SUCCESS if the function succeeds or corresponding error code otherwise.

Remarks:

None

A.2. Cryptographic Algorithm Functions

A.2.1. RSAPublicEncrypt

RSA public key encryption using PKCS#1 mode.

```
int WINAPI RSAPublicEncrypt(  
    unsigned char *output,  
    unsigned int *outputLen,  
    unsigned char *input,  
    unsigned int inputLen,  
    R_RSA_PUBLIC_KEY *publicKey);
```

Parameters:

output [out] Output cipher.

outputLen [out] Length of the output. It will be 128 if the RSA key is of 1024 bits.

input [in] Input plaintext

inputLen [in] Length of input. It can't exceed 117 if the RSA key is of 1024 bits.

publicKey [in] Pointer to struct R_RSA_PUBLIC_KEY, the public key used for encryption.

Return values:

Return RE_SUCCESS if the function succeeds or corresponding error code otherwise.

Remarks:

This function has the identical PKCS encryption as `_rsa_enc()`.

A.2.2. RSAPrivateDecrypt

RSA private key decryption using PKCS#1 mode.

```
int WINAPI RSAPrivateDecrypt(  
    unsigned char *output,  
    unsigned int *outputLen,  
    unsigned char *input,  
    unsigned int inputLen,  
    R_RSA_PRIVATE_KEY *privateKey);
```

Parameters:

output [out] Output plaintext.

outputLen [out] Length of the output. It can't exceed 117 if the RSA key is of 1024 bits.

input [in] Input cipher.

inputLen [in] Length of input. It will be 128 if the RSA key is of 1024 bits.

privateKey [in] Pointer to struct R_RSA_PRIVATE_KEY, the private key used for decryption.

Return values:

Return RE_SUCCESS if the function succeeds or corresponding error code otherwise.

Remarks:

This function has the identical PKCS encryption `_rsa_dec()`.

A.2.3. Sign

RSA signing using PKCS#1 mode.

```
int WINAPI Sign(
int digestAlgorithm,
unsigned char *plain,
unsigned int plainLen,
unsigned char *signature,
unsigned int *signatureLen,
R_RSA_PRIVATE_KEY *privateKey);
```

Parameters:

digestAlgorithm [in] Hash algorithm used in signing.

plain [in] Data to be signed

plainLen [in] Length of the data to be signed.

signature [out] Signature.

signatureLen [in] Length of the signature. It will be 128 if the RSA key is of 1024 bits.

privateKey [in] Pointer to struct R_RSA_PRIVATE_KEY the private key used for signing.

Return values:

Return RE_SUCCESS if the function succeeds or corresponding error code otherwise.

Remarks:

This function has the identical PKCS signing mode as `_rsa_sign()`.

A.2.4. Verify

RSA signature verification using PKCS#1 mode.

```
int WINAPI Verify(
int digestAlgorithm,
unsigned char *plain,
unsigned int plainLen,
unsigned char *signature,
unsigned int signatureLen,
R_RSA_PUBLIC_KEY *publicKey);
```

Parameters:

digestAlgorithm [in] Hash algorithm used in signing.

plain [in] Original data

plainLen [in] Length of the original data.

signature [in] Signature.

signatureLen [in] Length of the signature. It will be 128 if the RSA key is of 1024 bits.

publicKey [in] Pointer to struct R_RSA_PUBLIC_KEY, the public key used for signature verification.

Return values:

Return RE_SUCCESS if verification succeeds or corresponding error code otherwise.

Remarks:

This API does not have a corresponding Fortress function. The function `_rsa_veri()` can't directly handle the original data and thus has minor difference with this API. If you need the same functionality, you have to call `_sha1_xxx()` functions to complete the hash calculation first and then call `_rsa_veri` (the second mode) to verify the signature.

A.2.5. Digest

Hash algorithms, including SHA1, MD5, MD2.

```
int WINAPI Digest(  
int digestAlgorithm,  
unsigned char *plain,  
unsigned int plainLen,  
unsigned char *digest,  
unsigned int *digestLen);
```

Parameters:

digestAlgorithm [in] Hash algorithm used:

- DA_MD2 MD2
- DA_MD5 MD5
- DA_SHS SHA1

Plain [in] Message to be hashed.

plainLen [in] Length of the message.

Digest [out] Digest.

digestLen [out] Length of the digest.

Return values:

Return RE_SUCCESS if the function succeeds or corresponding error code otherwise.

Remarks:

None

A.2.6. DES

DES algorithm–ECB mode.

```
int WINAPI DES(  
unsigned char *key,  
int encrypt,  
unsigned char *output,  
unsigned char *input,  
unsigned int inputLen);
```

Parameters:

key [in] 8 octets secret key.

encrypt [in] flag, 1 means encryption and 0 means decryption.

output [out] output of encryption/decryption.

input [in] input data.

inputLen [in] Length of the input, must be multiple of 8.

Return values:

Return RE_SUCCESS if the function succeeds or corresponding error code otherwise.

Remarks:

When used for encryption, this API is the same as `_des_enc()` and when used for decryption, the same as `_des_dec()`.

A.2.7. TDES

TDES algorithm in ECB mode.

```
int WINAPI TDES(  
    unsigned char *key,  
    int encrypt,  
    unsigned char *output,  
    unsigned char *input,  
    unsigned int inputLen);
```

Parameters:

key [in] 16 octets secret key.

encrypt [in] flag, 1 means encryption and 0 means decryption.

output [out] output of encryption/decryption.

input [in] input data.

inputLen [in] Length of the input, must be multiple of 8.

Return values:

Return RE_SUCCESS if the function succeeds or corresponding error code otherwise.

Remarks:

When used for encryption, this API is the same as `_tdes_enc()` and when used for decryption, the same as `_tdes_dec()`.

A.3. Error Code Index

RE_SUCCESS	0x0000	Operation successful
RE_CONTENT_ENCODING	0x0400	Wrong content encoding
RE_DATA	0x0401	Wrong data
RE_DIGEST_ALGORITHM	0x0402	Invalid digest algorithm
RE_ENCODING	0x0403	Wrong encoding
RE_KEY	0x0404	Invalid key
RE_KEY_ENCODING	0x0405	Wrong key encoding
RE_LEN	0x0406	Wrong length
RE_MODULUS_LEN	0x0407	Wrong modulus length
RE_NEED_RANDOM	0x0408	Random number needed
RE_PRIVATE_KEY	0x0409	Error in private key encryption
RE_PUBLIC_KEY	0x040a	Error in public key decryption
RE_SIGNATURE	0x040b	Error in signing
RE_SIGNATURE_ENCODING	0x040c	Wrong signature encoding
RE_ENCRYPTION_ALGORITHM	0x040d	Wrong encryption algorithm