ProtectToolkit FM SDK
Programming Guide

THE
DATA
PROTECTION
COMPANY

Technical Support

If you encounter a problem while installing, registering or operating this product, please make sure that you have read the documentation. If you cannot resolve the issue, please contact your supplier or SafeNet support. SafeNet support operates 24 hours a day, 7 days a week. Your level of access to this service is governed by the support plan arrangements made between SafeNet and your organization. Please consult this support plan for further information about your entitlements, including the hours when telephone support is available to you.

Contact method	Contact information	
Address	SafeNet, Inc. 4690 Millennium Drive Belcamp, Maryland 21017 USA	
Phone	United States	(800) 545-6608, (410) 931-7520
	Australia and New Zealand	+1 410-931-7520
	China	(86) 10 8851 9191
	France	0825 341000
	Germany	01803 7246269
	India	+1 410-931-7520
	United Kingdom	0870 7529200, +1 410 931-7520
Web	www.safenet-inc.com	
Support and Downloads	www.safenet-inc.com/Support Provides access to the SafeNet Knowledge Base and quick downloads for various products.	

Technical Support Customer Portal	https://serviceportal.safenet-inc.com Existing customers with a Technical Support Customer Portal account can log in to manage incidents, get the latest software upgrades, and access the SafeNet Knowledge Base

Revision History

Revision	Date	Reason
A	27 October 2014	Release 5.0

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1 Overview

A Functionality Module (FM) is custom-developed, customer-specific code that operates within the secure confines of a Hardware Security Module (HSM). You can use the PTK FM SDK to develop FMs for the ProtectServer External 2 (PSE2) and ProtectServer Internal 2 (PSI-E2) HSMs that were introduced in PTK 5.0. The slots on a PTK client workstation may contain either local (PSI-E2) HSMs or remote (PSE2) HSMs. A mix of local and remote HSMs on a single PTK client workstation is not supported.

Note: The PTK 5.0 software, including the PTK FM SDK, is not supported on the legacy ProtectServer HSMs (PSe, PSI-E, and ProtectServer Gold)

FMs allow application developers to design security sensitive program code which can be downloaded into the HSM to operate as part of the HSM firmware. This functionality may be required to implement custom algorithms, or to isolate security sensitive code from the host environment. FMs can make full use of the HSM functionality, which is provided using a PKCS#11 compliant Application Programming Interface (API). The PTK FM SDK allows developers an extensive opportunity to create a large range of customized high security applications.

To assist in the development of FMs, the PTK FM SDK contains support for FM emulation on the Host System.

This document is intended for software developers, as a technical reference which describes the programming methodologies and functions used for the development of Functionality Modules and host-side applications.

Features

Host apps are supported on all platforms that the PTK SDK is supported on. FMs have to be cross compiled on Linux. The FM SDK provides the following components:

- Sample FM code
- Sample host-side code
- Build scripts
- FM binary image generation tools (Linux only)
- Host–side libraries
- FM libraries (Linux only)
- FM emulation libraries (Linux only)
- Java classes to access HSMs
- Java docs

The PTK FM SDK requires PTK-C.

Chapter 2
Installation

In order to use the FM SDK for building FMs or Host applications, you must install both the PTK FM SDK and PTK-C SDK software on your development workstation. On Linux, you must also install the FM toolchain.

Note: You cannot install the PTK runtime and FM SDK on the same workstation. It is recommended that you do your FM development on a separate workstation.

For details regarding the installation or uninstallation of the software discussed in this manual, please refer to the following documents:

- *ProtectToolkit C Installation Guide*

Installing the FM SDK

All of the software required to install and configure the FM SDK is included on the PTK software DVD.

Before you use the FM SDK, you must configure your environment, as described in "Environment Setup" on page 3.

Windows

To install the FM SDK on Windows, you must install the PTK FM SDK and PTK-C SDK software. Before using the FM SDK, you must run a script to configure your environment, as described in "Environment Setup" on page 3.

To install the FM SDK software

1. Log in to the computer as the Administrative user.

2. Install the PTK-C software, as described in the *PTK-C Installation Guide*.

3. Go to the folder for your architecture on the installation DVD:

   64-bit: `cd <DVD_root>\SDKs\Win64`
   32-bit: `cd <DVD_root>\SDKs\Win32`

4. Run the following **.msi** file to install the PTK FM SDK software:

   ```
   fm_sdk\PTKfmsdk.msi
   ```

To configure your FM SDK environment

1. See "Environment Setup" on page 3.

Linux

To install the FM SDK on Linux, you must install the PTK FM SDK and PTK-C SDK software. Before using the FM SDK, you must run a script to configure your environment. You can add the script to your startup file (for example, .bashrc) to have it set up the environment each time you open a shell, or you can source the script each time you open a shell that you will use for FM development. See "Environment Setup" on page 3.

To install the FM SDK software

1. Log in to the computer as the Administrative user.

2. Install the PTK-C software, as described in the *PTK-C Installation Guide*.

3. Go to the folder for your architecture on the installation DVD:

   64-bit: `cd <DVD_root>\SDKs\Linux64`
   32-bit: `cd <DVD_root>\SDKs\Linux`

4. Install the **RPM** file for your architecture to install the PTK FM SDK software:

   64-bit: `ptkc_sdk/PTKfmsdk-<release>.x86_64.rpm`
   32-bit: `ptkc_sdk/PTKfmsdk-<release>.x86.rpm`

To configure your FM SDK environment

1. See “Environment Setup” on page 3.

Chapter 3
Environment Setup

FM developers should ensure that their development environment is configured correctly and that all required files and library locations are set. This chapter is provided as a guideline for setting up the development environment so that required files can be accessed during the FM compile and link routines.

Environment Variables

In order to be able to use the build scripts, the following environment variables are used:

• CPROVDIR	Specifies the installed location of the Cprov SDK (PTK-C)
• FMDIR	Specifies the installed location of the FM SDK

Setting the Environment Variables

The environment variables are set using scripts.

To set the environment variables on Linux

1. Go to the PTK software installation directory:

```
cd /opt/safenet/protecttoolkit5/ptk
```

2. Source the **setvars.sh** script:

```
. ./setvars.sh
```

To set the environment variables on Windows

1. Go to the PTK FM SDK software installation directory:

```
cd <fmsdk_install_dir>\bin
```

2. Run the **fmsdkvars.bat** script:

```
. ./fmsdkvars.bat
```

Windows Environment Variable Paths

Please note that the Windows build scripts cannot handle space (“ ”) character in the environment variables mentioned above. If the paths to the install locations contain a space in the directory name (e.g. “C:\Program Files\SafeNet\Cprov SDK”), you should use the short names of the directories that contain spaces (e.g. “C:\Progra~1\SafeNet\CprovS~1”). The short format of the directory names can be discovered using the ‘/x’ switch in a dir command. For example, you can use “dir /x c:\progra*” command to discover the short name of the “Program Files” directory.

Note: If you are using the provided FM SDK **fmsdkvars.bat** script, the paths are already converted to their short form.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4
FM Architecture

FM Support within the HSM Hardware

FMs allow application developers to design security sensitive program code which can be downloaded into the HSM and operate as part of the HSM firmware. The PTK FM SDK also provides application developers with APIs to develop applications on a host to interface to the HSM.

The FM may contain custom-designed functions which then access the Cryptoki library to perform cryptographic operations. Alternatively, the FM may contain functions that conform to the PKCS#11 standard and contain additional operations that are performed prior to passing the request to the Cryptoki library. The former are referred to as custom functions and the latter are referred to as patched PKCS#11 functions.

The following diagrams show the various components of the FM system, relevant to the host and HSM:

- the first diagram shows the system where the API and FM utilize custom functions.
- the second diagram shows the system where the API utilizes PKCS#11 functions and the FM contains patched PKCS#11 functions.

These two scenarios are discussed for two reasons:

- FMs tend to be used in environments where either only custom functions or only PKCS#11 functions are utilized.
- for ease of illustration.

However, there is no constraint that prevents a FM from containing a mixture of both custom and patched PKCS#11 functions.

Each figure marks the boundaries of the host system and the adapter in order to clarify where each component resides. The boxes represent components and the arrows represent the interaction or data flow between the components. Only the message request path is shown in the diagrams, as this method allows illustration of which component originates the interaction. The message response follows the same path but in the opposite direction and is not shown on the diagram. The names given to these interfaces are directly, or indirectly related to the libraries provided in the PTK FM SDK.

The notation adopted to identify the data flows utilized in the diagram is as follows:

API (Function Type)

where API refers to the API used to interface between the two components and function type indicates the type of function.

For example in Figure 1, the arrow labeled MD (Custom Functions) indicates the flow of Custom function request packages that are passed between components via the Message Dispatch (MD) API (refer to Chapter 7 for details). EPP (Custom Functions) refers to Custom function request packages that are passed from the host across the PCI bus, in the case of local HSMs, or across a TCP/IP link, in the case of remote HSMs, via an unpublished SafeNet-proprietary protocol.

A more detailed discussion detailing the components and their interactions follows each diagram.

Custom Functions

Figure 1 details the components contained in the Host system and the HSM when using custom functions. The custom application is executed on the host system. A user defined protocol specifying the message response and request packages for each function must be defined by the application developer. This protocol is used to access the FM's custom functions. The host requests are transparently communicated directly to the FM module, which is then responsible for implementing the protocol.

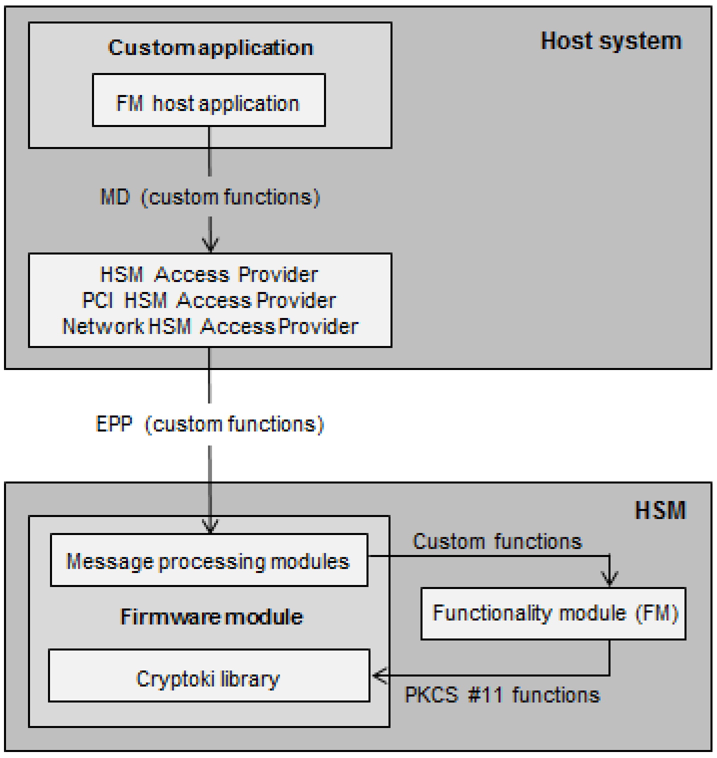

Figure 1 - The components and interfaces in a system using Functionality Modules for Custom Functions

These message response and request packages are transferred between the application and the PCI or NetServer HSM Access Provider, via the Message Dispatch (MD) API. The legacy FM Host API has been deprecated since Version 2.0 of ProtectProcessing Orange (legacy name of the FM SDK). As of FM SDK 5, the Host API functions are no longer supported nor maintained. Since the previous Host API library makes use of MD API to communicate with the HSM, existing binaries using the the old API should continue to function, but support will no longer be provided for developers not using the MD API.

The PCI driver provides the interface to the PCI bus and is used in systems which deploy local HSMs. The NetServer Driver provides the interface to the WAN/LAN network and is used in systems which deploy remote HSMs. It is not possible for a system to utilize remote and local HSMs at the same time. At configuration time, either the PCI or NetServer HSM Access Provider is specified as appropriate to the installation (refer to the HSM Access Provider Install Guide for details).

In the HSM, the message request/response is processed via modules collectively referred to here as the Message Processing Modules. Any message request/response which contains a custom function is passed to the FM for processing. The custom function can access the cryptographic functionality provided in the firmware via PKCS#11 function calls. FM functions have access to any of the Serial, C Run Time and original PKCS#11 functions from HSM firmware.

PKCS#11 Patched Functions

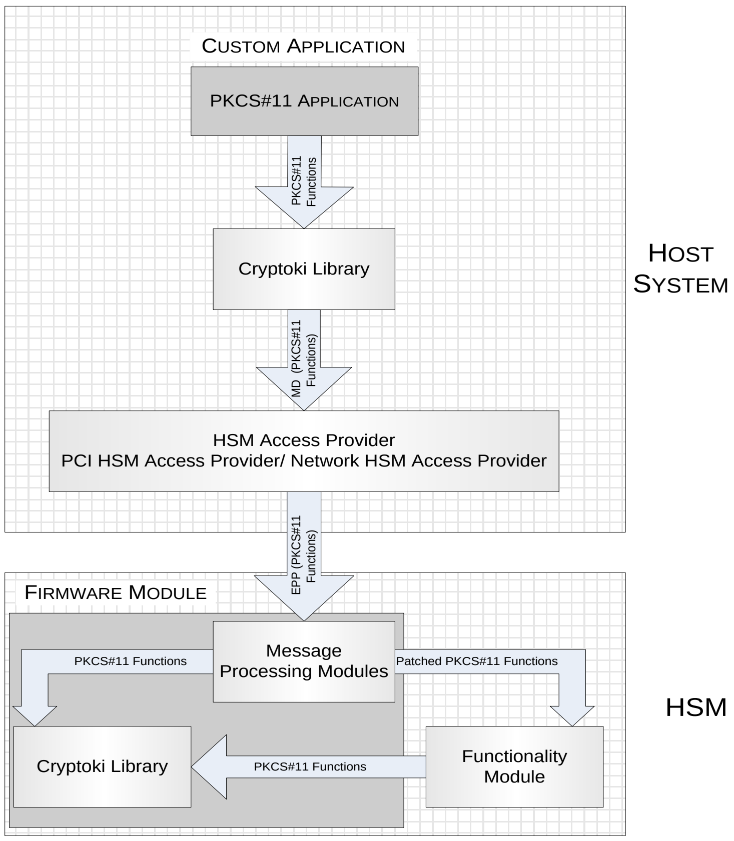

Figure 2 - The components and interfaces in a system using Functionality Modules for Custom Functions

Figure 2 details the components contained in the Host system and the HSM when using PKCS#11 functions. The custom application is executed on the host system. The application accesses patched PKCS#11 functions in the FM and the standard PKCS#11 functionality of the Cryptoki library provided in the firmware module via a standard PKCS#11 interface provided by the Cryptoki library on the host system.

The Message Processing Modules contain a list of patched PKCS#11 functions, to which the incoming function is compared. The Message Processing Modules call the PKCS#11 function from HSM firmware if the function isn't patched or gives control to an FM version of same function if it is patched. FM implementations of patched functions can call any of the Serial, C Run Time and original PKCS#11 functions from the HSM firmware. The PKCS#11 functions called from within the FM call the firmware implementation directly; bypassing the Message Processing Modules.

An FM can implement both Custom functions and PKCS#11 patched functions simultaneously if desired.

FM Support in Emulation Mode

In emulation mode, all components run on the host system. The diagram below shows the various components when the FM is executed in emulation mode.

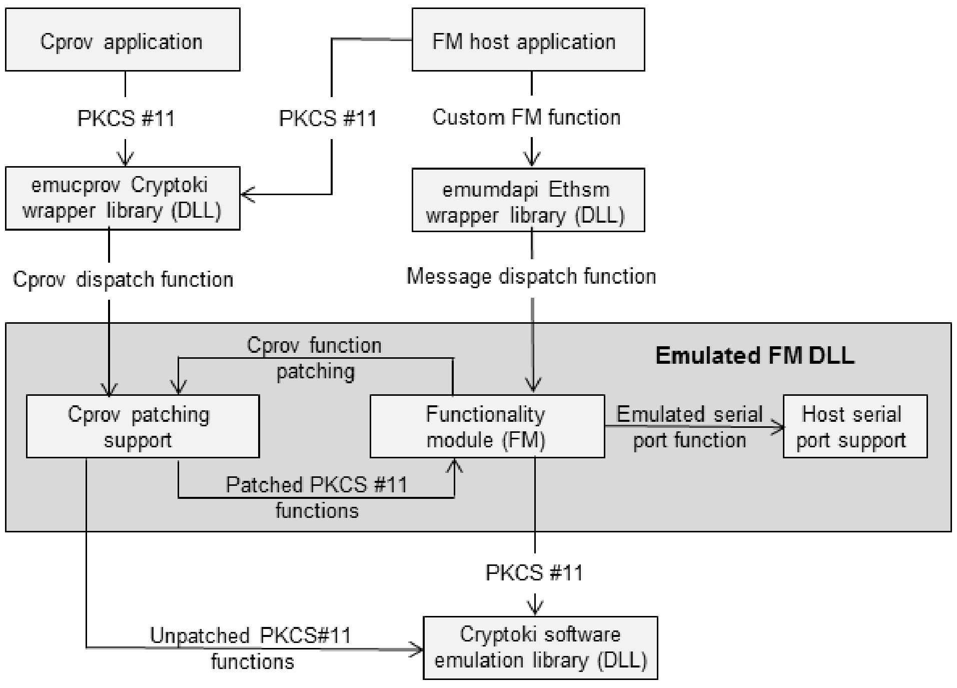
Cprov application
FM host application
PKCS #11
PKCS #11
Custom FM function
emucprov Cryptoki wrapper library (DLL)
emumdapi Ethsm wrapper library (DLL)
Cprov dispatch function
Message dispatch function
Emulated FM DLL
Cprov function patching
Cprov patching support
Functionality module (FM)
Emulated serial port function
Host serial port support
Patched PKCS #11 functions
PKCS #11
Unpatched PKCS#11 functions
Cryptoki software emulation library (DLL)

Figure 1 - The components of a system emulating Functionality Modules

The above figure shows the components of the FM emulation system. The Functionality Module is combined into a DLL with emulation libraries to provide FM SDK features such as the Cprov function patching and serial port access.

Applications that need to observe the effects of the FM, via PKCS#11 patching or custom functions, are run against emulation wrappers for cryptoki and ethsm. These wrapper libraries are built from source as part of the emulation FM build and result in dynamic libraries. This allows existing applications to run against the emulation FM.

The emulation wrapper libraries in turn load the emulated FM, which loads the PTK Software Emulation Cryptoki library. Messages are routed as per the above diagram by the emulation libraries the FM is linked against.

Emulation Mode Limitations

Supported Platforms

FM Emulation using the PTK FM SDK is supported on Linux only.

Applications being run against an emulation FM must be run locally. Netserver cannot be used to allow remote applications to connect to the emulated FM and cryptoki wrapper.

Supported C APIs

Emulation mode uses the standard C library installed on the host. The PTK FM SDK is C99 compliant, extended to support the following non-standard APIs:

ctype.h
isascii, toascii

string.h
strdup, strsep

Any code written using these APIs will fail in emulation mode unless you explicitly instruct your complier to allow them. The following example illustrates how to allow these non-standard APIs in emulation mode using GCC.

If your emulation environment uses glibc, such as the standard gcc environment on Red Hat, you may define the following to enable these non-standard libc functions in emulation mode: Note that we test for gcc not glibc because the compile time glibc flag is set in features.h, but we need to set override _GNU_SOURCE before features.h is included.

Important: Since every GNU header includes features.h, you must put this at the top of any source files you wish to use the extended functions in, or it will not be applied.

```
#if defined(EMUL) && defined (__GNUC__)
/*
* Define _GNU_SOURCE flag to enable non-standard API's that are
supported in the FM LIBC
* but not necessarily by all EMUL environments:
* ctype.h: isascii, toascii
* string.h: strdup,strsep
*/
#define _GNU_SOURCE
#define EXTENDED_OK
#endif
```

Source Level Debugging

When debugging an emulation FM you will be able to step through the application and into the message encoding function.

The emulation versions of the MD_Initialize and C_Initialize function call the FM’s Startup function. The startup function will only be executed once.

The emulation version of the MD_SendReceive function calls the FM’s Dispatch entry function via the ethsm emulation layer.

However, when the message encoding function calls the MD_Initialize or MD_SendReceive functions you will not be able to step into these functions because no symbols or source code is supplied.

The best method to step through your FM code is to set a break point at the start of the Startup and Dispatch entry point functions.

Random Number Generator

The emulation Cryptoki library does not provide true random numbers. Although the FIPS 140 approved Pseudo Random Number Generator is implemented in the emulation version there is insufficient entropy to make good quality random numbers.
Do not use the emulation to create production grade keys.

Tampering the Secure Memory

The emulation Cryptoki library does not support he concept of a hardware tamper event. You may delete the emulation Cryptoki data directory to simulate a total loss of secure memory. However you should only do this when the Cryptoki library is not running.

The emulation Cryptoki data directory default location is

Windows:	c:\cryptoki
Linux:	~/.cryptoki/cryptoki

The emulation data directory can be changed via the ET_PTKC_SW_DATAPATH entry in the PTK configuration file. See the *ProtectToolkit-C Administration Guide* for more information.

Cryptoki Function Patching

The emulation is capable of supporting Cryptoki function patching of any application that is run against the emulated cryptoki wrapper built with the emulated FM.   Unlike Protect Processing 3.0 and earlier releases, applications do not have to be recompiled with the emulated libraries, they simply have to have the emucprov and emumdapi wrapper libraries in the library search path ahead of the real cryptoki library.

ETHSM

The timeout parameter of MD_SendReceive() is ignored in emulation mode.

Cryptoki FM Object

Due to the way the emulated FM is linked with the application and cryptoki libraries, it does not appear as an object via cryptoki.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5 Development Lifecycle

Overview

The following diagram illustrates the recommended development cycle to be undertaken when developing functionality modules for a ProtectServer HSM.

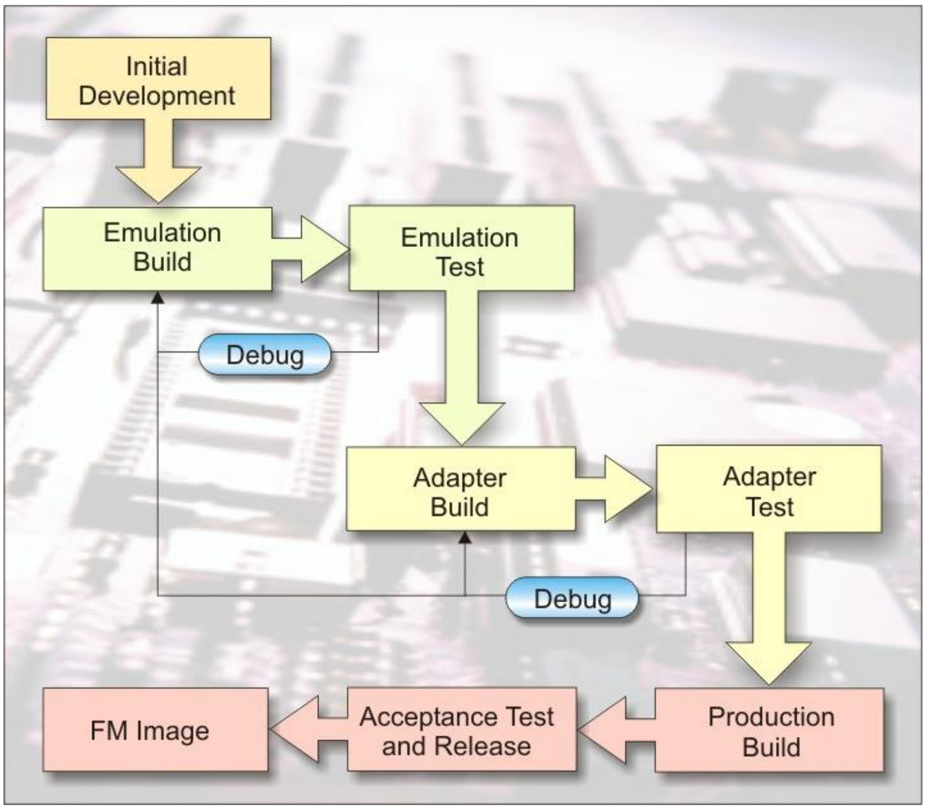

Figure 2 – FM Development Lifecycle

As can be seen, the flow of development activities comprise the following stages:

• Initial Development:	Includes the design and development of the functionality application code for initial testing.
• Emulation Build:	This process compiles the FM code for the emulation environment.
• Emulation Test:	The FM which was produced during the previous steps should now be completely tested in the emulation mode to ensure correct operation. Should errors be found in the emulation build, the developer should the familiar build/test/debug cycle until successful operation can be confirmed.
• Adapter Build:	The functionality module should now be built for the HSM environment. The FM is downloaded onto the HSM hardware and again tested to ensure that operation is as expected.

- Production Build: The final process during the development lifecycle will be to produce and release the functionality module for the operational environment it was intended.

Initial Development

This phase is the start of the development phase from the specification of the requirements to the completion of the design. Programming the FM, and the host side libraries and or application can also be considered part of this stage. It is assumed that at this stage the test procedures are also developed.

Emulation Build

In this phase, the FM, and the host-side libraries and applications are built for the emulation environment.

This stage is complete when the FM executes correctly within the emulation DLL, and the host-side executables (and possibly the DLLs) are generated successfully.

Emulation Test

Prior to this stage, the test procedures must have been created. The emulation binaries generated in the previous stage are used to execute the test procedures and determine whether the FM satisfies its requirements. Since the message dispatching code is not compiled in the emulation build, the tests that are aimed to detect problems in serialization of data, do not need to be performed in this stage.

The development usually stays in the Emulation Build/Emulation Test loop until all the problems detected are fixed. During this stage, developers are encouraged to use a debugger to step through the FM as well as the host source code, to make this task easier.

Adapter Build

After the emulation build passes all the tests, the FM must be tested in the HSM. Although the emulation build is a very close approximation of the HSM environment, there are components such as function patching and the dispatch/retrieval of messages between the host and HSM, which are not tested during the emulation test stage.

In this phase, the developer generates the binary FM image (refer to Chapter 5 for details), and signs it using either a temporary or a permanent development key. Once the image is signed, it can be downloaded to the HSM for the next stage of testing.

Adapter Test

In the HSM test stage, the development build of the FM is tested in its production environment. The tests which are performed in the emulation environment should also be repeated, in order to validate the implementation.

If problems are detected in this stage, the developer may choose to resolve the problems either in the emulation test stage or the HSM test stage. The choice usually depends on the seriousness of the problem and the area in which the problem was detected. Since the message serialization code is not compiled in the emulation environment, it is unnecessary to go back to emulation build stage when a problem in this area is detected.

Production Build

When convinced that the implementation of the FM and the host side code is correct, a production build of the system is performed (refer to Chapter 5 for details).

In this stage, the developer generates the FM binary image, and the responsible person signs it using the production private key.

Acceptance Test

When the production binaries are available, the acceptance tests are performed on the final system before the binaries are released.

Key Management

All FM images downloaded to the HSM must have an assigned signature. FMs are only executed inside the HSM after this signature has been validated. The management of the keys used to sign/verify the firmware is completely controlled by the developers of the FM. SafeNet does not have any responsibility for the FM key management scheme.

The certificate that is used to validate the FM binary image must exist in the Admin Token of the HSM on which the FM is to be installed. If the certificate does not already exist in the Admin Token, the Admin Token Administrator will be required to install the certificate in the Admin Token. Furthermore, the verification and downloading of the FM requires the HSM Administrator to provide the Admin Token password, enforcing the presence of the HSM Administrator at the time of the download operation.

As previously advised, there is no pre-defined key management scheme for the private key and the certificate. Therefore, one of the first things to be performed by the FM developer is to decide on the key management scheme to be used in the system.

Example Key-Management Scheme

This section outlines a sample approach to a key management scheme, which can be customized and extended.

It is recommended that the key used to sign FMs in the Adapter Build phase is not the same as the key used to sign it in the Production build phase. This would ensure that a FM in the Adapter Build or Adapter Testing phase cannot be used by end-users or customers. Additionally, the usage of a production-level FM signing key needs stricter access control requirements compared to the development signing keys. Using this key to sign FM images in Adapter Build phase would therefore make the task of development more difficult.

The easiest key management scheme for development keys is to generate a new self-signed key/certificate pair every time the FM image is created. This can be done using the ProtectToolkit C ‘ctcert’ tool. Please note that the signing key cannot be used from the Admin Token because of limitations on how the ‘mkfm’ utility addresses keys. Therefore, the key/certificate pair must be created on another token and the certificate must be imported into the Admin Token. Importing can be done either by backing up the certificate on smartcards and restoring it to the Admin Token, or exporting the certificate to a file and then re-importing it into the Admin Token using the ‘ctcert’ tool.

After the certificate is in the Admin Token, the Admin Token SO must login to the token and mark the certificate as “trusted”. This can be achieved using the ctcert tool.

After the certificate is marked as trusted, the raw FM binary image can be signed using the private key generated, as discussed previously. This can be achieved using the mkfm utility. The signed FM image can then be downloaded into the HSM using the ctconf utility.

There must be a non-development HSM to hold the production key/certificate pair. The key/certificate pair used to sign a production FM can use the most appropriate of the following three approaches:

Self Signed certificate

This scheme does not provide any authentication of the FM. However, it is very easy to setup and use. If the certificate must be handed to a third party, it must be done using a trusted channel – treating the certificate as a secret key. This scheme is most suitable for companies developing FMs for internal use only.

Certificate signed by a trusted third party

The signing key and the certificate are obtained from a trusted third party CA. This scheme ensures the authenticity of the certificate, and allows the certificate to be transmitted to another party over an untrusted channel.

Use of a local CA signing key

This scheme requires the FM developers to obtain a signing key/certificate from a trusted third party for local CA operations. Then, another signing key/certificate pair is generated locally for signing production level FMs. This allows multiple signing keys to be created, authenticating each FM separately.

Chapter 6 Sample FMs and Development Tips and Techniques

This chapter is intended as a tutorial and reference for developing functionality modules. There are a number of samples provided as part of the FM development kit.

Contents of the $(FMSDK) Directory

When installed for FM development (as opposed to host only development) the $(FMSDK) directory contains the following:

Directory or file	Description
bin/	Non-toolchain utilities and tools for creating an FM
Doc/	FM Development documentation
Include/	Header files specific to FM and FM Host Application development. These are used together with headers provided by PTK-C in $(CPROVDIR)/include.
lib/ppcfm/	Static libraries used for building an FM that will run in a PCIe2 HSM.
lib/linux-i386/	Static libraries used for building an emulation mode FM that will run on the host computer.
samples/	Sample FMs and FM Host applications descripbed below.
src/emul/	Source files used for building wrapper cryptoki and ethsm libraries required with FM emulation.
cfgbuild.mak	Common configuration makefile that sets up makefile and toolchain variables and rules required for building an FM. This should be included at the top of any FM’s makefile.

SDK Installation Tips

When installing the FM SDK, it defaults to a system installation path.   As long as CPROVDIR and FMDIR are set to the system paths, the samples may be copied elsewhere so they can be built and modified by a non-root user.

Protecting Data Storage of FM

When the FM is used to extend the HSM functionality, usually there would be data that needs to be protected by the HSM. Normally, this data would be stored as a Cryptoki Object in one of the tokens. However, protecting the contents of these objects poses a problem, because, setting the SENSITIVE attribute on the object would prevent access to the data from the FM, and leaving it open would allow any PKCS#11 application on the host side to get the contents of the data.

There are three possible solutions to this problem –

- Token blocking

- Using Privilege Level
- Using the SMFS

Token Blocking

As shown in Sample “restrict”, the FM can patch the C_OpenSession() PKCS#11 function, and prevent any session to be opened to the slot that contains the objects used by the FM. In PKCS#11, all of the functions that can access or export the object need a session handle to the token containing the object, and the FM patches the C_OpenSession() to prevent applications from obtaining the required session handle. This method effectively reserves one or more tokens of the HSM for the FMs internal use, and prevents any kind of access to the contents of this token from the host side.

Using Privilege Level

The CT_SetPrivilegeLevel function allows a simple solution to the problem stated above. As shown in Sample “xorsign”, the FM can make a call to temporally obtain the rights to read Sensitive object attributes.
This allows the FM designer to create and manage her keys using the tools provided with the HSM but such that they are safe from outside programs but still be able to access them from the trusted FM.

Using the SMFS

The Secure Memory File System provides access to the same low level key storage facility. By creating a new application directory the FM designer can store keys without these keys being visible through the HSMs Cryptoki interface.
The format of the keys is entirely up to the FM designer – they need not have attributes as the Cryptoki objects do.
There is no need to call C_Initialise or open sessions or search for object handles if you use the SMFS to store your keys.
FMs that store their keys in SMFS need to provide all the functions to generate, store, delete, backup and restore these keys.
FMs that open a SMFS file and keep the handle open should note the following.
On the first time (since the HSM was restarted) an application calls C_Initialize the HSM firmware will close all SMFS handles. So if you open a SMFS file during Startup the next C_Initialise call will close the file. Also the number of SMFS file handles is a limited resource (approx 16).
Therefore FM designers should not keep SMFS file handles. Instead use SMFS to backup the keys only.
Keep the keys in normal memory while the FM is running. Restore the keys fom SMFS during the FM initialization by opening/reading and closing the SMFS file. When changes are made to the keys then open/write/close the SMFS file to backup the changes.

Cprov Function Patching

Downloading bad FM code into the HSM could make the device unusable. Patching functions such as C_Initalize, C_OpenSession, C_Login and C_VerifyXXX must be done with extreme care.

One of the techniques is to put safety switches in the startup function, as seen in Sample ‘safedebug.

FM Message Dispatching

FM Message dispatching support allows for more than one request buffer, and more than one reply buffer to be presented to the HSM. The message dispatch layer provides scatter-gather support to combine all the request buffers into a single data buffer, and send it to the HSM. The reply data is treated the same way, but in the reverse direction; i.e. the data is scattered into multiple reply buffers. This feature can be very useful when information to be sent to the

HSM and the information received from the HSM are kept in different variables and / or buffers.

The scatter-gather operation on the reply buffers can behave in an unintuitive manner when the initial buffers are variable length. The device driver will start filling the host side initial buffers with the reply data and it will not place any data into subsequent buffers until the current one is completely filled. The effect of this is that the reply buffer fields may not contain the expected values when the amount of data placed in a variable length buffer is less than the maximum length of the buffer.

For example, if two reply buffers of 40 bytes each are passed to the message dispatch layer but the actual data to be returned in each buffer is only 32 bytes then the first 40 byte buffer will be filled with 32 bytes of data meant for the first buffer and 8 bytes of data meant for the second buffer. The second reply buffer of 40 bytes will only contain 26 bytes of data.

There are two possible cases to handle this behavior: -

3. After receiving the reply, re-align the data in the buffers. The order of re-alignment must be from the last buffer to the first. In order to be able to implement this, the reply data, in its entirety, must contain enough information to determine the length of each reply block.
4. Always merge the reply buffers to a single block before dispatching the request, by allocating another block, and moving data from the allocated buffer to the caller's reply buffers. This approach makes the code more reliable.

Handling Host Processes

The FM SDK allows a FM developer to determine the identity of processes sending messages to the HSM.

The functions FM_GetCurrentPid and FM_GetCurrentOid allow you to know what process is sending the current message. You must use a combination of both PID and OID to uniquely identify a process i.e. if two callers have the same PID but different OIDs then they should be seen by the FM as different processes.

If your Functionality Module supports the concept of a user login then you will need to track which host processes have logged in.
Therefore you can remember which process has logged in by storing the PID and the OID as the process successfully authenticates. When a process sends a message that requires authentication you can check to see if the process is the list of authenticated processes.

The Cryptoki system always uses the PID/OID to determine if a session handle or object handle is valid for the calling process.
Therefore if the FM makes Cryptoki calls while processing a request by and it is using a session handle obtained earlier from a different request then there is a possibility that the Cryptoki call with fail with CKR_CRYPTOKI_NOT_INITIALIZES error.
This is because process A calls the FM which then calls C_Initialize and opens a Cryptoki session. Then later Process B calls the FM and the FM tries to use the session handle. The Cryptoki will not recognise Process B. To overcome this problem you may want to modify the P ID and OID to a constant value that the underlying Cryptoki sees by using the FM_SetCurrentPid and FM_SetCurrentOid calls prior to making any Cryptoki calls.

The value –1 for PID and OID is a suitable choice for this purpose.

Memory Alignment Issues

The PowerPC processor in the PSI-E2 and PSE2 does not require fully aligned memory access, however unaligned access incurs a performance cost.

Memory Endian Issues

The processor in PSIe2 is big endian, where the processors in PSIe and PSG are little endian.

It is recommended that FM developers use the provided endian macros to encode all messages in network byte order. By using the endian macros on both host and FM, endian differences between host and hsm are not an issue.

The utility endian macros are provided in the PTK-C header file **endyn.h.**

FM Samples

There are eight sample FMs provided with the SDK. Most have similar file layout.

The sample files included as part of the FM development kit consist of six sample FMs and two examples of how to communicate to the HSM from java.

Each of the FM samples are structured in a similar way. In each sample directory there is –

• Makefile	- makefile to build host and HSM side code
• Fm	- directory holding HSM side source
• Host	- directory holding host side source
• Include	- optional directory to hold common header files

Within the FM directory are files like these -

• hdr.c	– header file for the production build of the FM binary image.
• sample.c	– HSM side; main source for FM
• Makefile	– Makefile to build the FM and the application

Within the host directory are files like this -

• stub_sample.c	– host side stub (request encoder/decoders) (needed only for custom API)
• sample.c	- main source for host side test application
• Makefile	– Makefile to build the host-side application for emulation, or production.

The samples are built using gnu make and the provided Makefiles. When working on a platform that has a native gnu make, such as Linux, you can use the system make command. When building the host part of the samples on Windows, a copy of gnumake is provided in <fm_installation_path>\bin.

- Production build, no debug information in binaries:
  make
- Production build, with debug information in binaries:
  make DEBUG=1
- Emulation build, no debug information in binaries:
  make EMUL=1
- Emulation build, with debug information in binaries:
  make EMUL=1 DEBUG=1

Binary files generated by the above variants are placed in different directories. Therefore, all variants can be generated in the same directory. The directory names used are:

obj-win32:	Binaries for the production, non-debug host build on win32 environment
obj-win32d:	Binaries for the production, debug host build on win32 environment
obj-linux-i386:	Binaries for the production, non-debug host build on Linux/i386 environment
obj-linux-i386d:	Binaries for the production, debug host build on Linux/i386 environment
obj-ppcfm:	Binaries for the production, non-debug FM build for the HSM environment
obj-ppcfmd:	Binaries for the production, debug FM build for the HSM environment
obj-linux-i386e:	Binaries for the emulation, non-debug FM build on Linux/i386 environment
obj-linux-i386ed:	Binaries for the emulation, debug FM build on Linux/i386 environment

The binaries generated from each variant can be deleted using the target ‘clean’. E,g,:

```
make EMUL=1 clean
```

The build scripts will generate the unsigned FM binary image when the HSM builds are performed. The binary images are named ‘<samplename>’ without any extensions. Since these images are not signed yet, it is not possible to download them to the HSM. In order to use the key management scheme #1 (i.e. using self-signed FM download certificates), you can follow the steps listed below:

1. Generate the key/certificate pair on the first token. Execute:

   ```
   ctcert c –s0 -k -trsa -z512 -lfm
   ```

   from a command prompt. This will generate a 512 bit RSA key pair, and a self-signed certificate. The key length is kept small in order to obtain a slight speed-up in the development phase. The production keys must be at least 1024 bits long. The labels of the generated keys will be as shown below:

Private Key:	fm (Pri)
Public Key:	fm (Pub)
Certificate:	fm

2. You must copy the certificate to the Admin Token. This can be done using by using `ctcert.exe` to export the certificate to a file.

   ```
   ctcert x –lfm –s0 –ffmcert.crt
   ```

3. Then import the certificate to the Admin Token. For this operation, the password of the Admin Token is required.

   ```
   ctcert i –ffmcert.crt –s2 –lfm
   ```

4. You must mark the certificate as “Trusted”. This can be done using the ‘ctcert’ utility with the ‘t’ command line option.

   ```
   ctcert t –lfm –s2
   ```

   Please refer to the ProtectToolkit C Programmers guide for full details of the CTCERT utility.

5. Now, the binary image can be signed using `mkfm.exe`. In the directory where the binary image is generated, execute the following command:

   ```
   mkfm –k “<Token Label>/<fm (Pri)>” -fsampleN –osampleN.fm
   ```

where “<Token Label>” is the label of the token in Slot 0, <fm (Pri)> is the label of the private signing key that was previously generated and sampleN is the binary image of the sample FM being signed. This will generate a signed FM binary image, named “sampleN.fm”. This command requires the user password of the token to be entered.

6. Exit from all cryptoki applications that are still active, and download the FM image to the HSM. Execute:

   ```
   ctconf –bfm –jsampleN.fm
   ```

   where “fm” is the name of the certificate in Admin Token used to verify the FM binary image integrity. After a while, the command will report a successful download. The download operation can be checked by executing the command:

   ```
   ctconf –s
   ```

   and ensuring that the FM name is correct, and the FM status is “Enabled”.

When running FMs in emulation mode, the HSM Software Emulation library is used and the above steps for signing and installing the FM do not apply. The emulated HSM does however require basic initialization using the same steps as a real HSM, as described in the *PTK-C Administration Guide*.

```
init slot0:
# init the slot
ctconf -n0
# init the slot's user pin
ctkmu p -s0
# list slots
ctkmu l
```

RSAENC:

Description: This sample demonstrates how custom functionality can be implemented with use of the RSA_Enc command.

This command combines several PKCS#11 commands such as C_Initialize, C_OpenSession, C_FindObjectsInit, C_FindObjects, C_EncryptInit, C_Encrypt, C_CloseSession in one single call.

Note that for running this sample TEST_RSA_KEY RSA key must be created. This may be achieved with the following command.
ctkmu c -nTEST_RSA_KEY -trsa -z1024 -aEDSUPT

XORSign:

Description: This sample demonstrates the addition of a new signing mechanism by patching existing PKCS#11 digest functions.

The new mechanism is bitwise exclusive OR with name CKM_XOR.

Execute the test app with –g to generate the required key:
xortest -g

restrict:

Description: This sample demonstrates restricting access to slot 0 by patching existing PKCS#11 C_OpenSession function.

safedebug:

Description: This sample demonstrates how to use smfs storage to determine if your fm should abort it's startup.

cipherobj:

Description: This sample demonstrates how to access primitive cryptographic services by using the Cipher Object system. Note that for running this sample TEST_DES3_KEY DES3 key must be created. This may be achieved with the following command.
ctkmu c -nTEST_DES3_KEY -tdes3 –aEDSVT

The key must be marked as sensitive (-T) as this example also demonstrates the CT_SetPrivilegeLevel(PRIVILEGE_OVERRIDE) api.

smfs:

Description: This sample demonstrates how to access the Secure Memory File System
.

The following two samples do not make FMs; instead they demonstrate how to access HSMs from java code.

javahsmreset:

Description: This is a JAVA version of the function hsmreset which is functionally identical to the 'C' hsmreset function. This function demonstrates how to interface to the Java MD API (JHSM).Only the Java source code is provided. It is recommended that the THREADS_FLAG environment variable is set to native for Unix/Linux platforms.

javahsmstate:

Description: This is a JAVA version of the function hsmstate which is functionally identical to the 'C' hsmstate function. This function demonstrates how to interface to the Java MD API (JHSM). Only the Java source code is provided. It is recommended that the THREADS_FLAG environment variable is set to native for Unix/Linux platforms.

Emulation builds and test steps

```
C:> make EMUL=1
C:> cd fm/obj-linux-i386e
C:> sampleN
```

Adapters builds and test steps:

```
C:> make
C:> cd obj-armfm
C:> mkfm –k devel_key -f sampleN –o sampleN.fm
C:> ctconf –j sampleN.fm
C:> sampleN
```

Chapter 7
Utilities Reference

CTCERT

The CTCERT utility referred to in this manual is provided as a part of the ProtectToolkit C package. Refer to the ProtectToolkit C Administration Manual for details.

CTCONF

The CTCONF utility referred to in this manual is provided as a part of the ProtectToolkit C package. Refer to the ProtectToolkit C Administration Manual for details.

CTFM

The CTFM utility referred to in this manual is provided as a part of the ProtectToolkit C package.
Refer to the ProtectToolkit C Administration Manual for details.

MKFM

Synopsis

mkfm **-f**<*filename*> **-k**<*key*> **-o**<*filename*> [**-3**]

Description

The **mkfm** utility is used to time stamp, hash and sign a FM binary image

Options

The following options are supported:

–**f***filename*	**--input-file=<***filename*>	This specifies the relative or full path to the FM binary image.
–**k***key*	**--private-key=<***key*>	This is the name of the private key, which is going to be used to sign the FM image. The format of the key is “TokenName(PIN)/KeyName”, or “TokenName/KeyName”. The private keys stored in admin token cannot be used with this utility.
–**o***filename*	**--output-file=<***filename*>	This specifies the relative or full path to the downloadable FM image.
-3		Specify this option if you want to sign a new FM with an existing certificate that was created for use with Protect Processor Orange 3 (PTK 3).

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 8
Cipher Object

Introduction

The PKCS #11 API provides a standard way of accessing and managing keys, and performing cryptographic operations. However, in order to provide a system independent layer, a considerable amount of overhead is introduced.

PTK provides an internal API, which bypasses the PKCS #11 subsystem to provide high performance cryptographic functionality. This chapter describes the functions of this API.

The Cipher Object Access API

Performing a cryptographic operation requires that you obtain a pointer to an instance of a CipherObject or a HashObject. A CipherObject may be used to encrypt, decrypt, sign (or MAC), or verify data. A hash object is used to perform a digest operation.
There are two functions for obtaining an instance of these objects. This section provides the details for these functions.

FmCreateCipherObject

Synopsis

#include “fmciphobj.h”

CipherObj * FmCreateCipherObject(FMCO_CipherObjIndex index);

Description

This function constructs an initializes a CipherObject of the specified type.

Parameters

index: The type of cipher object requested. It can have the following values (defined in fmciphobj.h):

This list is correct at time of writing – however the actual number of objects supported depends on the version of HSM firmware that the FM runs on.

- FMCO_IDX_AES: Implementation of the AES (Rijndael) algorithm;
- FMCO_IDX_CAST: Implementation of the CAST algorithm;
- FMCO_IDX_IDEA: Implementation of the IDEA algorithm;
- FMCO_IDX_RC2: Implementation of the RC2 algorithm;
- FMCO_IDX_RC4: Implementation of the RC4 algorithm;
- FMCO_IDX_DES: Implementation of the single DES algorithm;
- FMCO_IDX_TRIPLEDES: Implementation of the triple DES (with either double or triple length keys) algorithm;
- FMCO_IDX_DSA: Implementation of the DSA algorithm (only for signing and verification)
- FMCO_IDX_ECDSA: Implementation of the ECDSA algorithm (only for signing and verification)
- FMCO_IDX_HMACMD2: Implementation of the HMAC construct using MD2 hash algorithm (only for signing and verification)

- FMCO_IDX_HMACMD5: Implementation of the HMAC construct using MD5 hash algorithm (only for signing and verification)
- FMCO_IDX_HMACSHA1: Implementation of the HMAC construct using SHA-1 hash algorithm (only for signing and verification)
- FMCO_IDX_HMACRMD128: Implementation of the HMAC construct using RIPEMD-128 hash algorithm (only for signing and verification)
- FMCO_IDX_HMACRMD160: Implementation of the HMAC construct using RIPEMD-160 hash algorithm (only for signing and verification)
- FMCO_IDX_RSA: Implementation of the RSA algorithm (only single part operations supported).

Return Value

The address of the cipher object is returned. If the system doesn't have enough memory to complete the operation, NULL is returned.

Comments

The returned Cipher Object should be freed by calling its Free() function (See the function description).
The cipher objects may have some of the function's addresses set to NULL to indicate that they are not implemented. It is recommended that the function address be checked before using these functions. Please consult the *Cipher Object Functions* chapter for details.

NOTE – it is the Operating System firmware that provides the CipherObject – not the FM SDK. As new versions of OS firmware are developed and released more CiphrObjects may be added to the list of supported algorithms.
Therefore a firmware upgrade may be required to obtain a particular Cipher Algorith.

Cipher Object Functions

The Cipher Object is a wrapper that provides a common interface for all supported cryptographic algorithms. It is implemented as a structure, which contains the addresses of functions. The structure also contains a data pointer that keeps the internal state of the instance. The contents of the data field are private, and should not be accessed or modified externally.
In this section; the functions in the cipher object are specified.

New

Synopsis

#include “fmciphobj.h”

struct CipherObj * (*New)(struct CipherObj * ctx);

Description

Creates a new instance of the same type of the cipher object. This function can be used as a generic means to clone a cipher object if two cipher operations must be carried out in parallel.

Parameters

ctx: The address of an existing cipher object. This parameter must not be NULL.

Return Value

Address of a new instance of the cipher object. If the system does not have enough memory to complete the operation, NULL is returned.

Comments

The returned CipherObject should be freed by calling its Free() function (see the function description).

Free

Synopsis

#include “fmciphobj.h”

int (*Free)(struct CipherObj * ctx);

Description

This function must be used for all cipher objects that are no longer required to free all of its resources. The cipher object must not be used after this function returns.

Parameters

ctx	Address of the cipher object to be freed.

Return Value

None.

GetInfo

Synopsis

#include “fmciphobj.h”

int (*GetInfo)(struct CipherObj * ctx, struct CipherInfo * info);

Description

This function can be called to obtain the values of the algorithm-dependent parameters of a cipher object.

Parameters

ctx	The address of a cipher object.
Info	The address of the info structure that will contain the algorithm information when the function returns.

Return Value

0	Operation completed successfully. Otherwise: there was an error – this should not happen.

Comments

The info structure is defined as (from fmciphobj.h):

```
struct CipherInfo {
        char name[32];
        unsigned int minKeyLength;
        unsigned int maxKeyLength;
        unsigned int blockSize;
        unsigned int defSignatureSize;
        struct CipherObj * ciph;
};
```

The field meanings are:

- name: Name of the cipher algorithm. Zero terminated.
- minKeyLength: Minimum key length, in number of bytes
- maxKeyLength: Maximum key length, in number of bytes
- blockSize: Cipher block size, in number of bytes
- defSignatureSize: Default Signature size, in number of bytes
- ciph: The address of the cipher object (obsolete, and not filled in by most cipher objects).

Config (Obsolete)

Synopsis

```
#include "fmciphobj.h"
int (*Config)(struct CipherObj * ctx, const void * parameters, unsigned int length);
```

Description

This function can be used to restore the configuration of a cipher object. It is used in conjunction with the Status function.

Parameters

ctx	The address of the cipher object.
parameters	The address of the buffer that contains the information returned from the Status function.
length	The number of bytes in the parameters buffer.

Return Value

0	Operation was successful
	Otherwise: There was an error.

Comments

This function is now obsolete, and is not implemented by any of the cipher objects.

Status (Obsolete)

Synopsis

```
#include "fmciphobj.h"
int (*Status)(struct CipherObj * ctx, void * parameters, unsigned int length);
```

Description

This function can be used to take a snapshot of the current configuration of the cipher object, which can be used to restore it using the Config function.

Parameters

ctx	The address of the cipher object.
parameters	The address of the buffer that will receive the configuration information.
length	The number of bytes in the parameters buffer.

Return Value

0	Operation was successful
	Otherwise: There was an error.

Comments

This function is now obsolete, and is not implemented by any of the cipher objects.

EncInit

Synopsis

```
#include “fmciphobj.h”

int (*EncInit)(struct CipherObj * ctx,
               int mode,
               const void * key, unsigned int klength,
               const void * param, unsigned int plength);
```

Description

This function initializes the cipher object for an encrypt operation. It must be called prior to any EncryptUpdate, or EncryptFinal calls.

Parameters

ctx	The address of the cipher object instance.
mode	The encrypt mode. Different algorithms support different values for this parameter. Please consult the algorithms section for the possible values for a certain algorithm.
key	The address of a buffer containing the key value. The encoding of the key is algorithm dependent. However, for most block ciphers, this buffer contains the key value. Please consult the algorithms section for key encoding information.
klength	Number of bytes in the key buffer.
param	The address of the buffer containing various parameters for the encrypt operation. The contents and the encoding of this buffer are algorithm and mode dependent. However, for most block ciphers this buffer contains the IV when one of the CBC modes is used. Please consult the algorithms section for key encoding information.
plength	Number of bytes in the param buffer.

Return Value

0	The operation was successful.
	Otherwise: An error occurred.

EncryptUpdate

Synopsis

```
#include "fmciphobj.h"

int (*EncryptUpdate)(struct CipherObj * ctx,
        void * tgt, unsigned int tlength, unsigned int * plen,
        const void * src, unsigned int length);
```

Description

This function is used to encrypt some data. This function can be used in two ways: discovering the output buffer length, or performing encryption.
If the target buffer address, tgt, is NULL; then the variable pointed to by plen is updated to contain the length of the output that is required to perform the operation.
If the target buffer address is not NULL, then the currently buffered data and the input buffer contents are combined, and as many blocks as possible are encrypted. The result of the encrypted blocks are placed in the output buffer. If there are any remaining bytes, they are internally buffered. The number of bytes placed in the target buffer is also written to the variable pointed by plen.

Parameters

ctx	The address of a cipher object instance.
tgt	The address of the output buffer. It may be set to NULL for output buffer length estimation.
tlength	Total number of bytes available in the output buffer.
plen	Address of a variable that will receive the number of bytes placed in the target buffer. This variable must not be NULL.
src	Address of the buffer containing the input data.
length	Number of bytes in the src buffer.

Return Value

0	Operation completed successfully.
	Otherwise: There was an error.

EncryptFinal

Synopsis

```
#include "fmciphobj.h"

int (*EncryptFinal)(struct CipherObj * ctx,
        void * tgt, unsigned int tlength, unsigned int * plen);
```

Description

This function must be called to finish an encryption operation. It can be used for either discovering the target buffer length, or for actually performing the operation.
If the target buffer address, tgt, is NULL; then the variable pointed to by plen is updated to contain the length of the output that is required to perform the operation.
If the target buffer address is not NULL, then the currently buffered data is padded (if the mode permits it), and encrypted. The result is placed in the tgt buffer. The number of bytes placed in the target buffer is also written to the variable pointed by plen. If the current mode does not allow padding, and there is buffered data, then an error is returned.

Parameters

ctx	The address of a cipher object instance.
tgt	The address of the output buffer. It may be set to NULL for output buffer length estimation.
tlength	Total number of bytes available in the output buffer.
plen	Address of a variable that will receive the number of bytes placed in the target buffer. This variable must not be NULL.

Return Value

0	Operation completed successfully.
	Otherwise: There was an error.

Comments

This function finalizes the encryption context. EncInit() must be called again to encrypt more data, even when the key is the same.

DecInit

Synopsis

```
#include “fmciphobj.h”

int (*DecInit)(struct CipherObj * ctx,
               int mode,
               const void * key, unsigned int klength,
               const void * param, unsigned int plength);
```

Description

This function initializes the cipher object for a decrypt operation. It must be called prior to any DecryptUpdate, or DecryptFinal calls.

Parameters

ctx	The address of the cipher object instance.
mode	The decrypt mode. Different algorithms support different values for this parameter. Please consult the algorithms section for the possible values for a certain algorithm.
key	The address of a buffer containing the key value. The encoding of the key is algorithm dependent. However, for most block ciphers, this buffer contains the key value. Please consult the algorithms section for key encoding information.
klength	Number of bytes in the key buffer.
param	The address of the buffer containing various parameters for the decrypt operation. The contents and the encoding of this buffer are algorithm and mode dependent. However, for most block ciphers this buffer contains the IV when one of the CBC modes is used. Please consult the algorithms section for key encoding information.
plength	Number of bytes in the param buffer.

Return Value

0	The operation was successful.
	Otherwise: An error occurred.

DecryptUpdate

Synopsis

```
#include "fmciphobj.h"

int (*DecryptUpdate)(struct CipherObj * ctx,
        void * tgt, unsigned int tlength, unsigned int * plen,
        const void * src, unsigned int length);
```

Description

This function is used to decrypt some data. This function can be used in two ways: discovering the output buffer length, or performing decryption.
If the target buffer address, tgt, is NULL; then the variable pointed to by plen is updated to contain the length of the output that is required to perform the operation.
If the target buffer address is not NULL, then the currently buffered data and the input buffer contents are combined, and as many blocks as possible are decrypted. The result of the decrypted blocks are placed in the output buffer. If there are any remaining bytes, they are internally buffered. The number of bytes placed in the target buffer is also written to the variable pointed by plen.

Parameters

ctx	The address of a cipher object instance.
tgt	The address of the output buffer. It may be set to NULL for output buffer length estimation.
tlength	Total number of bytes available in the output buffer.
plen	Address of a variable that will receive the number of bytes placed in the target buffer. This variable must not be NULL.
src	Address of the buffer containing the input data.
length	Number of bytes in the src buffer.

Return Value

0	Operation completed successfully.
	Otherwise: There was an error.

DecryptFinal

Synopsis

```
#include "fmciphobj.h"

int (*DecryptFinal)(struct CipherObj * ctx,
        void * tgt, unsigned int tlength, unsigned int * plen);
```

Description

This function must be called to finish a decryption operation. It can be used for either discovering the target buffer length, or for actually performing the operation.
If the target buffer address, tgt, is NULL; then the variable pointed to by plen is updated to contain the length of the output that is required to perform the operation.
If the target buffer address is not NULL, and the mode is padded mode, the final block is decrypted and the padding bytes are removed before they are placed in the tgt buffer. If there is not exactly one block of data in the buffer, an error is returned.

Parameters

ctx	The address of a cipher object instance.
tgt	The address of the output buffer. It may be set to NULL for output buffer length estimation.
tlength	Total number of bytes available in the output buffer.
plen	Address of a variable that will receive the number of bytes placed in the target buffer. This variable must not be NULL.

Return Value

0	Operation completed successfully.
	Otherwise: There was an error.

Comments

This function finalizes the decryption context. DecInit() must be called again to decrypt more data, even when the key is the same.

SignInit

Synopsis

```
#include "fmciphobj.h"

int (*SignInit)(struct CipherObj * ctx,
                int mode,
                const void * key, unsigned int klength,
                const void * param, unsigned int plength);
```

Description

This function initializes the cipher object for a signature (MAC for block ciphers) operation. It must be called prior to any SignUpdate, SignFinal or SignRecover calls.

Parameters

ctx	The address of the cipher object instance.
mode	The signing mode. Different algorithms support different values for this parameter. Please consult the algorithms section for the possible values for a certain algorithm.
key	The address of a buffer containing the key value. The encoding of the key is algorithm dependent. However, for most block ciphers, this buffer contains the key value. Please consult the algorithms section for key encoding information.
klength	Number of bytes in the key buffer.
param	The address of the buffer containing various parameters for the signature or MAC operation. The contents and the encoding of this buffer are algorithm and mode dependent. Please consult the algorithms section for key encoding information.
plength	Number of bytes in the param buffer.

Return Value

0	The operation was successful.
Otherwise	An error occurred.

SignUpdate

Synopsis

```
#include “fmciphobj.h”

int (*SignUpdate)(struct CipherObj * ctx,
          const void * src, unsigned int length);
```

Description

This function can be used to update a multi-part signing or MAC operation.

Parameters

ctx	The address of a cipher object instance.
src	Address of the buffer containing the input data.
length	Number of bytes in the src buffer.

Return Value

0	Operation completed successfully.
	Otherwise there was an error.

Comments

Usually, only block cipher algorithms implement this function.

SignFinal

Synopsis

```
#include “fmciphobj.h”

int (*SignFinal)(struct CipherObj * ctx,
          void * tgt, unsigned int tlength, unsigned int * plen);
```

Description

This function must be called to finish a signing or MAC operation. It can be used for either discovering the target buffer length, or for actually performing the operation.
If the target buffer address, tgt, is NULL; then the variable pointed to by plen is updated to contain the length of the output that is required to perform the operation.
If the target buffer address is not NULL, then the signing operation is completed, and the signature is placed in the output buffer.

Parameters

ctx	The address of a cipher object instance.
tgt	The address of the output buffer. It may be set to NULL for output buffer length estimation.
tlength	Total number of bytes available in the output buffer.
plen	Address of a variable that will receive the number of bytes placed in the target buffer. This variable must not be NULL.

Return Value

0	Operation completed successfully.
	Otherwise there was an error.

Comments

This function finalizes the signature context. SignInit() must be called again to sign more data, even when the key is the same.

SignRecover

Synopsis

```
#include "fmciphobj.h"

int (*SignRecover)(struct CipherObj * ctx,
            void * tgt, unsigned int tlength, unsigned int * plen,
            const void * src, unsigned int length);
```

Description

This function implements a single part signing or MAC operation. It can be used for either discovering the target buffer length, or for actually performing the operation.
If the target buffer address, tgt, is NULL; then the variable pointed to by plen is updated to contain the length of the output that is required to perform the operation.
If the target buffer address is not NULL, then the signing operation is performed, and the signature is placed in the output buffer.

Parameters

ctx	The address of a cipher object instance.
tgt	The address of the output buffer. It may be set to NULL for output buffer length estimation.
tlength	Total number of bytes available in the output buffer.
plen	Address of a variable that will receive the number of bytes placed in the target buffer. This variable must not be NULL.
src	Address of the buffer containing the input data.
length	Number of bytes in the src buffer.

Return Value

0	The operation was successful.
	Otherwise an error occurred.

Comments

Most block cipher algorithm cipher objects do not implement this function.

VerifyInit

Synopsis

```
#include "fmciphobj.h"

int (*VerifyInit)(struct CipherObj * ctx,
                  int mode,
                  const void * key, unsigned int klength,
                  const void * param, unsigned int plength);
```

Description

This function initializes the cipher object for a signature or MAC verification operation. It must be called prior to any VerifyUpdate, VerifyFinal, Verify or VerifyRecover calls.

Parameters

ctx	The address of the cipher object instance.
mode	The verification mode. Different algorithms support different values for this parameter. Please consult the algorithms section for the possible values for a certain algorithm.
key	The address of a buffer containing the key value. The encoding of the key is algorithm dependent. However, for most block ciphers, this buffer contains the key value. Please consult the algorithms section for key encoding information.
klength	Number of bytes in the key buffer.
param	The address of the buffer containing various parameters for the signature or MAC operation. The contents and the encoding of this buffer are algorithm and mode dependent. Please consult the algorithms section for key encoding information.
plength	Number of bytes in the param buffer.

Return Value

0	The operation was successful.
	Otherwise an error occurred.

VerifyUpdate

Synopsis

```
#include "fmciphobj.h"

int (*VerifyUpdate)(struct CipherObj * ctx,
          const void * src, unsigned int length);
```

Description

This function can be used to update a multi part signature or MAC verification operation.

Parameters

ctx	The address of a cipher object instance.
src	Address of the buffer containing the input data.
length	Number of bytes in the src buffer.

Return Value

0	Operation completed successfully.
	Otherwise there was an error.

Comments

Usually, only block cipher algorithms implement this function.

VerifyFinal

Synopsis

```
#include "fmciphobj.h"

int (*VerifyFinal)(struct CipherObj * ctx,
        const void * sig, unsigned int slength,
        void * tgt, unsigned int tlength, unsigned int * plen);
```

Description

This function must be called to finish a signature or MAC verification operation. It can be used for either discovering the target buffer length, or for actually performing the operation.
If the target buffer address, tgt, is NULL; then the variable pointed to by plen is updated to contain the length of the output that is required to perform the operation.
If the target buffer address is not NULL, then the verification operation is completed. In addition to the verification, the recovered signature is placed in the output buffer.

Parameters

ctx	The address of a cipher object instance.
sig	The address of the buffer containing the signature or the MAC.
slength	Number of bytes in the sig buffer.
tgt	The address of the output buffer. It may be set to NULL for output buffer length estimation.
tlength	Total number of bytes available in the output buffer.
plen	Address of a variable that will receive the number of bytes placed in the target buffer. This variable must not be NULL.

Return Value

0	The signature or MAC was correct.
	Otherwise there was an error, or the signature or MAC was incorrect.

Comments

This function finalizes the verification context. VerifyInit() must be called again to verify more data, even when the key is the same.

VerifyRecover

Synopsis

```
#include "fmciphobj.h"

int (*VerifyRecover)(struct CipherObj * ctx,
        const void * sig, unsigned int slength,
        void * tgt, unsigned int tlength, unsigned int * plen,
        const void * src, unsigned int length);
```

Description

This function implements a single part signature or MAC verification operation with recovery. It can be used for either discovering the target buffer length, or for actually performing the operation.
If the target buffer address, tgt, is NULL; then the variable pointed to by plen is updated to contain the length of the output that is required to perform the operation.
If the target buffer address is not NULL, then the verification operation is performed. In addition to the verification, the recovered signature is placed in the output buffer.

Parameters

ctx	The address of a cipher object instance.
sig	The address of the buffer containing the signature or the MAC.
slength	Number of bytes in the sig buffer.
tgt	The address of the output buffer. It may be set to NULL for output buffer length estimation.
tlength	Total number of bytes available in the output buffer.
plen	Address of a variable that will receive the number of bytes placed in the target buffer. This variable must not be NULL.
src	Address of the buffer containing the input data.
length	Number of bytes in the src buffer.

Return Value

0	The operation was successful.
	Otherwise an error occurred.

Comments

Most block cipher algorithm cipher objects do not implement this function.
This function finalizes the verification context. VerifyInit() must be called again to verify more data, even when the key is the same.

Verify

Synopsis

```
#include "fmciphobj.h"

int (*Verify)(struct CipherObj * ctx,
                    const void * sig, unsigned int slength,
                    const void * src, unsigned int length);
```

Description

This function implements a single part signature or MAC verification operation without recovery. The return value indicates whether the signature or MAC was correct.

Parameters

ctx	The address of a cipher object instance.
sig	The address of the buffer containing the signature or the MAC.
slength	Number of bytes in the sig buffer.
src	Address of the buffer containing the input data.
length	Number of bytes in the src buffer.

Return Value

0	The operation was successful.
	Otherwise an error occurred.

Comments

Most block cipher algorithm cipher objects do not implement this function.

This function finalizes the verification context. VerifyInit() must be called again to verify more data, even when the key is the same.

LoadParam

Synopsis

```
#include “fmciphobj.h”

int (*LoadParam)(struct CipherObj * ctx,
            const void * param, unsigned int length);
```

Description

This function may be used to load the saved parameters of a cipher object. This function is used in conjunction with the UnloadParam function.

Parameters

ctx	The address of a cipher object instance.
param	The address of a buffer containing the encoded parameters.
length	Number of bytes in the param buffer.

Return Value

0	The operation was successful.
	Otherwise an error occurred.

Comments

The LoadParam and UnloadParam functions are not very useful, as they do not encode the key value along with the operational parameters.

UnloadParam

Synopsis

```
#include “fmciphobj.h”

int (*UnloadParam)(struct CipherObj * ctx,
            void * param, unsigned int length, unsigned int *plen);
```

Description

This function may be used to save the operational parameters of a cipher object. This function is used in conjunction with the LoadParam function.

Parameters

ctx	The address of a cipher object instance.
param	The address of a buffer that will receive the encoded parameters. This parameter must not be NULL.
length	Number of bytes in the param buffer.
plen	Address of a variable that will receive the number of bytes placed in the param buffer. This variable must not be NULL.

Return Value

0	The operation was successful.
	Otherwise an error occurred.

Comments

The LoadParam and UnloadParam functions are not very useful, as they do not encode the key value along with the operational parameters.

EncodeState (Obsolete)

This function definition is left for historical reasons. None of the Cipher objects implement this.

DecodeState (Obsolete)

This function definition is left for historical reasons. None of the Cipher objects implement this.

Algorithm-Specific Cipher Information

NOTE: Other algorithm details and hash object information is to be added at a later date.

DES Cipher Object

Operations Supported	Encrypt, Decrypt, multi part MAC, and multi part verify.
Key encoding	Every byte contains 7 key bits, and 1 parity bit. The parity bit is the least significant bit in each byte. There is no additional encoding of the key data. The key must be 8 bytes long.
Encrypt/Decrypt Modes	The least significant nibble (four bits) is used to determine the operational mode. The following list defines the possible values:

SYM_MODE_ECB (0)	Electronic Code Book (ECB) mode. It may be combined with a padding mode (see below).
SYM_MODE_CBC (1)	Cipher Block Chaining (CBC) mode. It may be combined with a padding mode (see below).
SYM_MODE_CFB (2)	Cipher Feedback (64-bit) mode
SYM_MODE_BCF (3)	Byte Cipher Feedback (8-bit CFB) mode
SYM_MODE_OFB (4)	Output Feedback (64-bit) mode
SYM_MODE_BOF (5)	Byte Output Feedback (8-bit OFB) mode

The most significant nibble defines the padding mode used. The following list defines the possible values:

SYM_MODE_PADNONE (0x00)	No padding applied. Data must be a multiple of the block size (8 bytes).

	SYM_MODE_PADNULL (0x80)	0 to 7 bytes with value 0 is added to the data to extend it to be a multiple of block size.
	SYM_MODE_PADPKCS1 (0x90)	PKCS#1 padding is applied. This causes 1 to 8 bytes of padding to be added to the data.
MAC modes	For MAC generation and verification operation the following modes are available:	
	0	Standard DES CBC
	1	Standard DES CBC with configurable MAC length
	In both methods, NULL padding is applied to the data.	
Encrypt/Decrypt Parameters	In all modes, except SYM_MODE_ECB, the parameter buffer must contain the IV (8 bytes). In SYM_MODE_ECB, there are no parameters.	
MAC Parameters	When mode 1 is used, parameter buffer contains 4 bytes, which contain a little endian encoding of an integer. The integer value must contain a value from 1 to 8, indicating the number of bytes of the final IV that will be used as the MAC.	

Triple DES Cipher Object

Operations Supported	Encrypt, Decrypt, multi part MAC, and multi part verify.	
Key encoding	Every byte contains 7 key bits, and 1 parity bit. The parity bit is the least significant bit in each byte. There is no additional encoding of the key data. The key must be 16 or 24 bytes.	
Encrypt/Decrypt Modes	The least significant nibble (four bits) is used to determine the operational mode. The following list defines the possible values:	
	SYM_MODE_ECB (0)	Electronic Code Book (ECB) mode. It may be combined with a padding mode (see below).
	SYM_MODE_CBC (1)	Cipher Block Chaining (CBC) mode. It may be combined with a padding mode (see below).
	SYM_MODE_CFB (2)	Cipher Feedback (64-bit) mode
	SYM_MODE_BCF (3)	Byte Cipher Feedback (8-bit CFB) mode
	SYM_MODE_OFB (4)	Output Feedback (64-bit) mode
	SYM_MODE_BOF (5)	Byte Output Feedback (8-bit OFB) mode
	The most significant nibble defines the padding mode used. The following list defines the possible values:	
	SYM_MODE_PADNONE (0x00)	No padding applied. Data must be a multiple of the block size (8 bytes).
	SYM_MODE_PADNULL (0x80)	0 to 7 bytes with value 0 is added to the data to extend it to be a multiple of block size.
	SYM_MODE_PADPKCS1 (0x90)	PKCS#1 padding is applied. This causes 1 to 8 bytes of padding to be added to the data.

MAC modes	For MAC generation and verification the following modes are available: 0 — Standard triple DES CBC 1 — Standard triple DES CBC with configurable MAC length 2 — X9.19 triple DES CBC 3 — X9.19 triple DES CBC with configurable MAC length 4 — Retail CFB MAC. In all methods, NULL padding is applied to the data.
Encrypt/Decrypt Parameters	In all modes, except SYM_MODE_ECB, the parameter buffer must contain the IV (8 bytes). In SYM_MODE_ECB, there are no parameters.
MAC Parameters	When mode is either 1 or 3, parameter buffer contains at least 4 bytes, which is the little endian encoding of an integer. The integer value must contain a value from 1 to 8, indicating the number of bytes of the final IV that will be used as the MAC. This is optionally followed by 8 bytes containing the IV. For mode 4 (Retail MAC CFB), the parameter buffer must have 8 bytes containing the encrypted IV. For the remaining two modes, the parameter buffer is either empty, or has 8 bytes containing the IV.

RSA Cipher Object

Operations Supported	New, Free, GetInfo, EncInit, DecInit, SignInit, VerifyInit, EncryptUpdate, DecryptUpdate, SignRecover, VerifyRecover and Verify. Therefore to perform an encrypt call: EncInit + EncryptUpdate To decrypt call: DecInit + DecryptUpdate To generate a signature call: SignInit + SignRecover To verify a signature call either: VerifyInit + VerifyRecover if you want to see the recovered signature or: VerifyInit + Verify if you don't care.
Key encoding	The key format depends on whether the operation is expecting a public key or a private key. Private Keys are used by: DecInit, SignInit Public Keys are used by: EncInit and VerifyInit Public Keys are stored in a CtPubRsaKey structure Private Keys are stored in a CtPriRsaKey structure (see the RSA Key Structures section below).
Modes	Supported modes are described below.
Parameters	The parameters for each mode are specified below.

RSA Modes

X509 Mode

#define RSA_MODE_X509 0

X509 Mode is the RAW uncooked mode. No padding or any other transformations are applied by the Cipher Object.
There is no parameter for this mode.

PKCS Mode

#define RSA_MODE_PKCS 1

PKCS Mode pads the input data into a specified block format according to the methods described in PKCS #1. The actual block padding method depends on whether encryption or signing operations are being performed.
For Encryption and Decryption Block Type 2 is used.
For Signing Block Type 1 is used.
There is no parameter for this mode.

9796 Mode

#define RSA_MODE_9796 2

ISO 9796 is a signature method only. Encrypt and Decrypt are not supported.
There is no parameter for this mode.

OAEP Mode

#define RSA_MODE_OAEP 3

OAEP is an Encryption/Decryption method only. Signing and Verification operations are not supported.
The padding is performed using the OAEP block format defined in PKCS #1.
This mode requires a parameter which is a structure of type CK_RSA_PKCS_OAEP_PARAMS (see cryptoki.h).
Restrictions apply to the values of members of the parameter structure:

- *hashAlg* must be CKM_SHA_1
- *mgf* must be CKG_MGF1_SHA1
- *source* must be CKZ_DATA_SPECIFIED

Example:

```
unsigned char data [SZ_DATA];
RSA_PUBLIC_KEY pub;
CipherObj * pRsa;
CK_RSA_PKCS_OAEP_PARAMS param;
param.hashAlg = CKM_SHA_1;
param.mgf = CKG_MGF1_SHA1;
param.source = CKZ_DATA_SPECIFIED;
param.pSourceData = data;
param.sourceDataLen = SZ_DATA;
pRSA->EncInit(pRSA, RSA_MODE_OAEP, &pub, sizeof(pub),
              &param, sizeof(param));
```

Note: The data pointed at by *pSourceData* must remain intact while the object is in use.

KEY WRAP OAEP Mode

#define RSA_MODE_KW_OAEP 4

Key Wrap OAEP is an Encryption/Decryption method only. Signing and Verification operations are not supported.
The padding is performed using the OAEP block format defined in PKCS #1 version 2.0
This mode requires a parameter which is a structure of type CK_KEY_WRAP_SET_OAEP_PARAMS (see cryptoki.h).

RSA Key Structures

```
#define MAX_RSA_MOD_BYTES (4096/8)
#define MAX_RSA_PRIME_BYTES ((MAX_RSA_MOD_BYTES / 2) + 4)
typedef unsigned char byte;

typedef struct {
   byte bits[2]; /* not used */
   byte mod [MAX_RSA_MOD_BYTES];
   byte exp [MAX_RSA_MOD_BYTES];
}
RSA_PUBLIC_KEY;

struct CtPubRsaKey {
        int isPub;                /* TRUE */
        unsigned int modSz; /* in bytes */
        RSA_PUBLIC_KEY key;
};
typedef struct CtPubRsaKey CtPubRsaKey;

typedef struct {
   byte bits[2]; /* not used */
   byte mod [MAX_RSA_MOD_BYTES];
   byte pub [MAX_RSA_MOD_BYTES];
   byte pri [MAX_RSA_MOD_BYTES];
   byte p   [MAX_RSA_PRIME_BYTES];
   byte q   [MAX_RSA_PRIME_BYTES];
   byte e1  [MAX_RSA_PRIME_BYTES];
   byte e2  [MAX_RSA_PRIME_BYTES];
   byte u   [MAX_RSA_PRIME_BYTES];
}
RSA_PRIVATE_KEY_XCRT;

struct CtPriRsaKey {
        int isPub;                /* FALSE */
        int isXcrt;        /* TRUE */
        unsigned int modSz; /* significant size in bytes */
        RSA_PRIVATE_KEY_XCRT key;
};

typedef struct CtPriRsaKey CtPriRsaKey;
```

Note: All values stored Big Endian i.e. most significant byte in mod[0] and least significant byte in mod[MAX_RSA_MOD_BYTES-1].

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 9 Hash Object

Generic Hash object. Wraps hashing algorithms into a common interface.

FmCreateHashObject

Synopsis

#include “fmciphobj.h”

HashObj * FmCreateHashObject(FMCO_HashObjIndex index);

Description

Returns the address of a hash object for digest operations.

Parameters

index: The type of hash object requested. It can have the following values (defined in fmciphobj.h):

- FMCO_IDX_MD2: Implementation of the MD-2 algorithm;
- FMCO_IDX_MD5: Implementation of the MD-5 algorithm;
- FMCO_IDX_RMD128: Implementation of the RIPEMD-128 algorithm;
- FMCO_IDX_RMD160: Implementation of the RIPEMD-160 algorithm;
- FMCO_IDX_SHA1: Implementation of the SHA-1 algorithm;

The following algorithms are supported in firmware 2.01 and above.

- FMCO_IDX_SHA256: Implementation of the SHA256 algorithm;
- FMCO_IDX_SHA384: Implementation of the SHA256 algorithm;
- FMCO_IDX_SHA512: Implementation of the SHA256 algorithm;

Return Value

The address of the hash object is returned. If the system doesn’t have enough memory to complete operation, NULL is returned.

Comments

The returned hash object should be freed by calling its Free() function (See the function description).

NOTE – it is the Operating System firmware that provides the HashObject – not the FM SDK. As new versions of OS firmware are developed and released more HashObjects may be added to the list of supported algorithms.
Therefore a firmware upgrade may be required to obtain a particular Hash Algorith.

Example

```
{
      char buf[100];
      char hash[100];
      int lenOut;

      HashObj *o = FmCreateHashObject(FMCO_IDX_SHA1);
      If ( o == NULL )
            Return error;
```

```
		o->Init(o);
		o->Update(o, data, len);
		o->Final(o, hash, sizeof(hash), &lenOut);
		o->Free(o);
	}
```

Free

Synopsis

```
#include "fmciphobj.h"

int (*Free)(struct HashObj * ctx);
```

Description

HashObj destructor

The hash object Free function releases resources used by the object. The object itself will be freed.

Parameters

ctx	pointer to object to destroy

Return Value

see CiphObjStat in cipherr.h

Init

Synopsis

```
#include "fmciphobj.h"

int (*Init)(struct HashObj * ctx);
```

Description

Configures the object to perform a hash operation or resets the current Hash operation.

Parameters

ctx	IN/OUT object to modify

Return Value

See CiphObjStat in cipherr.h

Update

Synopsis

```
#include "fmciphobj.h"

int (*Update)(struct
        HashObj * ctx,
        const void * buf,
        unsigned int length
);
```

Description

Uses the object to perform a hash operation or to process more data with the algorithm.

The data passed in buf is passed through the hash algorithm

Parameters

ctx	IN/OUT object to modify
buf	IN message to hash
length	IN length of message

Return Value

See CiphObjStat in cipherr.h

Final

Synopsis

```
#include "fmciphobj.h"

int (*Final)(struct
            HashObj * ctx,
            unsigned char * hashVal,
            unsigned int length,
            unsigned int * plength
);
```

Description

Final uses the object to finish a hash operation.

If 'hashVal' is NULL no operation is performed but the length that would be output is returned in 'plength'.

Parameters

ctx	IN/OUT object to modify
hashVal	OUT where to place hash or NULL for length prediction
length	IN length of hashVal (only used if hashVal not NULL)
plength	OUT number of bytes (actually or potentially) returned in hashVal

Return Value

See CiphObjStat in cipherr.h

GetInfo

Synopsis

```
#include "fmciphobj.h"

int (*GetInfo)(struct HashObj * ctx, struct HashInfo * hinfo);
```

Description

HashObjGetInfo will return information about an initialized HashObj. No sensitive information is returned by this function.

Parameters

ctx	IN object to query
hinfo	OUT pointer to where to store the result (see the *HashInfo* description below)

HashInfo Structure

Allows application to determine characteristics of the digest algorithm.

```
struct HashInfo {
        char name[32];     /**< null terminated ascii string e.g. "SHA-1" */
        unsigned int blockLength; /**< optimal hash block size */
        unsigned int hashLength;   /**< size of hash value */
        struct HashObj * hobj; /**< version 1 */
};
typedef struct HashInfo HashInfo;
```

Return Value

See CiphObjStat in cipherr.h

LoadParam

Synopsis

```
#include "fmciphobj.h"

int (*LoadParam)(struct
        HashObj * ctx,
        const unsigned char * parameters,
        unsigned int paramlen
);
```

Description

HashObjLoadParam directly modifies a Hash Object state.

Loads the internal parameters of the hash object from a byte array. If the internal data contains integers, the input byte array should contain big endian values for these integers.

See the particular Hash Class implementation description for details on valid parameter types and their values.

See UnloadParam

Parameters

ctx	IN/OUT object to modify
parameters	IN hash class specific information
paramlen	IN length (in bytes) of parameters

Return Value

See CiphObjStat in cipherr.h

UnloadParam

Synopsis

```
#include "fmciphobj.h"

int (*UnloadParam)(struct
            HashObj * ctx,
            unsigned char * parameters,
            unsigned int paramlen,
            unsigned int * plen
);
```

Description

HashObjUnloadParam queries a Hash Object state and return certain information.

Writes the internal parameters of the hash object to a byte array. If the internal data contains integers, the output byte array will contain big endian values for these integers.

See the particular Hash Class implementation description for details on valid parameter types and their values.

See LoadParam

Parameters

ctx	IN object to query
parameters	OUT hash class specific information (depends on pType)
paramlen	IN length of parameters (in bytes)
plen	OUT where to store the number of bytes returned in parameters (may be * NULL)

Return Value

See CiphObjStat in cipherr.h

Chapter 10
Setting Privilege Level

Introduction

CT_SetPrivilege allows elevation of privilege level to circumvent built-in security mechanisms on PKCS#11 objects. Elevated privilege level allows override of sensitive attribute and key usage

Two possible settings are available as follows:

PRIVILEGE_NORMAL=0
PRIVILEGE_OVERRIDE=1

SetPrivilegeLevelSynopsis

Synopsis

```
void CK_ENTRY CT_SetPrivilegeLevel( int level );
```

Description

This function is a SafeNet extension to PKCS#11. It can be used to set the privilege level of the caller to the specified value, if the caller has access to the function.
The function is available in the software cryptoki library to support FM emulation
The function cannot be called from outside the HSM (only inside from an HSM).

Use the CT_SetPrivilegeLevel function to set elevated privilege for a short time during the processing of a message. When the privileged access is complete call the CT_SetPrivilegeLevel function to set the privilege back to normal.

In the environment of a FM, the privilege is automatically returned to normal when the current message is complete i.e. when the FM Dispatch function returns or the currently intercepted Cryptoki function returns.

PRIVILEGE_OVERRIDE mode allows the FM to read Sensitive attributes and perform Cryptographic Initialization calls that contradict the usage attributes i.e. you can call C_EncryptInit with an object that has CKA_ENCRYPT set to FALSE.

Parameters

Level	- desired privilege.

Chapter 11
SMFS Reference

Introduction

SMFS is a Secure Memory File System (as exported to FMs).
It allows FMs to store keys into tamper protected battery backed Static RAM (SRAM)
It has the following general specifications:

- Arbitrary depth directory structure supported.
- File names are any character other than '\0' or '/'.
- Path separator is '/' (the windows '\' is not allowed)
- Files are fixed size and initialized with zeros when created.
- Directories will expand in size as needed to fit more files.

Important Constants

- Max file name length is 16
- Max path length is 100
- Max number of open files is 32
- Max number of file search handles is 16

Error Codes

SMFS_ERR_ERROR	1	A general error has occurred
SMFS_ERR_NOT_INITED	2	The SMFS has not been initialized
SMFS_ERR_MEMORY	3	The SMFS has run out of memory
SMFS_ERR_NAME_TOO_LONG	4	The name given for a file is too long
SMFS_ERR_RESOURCES	5	The SMFS has run out of resources
SMFS_ERR_PARAMETER	6	An invalid parameter was passed to SMFS
SMFS_ERR_ACCESS	7	User does not have request access to file
SMFS_ERR_NOT_FOUND	8	Requested file was not found
SMFS_ERR_BUSY	9	Operation is being attempted on an open file
SMFS_ERR_EXIST	10	A file being created already exists
SMFS_ERR_FILE_TYPE	11	Operation being performed on wrong file type

File Attributes Structure (SmFsAttr)

Synopsis

```
SmFsAttr {
            unsigned int        Size;
            unsigned int        isDir;
               };
```

Description

This structure holds the file or directory attributes

Members

Size	- Current file size in bytes or directory size in entries
isDir	- Flag specifying if file is a directory

Function Descriptions

SmFsCreateDir

Synopsis

```
int SmFsCreateDir(const char * name,
                        unsigned int entries);
```

Description

Allocates SRAM memory and a directory entry for a directory.

Parameters

name	- pointer to the absolute path of the directory to create
entries	- maximum number of entries that may exist I this directory

Return Value

Returns 0 for success or an error condition.

SmFsCloseFile

Synopsis

```
int SmFsCloseFile( SMFS_HANDLE fh）;
```

Description

Close the file by removing it from the file descriptor table.

Parameters

fh	- file handle of file to close

Return Value

Returns 0 or an error condition.

SmFsCalcFree

Synopsis

```
unsigned int SmFsCalcFree( void );
```

Return Value

Returns amount of free memory (in bytes) in the file system.

SmFsCreateFile

Synopsis

```
int SmFsCreateFile(const char * name,
                   unsigned int len);
```

Description

Allocates SRAM memory and a directory entry for a file. Once a file has been created, its size can not be changed.

Parameters

name	-	pointer to the absolute path of file to create
len	-	size of file to create (in bytes)

Return Value

Returns 0 for success or an error condition.

SmFsDeleteFile

Synopsis

```
int SmFsDeleteFile(const char * name);
```

Description

Deletes a file from secure memory by removing the directory entry `and` zeroing out its data area.

Parameters

name	-	pointer to the absolute path of the file to delete

Return Value

Returns 0 or an error condition.

SmFsFindFile

Synopsis

```
int SmFsFindFile( int sh,
                  char * name,
                  unsigned int size
                      );
```

Description

Fetch name of next directory entry from file search context

Parameters

sh	-	search handle to continue
name	-	pointer to location to hold found file name matching pattern
pattern	-	length of name buffer

Return Value

Returns 0 or an error condition.

SmFsFindFileClose

Synopsis

```
int SmFsFindFileClose( int sh);
```

Description

Close a file search context.

Parameters

sh	-	search handle to close

Return Value

Returns 0 or an error condition.

SmFsFindFileInit

Synopsis

```
int SmFsFindFileInit( int *sh,
                      const char * path,
                      const char * pattern
                         );
```

Description

Creates a file iteration context.

Wild cards are:
? – match any character
* - match many charavters

Parameters

sh	-	pointer to location to hold search handle
path	-	pointer to the absolute path where to search for file
pattern	-	pointer to pattern of file name (including wild cards) to search for

Return Value

Returns 0 or an error condition.

SmFsGetFileAttr

Synopsis

```
int SmFsGetFileAttr( const char * name,
                     SmFsAttr * a);
```

Description

Get attributes of an open file. Returns an attributes structure for the unopen file ‘name’.

Parameters

name	- pointer to absolute path
a	- pointer to the returned attributes structure

Return Value

Returns 0 or an error condition.

SmFsGetOpenFileAttr

Synopsis

```
int SmFsGetOpenFileAttr( SMFS_HANDLE fh,
                         SmFSAttr * a);
```

Description

Returns an attributes strucrure for the open file ‘name’.

Parameters

fh	- file handle
a	- pointer to the returned attributes structure

Return Value

Returns 0 or an error condition.

SmFsOpenFile

Synopsis

```
int SmFsOpenFile( SMFS_HANDLE * fh,,
                  const char * name,);
```

Description

Finds the file and creates and entry for it in the file descriptor table. The table index returned in ‘fh’ and is used by other file functions.

Parameters

fh	- pointer to the file handle
name	- pointer to absolute path

Return Value

Returns 0 or an error condition.

SmFsReadFile

Synopsis

```
int SmFsReadFile( SMFS_HANDLE fh,
                          unsigned int offset,
                          char *buf,
                          unsigned int bc）;
```

Description

Reads data from file.

Parameters

fh	- open file handle
offset	- zero based starting point
buf	- pointer to the returned result
bc	- the number of bytes to read from file

Return Value

Returns 0 or an error condition.

SmFsRenameFile

Synopsis

```
int SmFsRenameFile( const char * oldName,
                    const char * newName
                   );
```

Description

Renames a file.

Parameters

oldName	- pointer to the absolute path of file to rename
newName	- pointer of new file name only (no path)

Return Value

Returns 0 or an error condition.

SmFsWriteFile

Synopsis

```
int SmFsWriteFile( SMFS_HANDLE fh,
                          unsigned int offset,
                          char *buf,
                          unsigned int bc）;
```

Description

Write data to file.

Parameters

fh	- open file handle

offset	- zero based starting point
buf	- data to be written
bc	- the number of bytes to write

Return Value

Returns 0 or an error condition.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 12 FMDEBUG Reference

Introduction

FMDEBUG provides debug functions to FM writers. Debug information is available via the hsmtrace utility on the host.

On Linux, these debug messages are also written to /var/log/messages

Historically, debug logging has been via a simulated serial port 0. This is maintained for backwards compatibility. In PTK 5 support was added for standard C printf to write to the hsmtrace log. This is the recommended method.

Note : These functions and macros are supported under the FM emulation build as well. In this case the printing is done to stdout instead of the serial port.

Function Descriptions

debug (macro)

Synopsis

```
debug( statements )
```

Description

This macro is used to conditionally include code in the DEBUG build of the FM or FM emulation.

By placing the statements inside the debug macro the statements will appear only in the DEBUG build and will not be present in the Release build.

Example

```
rv = funct();
debug( if ( rv ) dbg_print(“Error %x from func\r\n”, rv); )
if ( rv ) return rv;
```

In this example the error message will only be displayed if the code is compiled for DEBUG and the funct returns an error code.

printf/vprintf

In addition to FMDEBUG logging, FM SDK 5.0 introduces support for the C standard printf() and vprintf() functions. These functions can be called at any time, with or without the debug library, and accept all standard C99 formating specifiers.

In FMs, these functions do not print to stdout, but instead send log messages to the hsmtrace log. Since these are formatting messages for a log rather than stdout, there are two differences from the standard C implementations.

1. Each printf()/vprintf() call prefices its output with a log header that includes the FM's id.
2. Each call to printf()/vprintf() has a newline appended to its output.

Should an FM developer require raw character logging as existed in PPO toolkits, the FMDEBUG and Serial Port 0 logging APIs may be used

DBG_INIT

Synopsis

int dbg_init()

Description

Not required. Retained for backwards compatibility with PSG.

This macro is used to initialize the debug library and claim serial port 1 of the PSG. The port is also moded up for (115200, 8, none, 1) serial mode operations.

DBG

Synopsis

int dbg(buf, len)

Description.

This macro is used to send a non-terminated string to serial port 1 of the PSG.

On PSI-E and newer, this API writes to the HSM trace log.

On PSI-E2, printf is preferred over use of this API.

Parameters

buf	array of printable characters to output to the serial port
len	length of buffer to output

DBG_PRINT

Synopsis

include <fmdebug.h>

int dbg_print(char *format, ...);

Description

This function formats and dumps the given string to serial port 1 of the PSG.
Its use mirrors that of the C function printf.

On PSI-E and newer, this API writes to the HSM trace log.

On PSI-E2, printf is preffered over use of this API.

Parameters

format	format of the string to print. This argument is followed by the values to place inside the format string.

Return Value

Returns 0 or -1 for failure.

DBG_STR

Synopsis

include <fmdebug.h>

int dbg_str()

Description

This macro is used to output a null terminated string to serial port 1 of the PSG.

On PSI-E and newer, this API writes to the HSM trace log.

On PSI-E2, printf is preferred over use of this API.

Parameters

str	string to output to serial port

DUMP

Synopsis

include <fmdebug.h>

void dump(char *desc, unsigned char *data, short len);

Description

This function converts unprintable character values into hex values and sends them to serial port 1 of the PSG.

On PSI-E and newer, this API writes to the HSM trace log.

Parameters

desc	pointer to string that holds the description of the dumped buffer, this string is dumped immediately before the dumped buffer.
data	pointer to buffer to be dumped
len	the length of he buffer to be dumped (in bytes)

DBG_FINAL

Synopsis

include <fmdebug.h>

int dbg_final()

Description

Not required. Retained for backwards compatibility with PSG.

This macro is used to finalize the debug library and release serial port 1 of the PSG.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 13
Libraries Reference

Function Summary

Function	Page
MD_Initialize	74
MD_Finalize	75
MD_GetHsmCount	76
MD_GetHsmState	77
MD_ResetHsm	79
MD_SendReceive	80
MD_GetParameter	83
SVC_GetReplyBuffer	87
SVC_ConvertReqToReply	88
SVC_ResizeReplyBuffer	89
SVC_DiscardReplyBuffer	90
SVC_GetPid	91
SVC_GetOid	92
SVC_GetRequest	93
SVC_GetRequestLength	94
SVC_GetReply	95
SVC_GetReplyLength	96
SVC_GetUserReplyBufLen	97
SVC_SendReply	98
FMSW_RegisterDispatch	99
SERIAL_SendData	100
SERIAL_ReceiveData	101
SERIAL_WaitReply	102
SERIAL_FlushRX	103
SERIAL_GetNumPorts	104
SERIAL_InitPort	105
SERIAL_GetControlLines	106
SERIAL_SetControlLines	107
SERIAL_SetMode	108
SERIAL_Open	109
SERIAL_Close	110
OS_GetCprovFuncTable	111
THR_BeginTiming	112
THR_UpdateTiming	113
FM_GetCurrentPid	114
FM_GetCurrentOid	115
FM_SetCurrentPid	116
FM_SetCurrentOid	117
FM_SetAppUserData	118
FM_GetAppUserData	119
FM_SetSlotUserData	120
FM_GetSlotUserData	121
FM_SetTokenUserData	122
FM_GetTokenUserData	123
FM_SetTokenAppUserData	124
FM_GetTokenAppUserData	125
FM_SetSessionUserData	126
FM_GetSessionUserData	127

Host Functions

The FM SDK has a number of host libraries that must be linked into the host application in order to be able to communicate with an FM. This section provides information on these functions.

MD API

The following functions labelled by the MD_ prefix form the Message Dispatch (MD) API. The function prototypes are defined in the header file md.h. The libraries etpso (for local HSMs) and etnetclient (for remote HSMs) implement the PCI bus and NetServer driver respectively. The driver is accessible via the MD API.

MD_Initialize

This function is used to initialize the message dispatch library. Until this function is called, all other functions will return error code MDR_NOT_INITIALIZED.

The message dispatch library is designed to operate on a stable HSM system (either local or remote to the Host computer). During the initialization of the message dispatch library, the number of accessible HSMs is determined and HSM indices are allocated to accessible HSMs. These variables are utilized in other functions; therefore, if the HSM system should change the message dispatch library should be re-initialized.

Synopsis

#include <md.h>

```
MD_RV MD_Initialize(void)
```

Input Requirements

None

Input Parameters

None

Output Requirements

The function returns either MDR_OK or MDR_UNSUCCESSFUL.

MD_Finalize

This function is used to finalize the message dispatch library. After this function returns, only the MD_Initialize() function should be called. All other functions will return error code MDR_NOT_INITIALIZED.

Synopsis

#include <md.h>

```
void MD_Finalize(void)
```

Input Requirements

The message dispatch library has been initialized via the MD_Initialize() function.

Input Parameters

None

Output Requirements

None

MD_GetHsmCount

This function retrieves the number of accessible HSMs at the time the message dispatch library was initialized (i.e. the time the MD_Initialize () function was called).

Synopsis

#include <md.h>

```
MD_RV MD_GetHsmCount(uint32* pHsmCount)
```

Input Requirements

The message dispatch library has been initialized via the MD_Initialize () function.

Input Parameters

pHsmCount:	Pointer to the variable which will hold the number of visible HSMs when the function returns. The pointer must not be NULL.

Output Requirements

The HSM Count is returned in pHsmCount.
The function returns the following codes:

Function Code	**Qualification**
MDR_OK	
MDR_INVALID_PARAMETER	If pHsmCount was NULL.
MDR_NOT_INITIALIZE	If the message dispatch library was not previously initialized successfully.

MD_GetHsmState

This function retrieves the current state of the specified HSM.

Synopsis

#include <md.h>

```
MD_RV MD_GetHsmState(uint32      hsmIndex,
                     HsmState_t* pState,
                     uint32*     pErrorCode);
```

Input Requirements

The message dispatch library has been initialized via the MD_Initialize() function.

Input Parameters

hsmIndex:	Zero based index of the HSM to query. For remote HSMs, HSMs are numbered according to the order that the HSMs IP addresses were entered in the ET_HSM_NETCLIENT_SERVERLIST registry key. Refer to the HSM Access Provider Install Guide for further details. When MD_Initialize () is invoked the message dispatch library assigns an index to each available HSM.
pState:	Pointer to a variable to hold the HSM state. The pointer must not be NULL.
pErrorCode:	Pointer to a variable which will hold the HSM error code when the function returns. If the HSM is halted, this variable contains an error code indicating the conditions which caused the halt. The pointer may be NULL.

Output Requirements

pState:	When the function returns pState points to a variable containing one of the following values. These values are defined in hsmstate.h :

Label	Value	Meaning
S_WAIT_ON_TAMPER	1	The HSM is waiting for the tamper cause to be removed.
S_HALT	6	The HSM is halted due to a failure.
S_POST	7	The HSM is initializing, and performing POST (Power On Self Test).
S_TAMPER_RESPOND	8	The HSM is responding to tamper.
S_NORMAL_OPERATION	0x8000	The HSM is in one of the following three states: S_NONFIPS_MODE, S_WAIT_FOR_INIT, or S_FIPS_MODE.

pErrorCode:	When the function returns pErrorCode points to a variable containing one of the following values. These values are defined in inc_scfs.h :

Label	Value	Meaning
SCFS_BAD_SDRAM	0x00000001	The HSM has come out of reset but has crashed whilst checking the SDRAM.
SCFS_BAD_SRAM	0x00000004	Failure in Secure Memory.

Label	Value	Meaning
SCFS_BAD_SMFS	0x0000000A	Failure in the Secure Memory File System.
SCFS_BAD_CRYPT	0x00000021	Failure whilst testing cryptographic algorithms.
SCFS_BAD_RBG	0x00000023	Failure whilst testing the random number generator.
SCFS_UART_COM	0x00000026	Failure to detect the Real Time Clock.
SCFS_FLASH_FM	0x00000040	Failure in the Flash ROM – section containing the functionality module.
SCFS_FLASH_SA0	0x00000041	Failure in the Flash ROM – section containing the hardware info.
SCFS_TAMP_ACTIVE	0x00000083	The HSM has been tampered and the tamper is active

The function returns the following codes:

Function Code	Qualification
MDR_OK	
MDR_UNSUCCESSFUL	
MDR_NOT_INITIALIZE	If the message dispatch library was not previously initialized successfully.
MDR_INVALID_HSM_INDEX	If HSM index was not in the range of accessible HSMs.

MD_ResetHsm

This function is used to reset the specified HSM.

Synopsis

#include <md.h>

```
MD_RV MD_ResetHsm(uint32 hsmIndex);
```

.

Input Requirements

The message dispatch library has been initialized via the MD_Initialize () function.
The remote server may disable or limit the use of this function via the ET_HSM_NETSERVER_ALLOW_RESET environment variable. Refer to the HSM Access Provider Install Guide for further details. If this limitation has been set, then this function may only be called when the HSM stat is not S_NORMAL_OPERATION. Refer to the section describing the MD_GetHsmState function for further details.

Input Parameters

hsmIndex:	Zero based index of the HSM to query. For remote HSMs, the HSM indices are numbered according to the order that the HSMs' IP addresses were entered in the ET_HSM_NETCLIENT_SERVERLIST registry key. Refer to the HSM Access Provider Install Guide for further details. When MD_Initialize () is invoked the message dispatch library assigns an index to each available HSM.

Output Requirements

The HSM is reset.
The function returns the following codes:

Function Code	**Qualification**
MDR_OK	
MDR_UNSUCCESSFUL	
MDR_NOT_INITIALIZE	If the message dispatch library was not previously initialized successfully.
MDR_INVALID_HSM_INDEX	If HSM index was not in the range of accessible HSMs.

MD_SendReceive

This function is used to send a request and receive the response.

Synopsis

#include <md.h>

```
MD_RV MD_SendReceive(uint32       hsmIndex,
                     uint32       originatorId,
                     uint16       fmNumber,
                     MD_Buffer_t* pReq,
                     uint32       timeout,
                     MD_Buffer_t* pResp,
                     uint32*      pReceivedLen,
                     uint32*      pFmStatus);
```

Input Requirements

The message dispatch library has been initialized via the MD_Initialize () function.

Input Parameters

hsmIndex: Zero based index of the HSM to query. For remote HSMs, HSMs are numbered according to the order that the HSMs IP addresses were entered in the ET_HSM_NETCLIENT_SERVERLIST registry key. Refer to HSM Access Provider Install Guide for further details. When MD_Initialize () is invoked the message dispatch library assigns an index to each available HSM.

fmNumber: Identifies whether the request is intended for a Functionality Module(FM) or not. This value must be set to FM_NUMBER_CUSTOM_FM (#include csa8fm.h).

originatorId: Id of the request originator. This value is typically 0. The value should only be non-zero when the calling application is acting as a proxy.

pReq: Array of request buffers to send to the FM module. For user-defined functions, the structure and content of the array of buffers is user defined. Refer to

javahsmreset:javahsmstate: for an example of how to construct the response and request buffers for a user-defined function in Java.

Each buffer in the array is an MD_Buffer_t struct, which contains a pointer to the data and the number of bytes of data, as detailed below.

```
typedef struct
{
      uint8*      pData;
      unit32      length;
} MD_Buffer_t;
```

In the final MD_Buffer_t struct the pData field must contain a NULL pointer and the length field should be set to 0. This indicates the end of the array of buffers. This scheme allows arrays with variable number of buffers to be passed into the function.

The following diagram illustrates an array of buffers containing two buffers. The first buffer contains 6 bytes of data and the second buffer contains 4 bytes of data. The last array element contains an array with the pData field set to NULL and the length field set to 0 to indicate the end of the array.

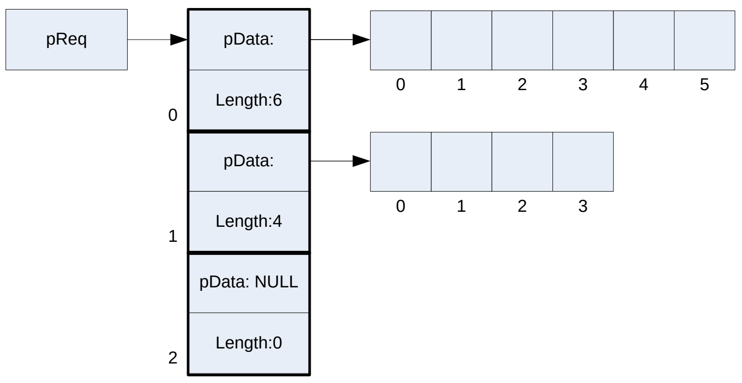

Figure 3 – An example of a request buffers data type for function MD_SendReceive

timeout:	The message timeout in ms. If set to 0, an internal default of 10 minutes is used.
pResp:	Response buffers. When the function returns, the response from the FM is contained in these buffers. Refer to the description of the pReq buffers above for details regarding how these buffers must be constructed. The memory for the pResp buffers must be allocated in the context of the application which calls the function. The pData field and length fields must be assigned appropriately to conform to the anticipated response packet. The buffers are filled in order until either the entire response is copied or the buffers overflow (this condition determined by pReceivedLen described below). The value of this parameter can be NULL if the FM function will not return a response.
pReceivedLen:	Address of variable to hold the total number of bytes placed in the response buffers. The memory for this variable must be allocated in the context of the application which calls the function. The value of this parameter can be NULL if the FM function will not return a response.
pFmStatus:	Address of variable to hold the status/return code of the Functionality Module which processed the request. The meaning of the value is defined by the FM. The value of this parameter can be NULL.

Output Requirements

The request is sent to the appropriate FM module. Where applicable the response is returned in the response buffers.

The function returns the following codes:

Function Code	Qualification
MDR_OK	
MDR_UNSUCCESSFUL	
MDR_INVALID_PARAMETER	If the pointer supplied for pReq is NULL, if the request requires a response and the pointer supplied for pResp is NULL or if pReserved is not zero.
MDR_NOT_INITIALIZE	If the message dispatch library was not previously initialized successfully.
MDR_INVALID_HSM_INDEX	If HSM index was not in the range of accessible HSMs.
MDR_INSUFFICIENT_RESOURCE	If there is insufficient memory on either the host or adapter
MDR_OPERATION_CANCELLED	The operation was cancelled in the HSM. This code will not be returned.

Function Code	**Qualification**
MDR_INTERNAL_ERROR	The HSM has detected an internal error. This code will be returned if there is a fault in the firmware or device driver.
MDR_ADAPTER_RESET	The HSM was reset during the operation. This could be possibly due to the MD_ResetHsm command being issued during the operation.
MDR_FM_NOT_AVAILABLE	An invalid FM number was used.

MD_GetParameter

This function obtains the value of a system parameter.

Synopsis

#include <md.h>

```
MD_RV MD_GetParameter(MD_Parameter_t  parameter,
                      void*           pValue,
                      unsigned int    valueLen);
```

Input Requirements

The message dispatch library has been initialized via the MD_Initialize () function.

Input Parameters

parameter: The parameter to query. The following parameter may be queried. The parameter is defined in md.h

Parameter	Value	Meaning
MDP_MAX_BUFFER_LENGTH	1	The recommended maximum buffer size, in number of bytes, for messages that can be sent using the MD library. Whilst messages larger than this buffer size may be accepted by the library, it is not recommended to do so. Different types of HSM access providers have different values for this parameter. When this parameter returns 0 via pValue this means that there is no limit to the amount of data that can be sent using this library.

pValue: The address of the buffer to hold the parameter value. The memory for the buffer must be allocated in the context of the application which calls the function. The size of the buffer is determined by the parameter that is being obtained. The following table specifies the buffer requirements for the parameter.

Parameter	Size	Type
MDP_MAX_BUFFER_LENGTH	4 bytes	unsigned integer

valueLen: The length of the buffer, pValue, in bytes. If the buffer length is not correct, MDR_INVALID_PARAMETER is returned.

Output Requirements

The function returns the following codes:

Function Code	Qualification
MDR_OK	
MDR_INVALID_PARAMETER	If the pointer supplied for pReq is NULL, if the request requires a response and the pointer supplied for pResp is NULL or if pReserved is not zero.

FM Host Legacy Functions API

Host legacy functions are no longer supported. If you have an existing host application that uses host legacy functions, it will continue to work. It is recommended, however, that you do not use any host legacy functions in new or migrated host applications.

The functions listed in the following table have been superseded as shown.

Legacy Function	**Superceded by**
FM_Initialize	MD_Initialize
FM_Finalize	MD_Finalize
FM_DispatchRequest	MD_SendReceive

HSM Functions

The FMS SDK has a number of libraries that are required to use the functionality provided to the FM. This section provides information on these functions and the libraries that provide the function set.

Apart from the functions described in this section, the full set of PKCS#11 functions are also available to the FM. The PKCS#11 functions are described in the Cprov Programmer Manual, and the PKCS#11 standard. The library libfmcprov.a provides the PKCS#11 functions.

Summary

Subset of ISO C99 standard library

The FM SDK supports a subset of the ISO C 99 standard library as defined by ISO/IEC 9899:1999. In general, floating point math, multibyte character, localization and I/O APIs are not supported. printf and vprintf are exceptions and are redirceted to the logging channel.

In addition to the standard library, C99 language features not present in ANSI C (C89/90) can be used.

The following functions are provided by libfmcrt.a :

assert.h
assert

ctype.h
isalnum, isalpha, isblank, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, tolower, toupper

stdio.h
printf, sprintf, sscanf, vprintf, vsprintf, snprintf, vsnprintf, vsscanf

stdarg.h
va_arg(), va_start(), va_end(), va_copy()

stdlib.h
abs, atoi, atol, atoll, bsearch, calloc, div, free, labs, llabs, ldiv, lldiv, malloc, qsort, rand, realloc, srand, strtol, strtoll, strtoul, strtoull

string.h
memchr, memcmp, memcpy, memmove, memset, strcat, strchr, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, strerror, strtok

time.h
asctime, clock, ctime, gmtime, localtime, mktime, strftime, time
difftime

Extensions to the Standard C API

The FMS SDK also supports the following common, but non-ISO library functions in their GNU form.

NOTE: These may not be available in emulation mode. See “Supported C APIs” on page 8.

ctype.h
isascii, toascii

string.h
strdup, strsep

Unsupported Standard C APIs

The following standar headers and their contained APIs are not supported by the FM SDK:

complex.h, fenv.h, float.h, locale.h, math.h, signal.h, tgmath.h, wchar.h, and wctype.h

HIFACE reply management functions

The following functions are provided by libfmcsa8k.a :

SVC_GetReplyBuffer
SVC_ConvertReqToReply
SVC_SendReply
SVC_ResizeReplyBuffer
SVC_DiscardReplyBuffer
SVC_GetUserReplyBufLen
SVC_GetPid
SVC_GetOid
SVC_GetRequest
SVC_GetRequestLength
SVC_GetReply
SVC_GetReplyLength

Functionality module dispatch switcher function

The following functions are provided by libfmcsa8k.a :

FMSW_RegisterDispatch

Cprov function patching helper function

The following functions are provided by libfmcsa8k.a :

OS_GetCprovFuncTable

Serial communication functions

The following functions are provided by libfmserial.a :

SERIAL_GetNumPorts;
SERIAL_Open;
SERIAL_Close;

SERIAL_InitPort;
SERIAL_SendData;
SERIAL_WaitReply;
SERIAL_ReceiveData;
SERIAL_FlushRX;
SERIAL_GetControlLines;
SERIAL_SetControlLines;
SERIAL_SetMode;

The above functions are detailed in the following pages.

HIFACE reply management functions

This section contains the reply buffer management functions of the service module.

SVC_GetReplyBuffer

This function is used to allocate a reply buffer of the specified length, and associate it with the token. The contents of the allocated reply buffer will be sent back to the host when SVC_SendReply() function is called.

Synopsis

```
#include <csa8hiface.h>    ditto for all SVC functions
void *SVC_GetReplyBuffer( HI_MsgHandle token,
                          uint32 replyLength);
```

Input Parameters

token:	The token identifying the request.
replyLength:	The Length of the reply buffer requested by the caller.

Output Requirements

If the reply buffer is allocated successfully, a pointer to the allocated reply buffer is returned. Otherwise, NULL is returned.

SVC_ConvertReqToReply

This function is used to treat the request buffer as the reply buffer for in-place processing of request data. The returned address of the reply buffer is not necessarily equal to the request buffer address. However, the contents of the reply buffer will always be the same as the contents of the request buffer.

Synopsis

void *SVC_ConvertReqToReply(HI_MsgHandle token);

Input Parameters

token: The token identifying the request.

Output Requirements

If a Reply Buffer is already allocated for the specified token, NULL is returned. Otherwise a pointer to the reply buffer is returned. The reply buffer will contain the data in the request buffer.

SVC_ResizeReplyBuffer

This function is used to resize the reply buffer associated with the specified token. The returned address need not be equal to the previous address of the reply buffer. However, the contents of the matching parts of the old and new reply buffers will always be the same.

Synopsis

```
void *SVC_ResizeReplyBuffer(HI_MsgHandle token,
                            uint32 replyLength);
```

Input Parameters

token:	The token identifying the request.
replyLength:	The new length of the reply buffer requested by the Destination Module.

Output Requirements

If the buffer is resized successfully, a pointer to the reply buffer is returned. Otherwise NULL is returned. The old reply buffer is not freed in this case.

SVC_DiscardReplyBuffer

This function is used to discard the current reply buffer. It is usually called when a processing error is detected after the reply has been allocated.

Synopsis

```
void SVC_DiscardReplyBuffer( HI_MsgHandle token );
```

Input Parameters

token: The token identifying the request.

Output Requirements

None.

SVC_GetPid

This function retrieves the process identifier (Pid) recorded in the request. The Pid is the Process Id of the host application that originated the request.

Synopsis

```
uint32 SVC_GetPid( HI_MsgHandle token );
```

Input Parameters

token: The token identifying the request.

Output Requirements

The Process identifier recorded in the request is returned.

SVC_GetOid

This function retrieves the originator identifier (Oid) recorded in the request. The Oid is set by the host application using the FM_Dispatch() function. The value is passed in from the host application.

Synopsis

uint32 SVC_GetOid(HI_MsgHandle token);

Input Parameters

token: The token identifying the request.

Output Requirements

The originator identifier recorded in the request is returned.

SVC_GetRequest

This function retrieves the address of request data.

Synopsis

```
void *SVC_GetRequest( HI_MsgHandle token );
```

Input Parameters

token: The token identifying the request.

Output Requirements

The request buffer address is returned.

SVC_GetRequestLength

This function retrieves the length of request data in number of bytes.

Synopsis

uint32 SVC_GetRequestLength(HI_MsgHandle token);

Input Parameters

token: The token identifying the request.

Output Requirements

The number of bytes in the request buffer is returned.

SVC_GetReply

This function retrieves the address of current reply buffer.

Synopsis

void *SVC_GetReply(HI_MsgHandle token);

Input Parameters

token: The token identifying the request.

Output Requirements

If there is a reply buffer associated with the token, the reply buffer address is returned. Otherwise, NULL is returned.

SVC_GetReplyLength

This function retrieves the length of reply data in number of bytes.

Synopsis

```
uint32 SVC_GetReplyLength( HI_MsgHandle token );
```

Input Parameters

token: The token identifying the request.

Output Requirements

If there is a reply buffer associated with the token, the number of bytes in the reply buffer is returned. Otherwise, 0 is returned.

SVC_GetUserReplyBufLen

This function retrieves the length of reply buffer the host application has. If the current reply length is larger than the value returned by this function, part of the reply will be discarded on the way back.

Synopsis

uint32 SVC_GetUserReplyBufLen(HI_MsgHandle token);

Input Parameters

token: The token identifying the request.

Output Requirements

The number of bytes available to place the reply data in the host system is returned.

SVC_SendReply

This function returns the reply to the host. If there is a reply buffer associated with the token, the data recorded in reply buffer is also returned.

Synopsis

```
void SVC_SendReply( HI_MsgHandle token,
                    uint32 applicationStatus);
```

Input Parameters

token:	The token identifying the request.
applicationStatus:	A status code for the execution of the request, which will be reported to the host application. The valus of this parameter does not affect the reply delivery in any way.

Output Requirements

None.

Functionality module dispatch switcher function

This section contains the firmware message dispatch management functions:

FMSW_RegisterDispatch

This function registers a FM API dispatch handler routine to the system. When a request is sent to the FM_NUMBER_CUSTOM_FM, the registered function is called.

The type FMSW_DispatchFn_t is a pointer to a function with the following declaration:

```
void DispatchHandler(uint32 token, void *reqBuffer, uint32 reqLength);
```

The token is an opaque handle value identifying the request. The same token must be passed to SVC_Xxx() functions.

The pair (reqBuffer, reqLength) defines the concatenated data that has been received on the request. See FM_Dispatch() function for the details of request dispatching.

This function is used when an FM exports a custom API. It is usually called from the startup() function.

Synopsis

```
#include <fmsw.h>
FMSW_STATUS FMSW_RegisterDispatch(
                    FMSW_FmNumber_t fmNumber,
                    FMSW_DispatchFn_t dispatch);
```

Input Parameters

fmNumber: The FM identification number
dispatch: pointer on API handler function

Output Requirements

Return Value:

- FMSW_OK - The function was registered successfully.
- FMSW_BAD_POINTER - The function pointer is invalid
- FMSW_INSUFFICIENT_RESOURCES - Not enough memory to complete operation
- FMSW_BAD_FM_NUMBER - The FM number is incorrect.
- FMSW_ALREADY_REGISTERED - A dispatch function was already registered.

Serial communcation functions

This section contains functions for using the serial ports on the HSM. Note that in emulation mode, the serial ports on the host system are used.

If you specify serial port 0, the output is redirected to the hsmtrace log.

SERIAL_SendData

Synopsis

```
#include <serial.h>          applies to all SERIAL_* functions
int SERIAL_SendData(int port,
                    unsigned char *buf,
                    int bufLen,
                    long timeout);
```

Description

SERIAL_SendData() function is used to send a character array over a serial port.

Parameters

port:	serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.
buf:	pointer to an array of bytes to be sent
bufLen:	length of the buffer, in bytes
timeout:	#milliseconds to wait for a character to be sent. A timeout of -1 will use the default timeout. NOTE: The timeout value refers to the total time taken to send the data. For example, a 2 millisecond timeout for sending 10 characters in 9600 baud setting will always fail – the timeout must be at least 10 milliseconds.

Return Code

0:	The characters were sent successfully.
-1:	There was an error.

SERIAL_ReceiveData

The SERIAL_ReceiveData() function is used to receive an arbitrary length of characters from the serial port.

Synopsis

```
int SERIAL_ReceiveData(int port,
                       unsigned char *buf,
                       int *len,
                       int bufLen,
                       long timeout);
```

Parameters

port:	serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.
buf:	pointer to an array of bytes, which will hold the received data.
len:	pointer to an integer which will hold the actual number of characters received.
bufLen:	Both the maximum amount of data, in bytes, of the buffer,and the number of bytes requested from the serial port.
timeout:	#milliseconds to wait for a character to appear. A timeout of -1 will use the default timeout

Return Code:

0:	Requested number of bytes has been received.
-1:	Less than the requested number of bytes have been received.

SERIAL_WaitReply

The SERIAL_WaitReply() function waits for a character to appear on the serial port.

Synopsis

int SERIAL_WaitReply(int port);

Parameters

port:	serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.

Return Code

0:	There is a character at the serial port.
-1:	Timeout occurred, and no data appeared.

SERIAL_FlushRX

This function flushes the receive buffer of the specified serial port.

Synopsis

```
void SERIAL_FlushRX( int port );
```

Parameters

port: serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.

SERIAL_GetNumPorts

This function returns the number of serial ports available.

Synopsis

int SERIAL_GetNumPorts(void);

Parameters

None.

Return Value

The number of serial ports available.

SERIAL_InitPort

This function initializes the specified serial port to the parameters "9600 8N1" with no handshake.

Synopsis

int SERIAL_InitPort(int port);

Parameters

port: serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.

Return Code:

0:	The serial port was initialized successfully.
-1:	There was an error initializing the port.

SERIAL_GetControlLines

This function reads the current state of the control lines, and writes a bitmap into the address pointed to by 'val'. Only the input bits (CTS, DSR, DCD, RI) reflect the current status of control lines.

Synopsis

```
int SERIAL_GetControlLines( int port,
                            unsigned char *bitmap);
```

Parameters

port:	serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.
bitmap:	Pointer to a character, which will have the resulting bitmap

Return Code:

0:	The function succeeded
-1:	The function failed. The value in the bitmap is not valid

Comments:

```
#define MCL_DSR         0x01
#define MCL_DTR         0x02
#define MCL_RTS         0x04
#define MCL_CTS         0x08
#define MCL_DCD         0x10
#define MCL_RI          0x20
#define MCL_OP_SET      1
#define MCL_OP_CLEAR    2
```

SERIAL_SetControlLines

This function is used to modify the control lines (DTR/RTS).

Synopsis

```
int SERIAL_SetControlLines( int port,
                            unsigned char bitmap,
                            int op);
```

Parameters

port:	serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.
bitmap:	bitmap of control lines to be modified. Input control lines are silently ignored.
op:	One of MCL_OP_SET/MCL_OP_CLEAR to set/clear the control lines specified in the bitmap parameter

Return Code

0:	The function succeeded
-1:	The function failed

Comments

The same constants used in SERIAL_GetControlLines() function are also used in this function.

SERIAL_SetMode

Used to set the serial port communication parameters.

Synopsis

```
int SERIAL_SetMode( int port,
                    int baud,
                    int numBits,
                    SERIAL_Parity parity,
                    int numStop,
                    SERIAL_HSMode hs);
```

Parameters

port:	serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.
baud:	baud rate.
numBits:	Number of bits in a character. Should be 7 or 8
parity:	One of the following: SERIAL_PARITY_NONE SERIAL_PARITY_ODD SERIAL_PARITY_EVEN SERIAL_PARITY_ONE SERIAL_PARITY_ZERO
numStop:	Number of stop bits in a character. Should be 1 or 2
hs:	Handshake type. Should be one of the following: SERIAL_HS_NONE SERIAL_HS_RTSCTS SERIAL_HS_XON_XOFF Note: Serial flow control is not implemented in the current HSM firmware. This value should be set to SERIAL_HS_NONE.

Return Code

0:	Mode changed successfully
-1:	There was an error

SERIAL_Open

Gets a weak ownership of the port. Subsequent calls to this function with the same parameter will fail unless SERIAL_ClosePort() is called for the same port.

Synopsis

int SERIAL_Open(int port);

Parameters

port:	serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.

Return Code

0:	Port opened successfully
others:	An error prevented the serial port from opening

Comments

This function in no way guarantees safe sharing of the ports.
Any application can call SERIAL_ClosePort() to get the access, or can use SERIAL functions without opening the port first.

SERIAL_Close

This function is used to release ownership of the serial port.

Synopsis

void SERIAL_Close(int port);

Parameters

port:	serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.

Return Code

Comments

See SERIAL_OpenPort

Cprov function patching helper function

This section contains information about Cprov Function patching operations.

The function patching is performed using a structure named CprovFnTable_t (defined in header file cprovtbl.h). The structure contains the number of functions in the table – which can be used as a structure version, the addresses of the standard Cprov functions, and SafeNet extended functions.

The functions in the table are named the same as the actual functions. i.e. C_Initialize function pointer is named C_Initialize in the structure. The order and place of the function pointers in the structure are guaranteed to be preserved indefinitely, even if PKCS #11 functions are extended, or more proprietary functions are added to the firmware. This contract allows for binary compatibility of FMs in future releases of the HSM firmware.

OS_GetCprovFuncTable

This function is used to obtain the address of Cprov function table structure, used by the Cprov Filter component in the firmware. Changing the addresses of functions in the structure allows custom functions to be called when a Cprov function is requested from the host side. The Cprov functions called from the FM bypass the Cprov Filter, calling the functions in the firmware directly.

Synopsis

```
#include <cprovpch.h>
CprovFnTable_t *OS_GetCprovFuncTable(void);
```

Return Code

The address of the Cprov function table structure. It will never be NULL.

High Resolution Timer Functions

These functions can be used to measure time intervals with very high resolution. The accuracy of the timing is around 1 microsecond.

These functions both use the structure, THR_TIME. This structure contains two values: the seconds (named secs), and the nanoseconds (named ns). The nanoseconds is always less than 10^9 (which is equal to 1 second).

THR_BeginTiming

This function can be used to start a high-resolution timing operation. The timing resolution is 20ns, and the accuracy of the timer is about 1 microsecond.

Synopsis

```
#include <timing.h>          ditto for other timing functions
void THR_BeginTiming(THR_TIME *start);
```

Parameters

start:	Address of the THR_TIME structure, which will keep the information needed to measure the timing interval.

Return Code

THR_UpdateTiming

Synopsis

```
void THR_UpdateTiming(const THR_TIME *start,
                      THR_TIME*elapsed);
```

Description

This function is used to update the timing operation. Since the start structure is not modified, it can be used multiple times with the same set of parameters.

Parameters

start:	Address of the THR_TIME structure that was passed to the THR_BeginTiming() function. The contents of the structure will not be modified.
elapsed:	Address of the THR_TIME structure, which will contain the elapsed time since THR_BeginTiming() was called. The contents of this structure will be overwritten.

Return Code

Current Application ID functions

These functions can be used to obtain and manipulate the PID (process ID) and OID (Originator ID - currently unused) of the calling application.

Normally, SVC_GetPid() and SVC_GetOid() functions are used to obtain these values. However, in patched PKCS#11 functions, the necessary value of *token* is not available, therefore the provided functions must be used instead.

FM_GetCurrentPid

This function returns the PID recorded in the current request originated from the host side. if there is no active request (e.g. a call from Startup()function), FM_DEFAULT_PID is returned.

Synopsis

```
#include <fmappid.h>        ditto for other FM ID functions
unsigned long FM_GetCurrentPid(void);
```

Return Code

The PID of the application which originated the request.

FM_GetCurrentOid

This function returns the OID recorded in the current request originated from the host side. if there is no active request (e.g. a call from Startup()function), FM_DEFAULT_OID is returned.

Synopsis

unsigned long FM_GetCurrentOid(void);

Return Code

The OID of the application which originated the request.

FM_SetCurrentPid

This function overrides the PID recorded in the current request originated from the host side. If there is no active request the function does nothing.

Synopsis

```
unsigned long FM_SetCurrentPid(unsigned long pid);
```

Parameters

pid:	The new PID to be recorded in the request.

Return Code

FM_SetCurrentOid

This function overrides the OID recorded in the current request originated from the host side. If there is no active request the function does nothing.

Synopsis

```
unsigned long FM_SetCurrentOid(unsigned long oid);
```

Parameters

oid:	The new OID to be recorded in the request.

Return Code

PKCS#11 State Management Functions

The functions listed in this section allows the FM to ask the firmware to associate user data with certain firmware structures. The firmware guarantees the cleanup of the associated buffer when the structure in question is destroyed.

The freeing of the user data is performed by a callback to a user function. If the data is allocated using malloc(), and it contains no pointers to other allocated structures, the free function is typically the standard free() function.

FM_SetAppUserData

This function can be used to associate user data with the calling application. The data is associated with the PID of the calling application. The function specified in this call will be called to free the data when the last application using the library finalizes (e.g. when it calls C_Finalize()).

If the application already has associated user data, it will be freed (by calling the current free function) before the new data association is created.

Synopsis

```
#include <objstate.h>
CK_RV FM_SetAppUserData(FmNumber_t fmNo,
                        CK_VOID_PTR userData,
                        CK_VOID (*freeUserData)(CK_VOID_PTR));
```

Parameters

fmNo:	The fm number of the caller. It must be FM_NUMBER_CUSTOM_FM in this release of the software.
userData:	Address of the memory block that will be associated with the session handle. if it is NULL, the current associated buffer is freed.
freeUserData:	Address of a function that will be called to free the userData if the library decides that it should be freed. it must be non-NULL if userData is not NULL.

Return Code

CKR_OK	The operation was successful.
CKR_ARGUMENTS_BAD	freeUserData was NULL, when userData was not NULL; or fmNo was not FM_NUMBER_CUSTOM_FM.
CKR_CRYPTOKI_NOT_INITIALIZED	Cryptoki is not yet initialized.

FM_GetAppUserData

This function is used to obtain the userData associated with the current application. If there are no associated buffers, NULL is returned in ppUserData.

Synopsis

```
#include <objstate.h>
CK_RV FM_SetAppUserData(FmNumber_t fmNo,
                        CK_VOID_PTR_PTR ppuserData);
```

Parameters

fmNo:	The fm number of the caller. It must be FM_NUMBER_CUSTOM_FM in this release of the software.
ppuserData:	Address of a variable (of type CK_Void_PTR) which will contain the address of the user data if this function returns CKR_OK. It must be non-NULL.

Return Code

CKR_OK	The operation was successful. The associated user data is placed in the variable specified by ppUserData.
CKR_ARGUMENTS_BAD	ppUserData was NULL, or fmNo was not FM_NUMBER_CUSTOM_FM.
CKR_CRYPTOKI_NOT_INITIALIZED	Cryptoki is not yet initialized.

FM_SetSlotUserData

This function can be used to associate user data with a slot. The data is associated with the slot identified by slotId. The function specified in this call will be called to free the data when the last application using the library finalizes.

If the slot already has associated user data it will be freed, by calling the current free function, before the new data association is created.

Synopsis

```
#include <objstate.h>
CK_RV FM_SetSlotUserData(FmNumber_t fmNo,
                          CK_SLOTID slotId,
                          CK_VOID_PTR userData
                          CK_VOID(*freeUserData)(CK_VOID_PTR));
```

Parameters

fmNo:	The fm number of the caller. It must be FM_NUMBER_CUSTOM_FM in this release of the software.
slotId:	The slot ID of the slot.
userData:	Address of the memory block that will be associated with the session handle. if it is NULL, the current associated buffer is freed.
freeUserData:	Address of a function that will be called to free the userData, if the library decides that it should be freed. It must be non-NULL if userData is not NULL.

Return Code

CKR_OK	The operation was successful.
CKR_ARGUMENTS_BAD	freeUserData was NULL, when userData was not NULL; or fmNo was not FM_NUMBER_CUSTOM_FM.
CKR_CRYPTOKI_NOT_INITIALIZED	Cryptoki is not yet initialized.

FM_GetSlotUserData

This function is used to obtain the userData associated with the specified slot. if there are no associated buffers, NULL is returned in ppUserData.

Synopsis

```
#include <objstate.h>
CK_RV FM_GetSlotUserData(FmNumber_t fmNo,
                         CK_SLOTID slotId,
                         CK_VOID_PTR userData
                         CK_VOID_PTR_PTR ppUserData);
```

Parameters

fmNo:	The fm number of the caller. It must be FM_NUMBER_CUSTOM_FM in this release of the software.
slotId:	The slot ID indicating the slot to be used.
ppuserData:	Address of a variable (of type CK_Void_PTR) which will contain the address of the user data if this function returns CKR_OK. It must be non-NULL.

Return Code

CKR_OK	The operation was successful. The associated user data is placed in the variable specified by ppUserData.
CKR_ARGUMENTS_BAD	ppUserData was NULL or fmNo was not FM_NUMBER_CUSTOM_FM.
CKR_SLOT_ID_INVALID	The specified slot ID is invalid.
CKR_CRYPTOKI_NOT_INITIALIZED	Cryptoki is not yet initialized.

FM_SetTokenUserData

This function can be used to associate user data with a token. The data is associated with the token in slotId by the library. The function specified in this call will be called to free the data when the last application using the library finalizes, or when the token is removed from that slot.

If the token already has associated user data it will be freed, by calling the current free function, before the new data association is created.

Synopsis

```
#include <objstate.h>
CK_RV FM_SetTokenUserData(FmNumber_t fmNo,
                          CK_SLOTID slotId,
                          CK_VOID_PTR userData
                          CK_VOID(*freeUserData)(CK_VOID_PTR));
```

Parameters

fmNo:	The fm number of the caller. It must be FM_NUMBER_CUSTOM_FM in this release of the software.
slotId:	The slot ID of the slot containing the token.
userData:	Address of the memory block that will be associated with the session handle. if it is NULL, the current associated buffer is freed.
freeUserData:	Address of a function that will be called to free the userData, if the library decides that it should be freed. It must be non-NULL if userData is not NULL.

Return Code

CKR_OK	The operation was successful.
CKR_ARGUMENTS_BAD	freeUserData was NULL or fmNo was not FM_NUMBER_CUSTOM_FM.
CKR_SLOT_ID_INVALID	The specified slot ID is invalid.
CKR_TOKEN_NOT_PRESENT	The specified slot does not contain a token.
CKR_CRYPTOKI_NOT_INITIALIZED	Cryptoki is not yet initialized.

FM_GetTokenUserData

This function is used to obtain the userData associated with the specified token. If there are no associated buffers, or if the token is not present, NULL is returned in ppUserData.

Synopsis

```
#include <objstate.h>
CK_RV FM_GetTokenUserData(FmNumber_t fmNo,
                          CK_SLOTID slotId,
                          CK_VOID_PTR_PTR ppUserData);
```

Parameters

fmNo:	The fm number of the caller. It must be FM_NUMBER_CUSTOM_FM in this release of the software.
slotId:	The slot ID of the slot containing the token.
ppUserData:	Address of a variable (of type CK_VOID_PTR) which will contain the address of the user data if this function returns CKR_OK. It must be non-NULL.

Return Code

CKR_OK	The operation was successful.
CKR_ARGUMENTS_BAD	ppUserData was NULL or fmNo was not FM_NUMBER_CUSTOM_FM.
CKR_SLOT_ID_INVALID	The specified slot ID is invalid.
CKR_CRYPTOKI_NOT_INITIALIZED	Cryptoki is not yet initialized.

FM_SetTokenAppUserData

This function can be used to associate user data with a token in the context of the calling application. The data is associated with the token (token, PID) pair. The function specified in this call will be called to free the data when the last application using the library finalizes, or when the token is removed from that slot.

If the token already has associated user data it will be freed, by calling the current free function, before the new data association is created.

Synopsis

```
#include <objstate.h>
CK_RV FM_SetTokenAppUserData(FmNumber_t fmNo,
                             CK_SLOTID slotId,
                             CK_VOID_PTR userData
                             CK_VOID(*freeUserData)(CK_VOID_PTR));
```

Parameters

fmNo:	The fm number of the caller. It must be FM_NUMBER_CUSTOM_FM in this release of the software.
slotId:	The slot ID of the slot containing the token.
userData:	Address of the memory block that will be associated with the session handle. if it is NULL, the current associated buffer is freed.
freeUserData:	Address of a function that will be called to free the userData, if the library decides that it should be freed. It must be non-NULL if userData is not NULL.

Return Code

CKR_OK	The operation was successful.
CKR_ARGUMENTS_BAD	freeUserData was NULL or fmNo was not FM_NUMBER_CUSTOM_FM.
CKR_SLOT_ID_INVALID	The specified slot ID is invalid.
CKR_TOKEN_NOT_PRESENT	The specified slot does not contain a token.
CKR_CRYPTOKI_NOT_INITIALIZED	Cryptoki is not yet initialized.

FM_GetTokenAppUserData

This function is used to obtain the userData associated with the specified token in the application context. If there are no associated buffers, or if the token is not present, NULL is returned in ppUserData.

Synopsis

```
#include <objstate.h>
CK_RV FM_GetTokenAppUserData(FmNumber_t fmNo,
                             CK_SLOTID slotId,
                             CK_VOID_PTR_PTR ppUserData);
```

Parameters

fmNo:	The fm number of the caller. It must be FM_NUMBER_CUSTOM_FM in this release of the software.
slotId:	The slot ID of the slot containing the token.
ppUserData:	Address of a variable (of type CK_VOID_PTR) which will contain the address of the user data if this function returns CKR_OK. It must be non-NULL.

Return Code

CKR_OK	The operation was successful.
CKR_ARGUMENTS_BAD	ppUserData was NULL or fmNo was not FM_NUMBER_CUSTOM_FM.
CKR_SLOT_ID_INVALID	The specified slot ID is invalid.
CKR_CRYPTOKI_NOT_INITIALIZED	Cryptoki is not yet initialized.

FM_SetSessionUserData

This function can be used to associate user data with a session handle. The data is associated with the (PID, hSession) pair by the library. The function specified in this call will be called to free the user data if the session is closed (via a C_CloseSession() or a C_CloseAllSessions() cal), or the application owning the session finalizes.

If the session handle already contains user data it will be freed, by calling the current free function, before the new data association is created.

Synopsis

```
#include <objstate.h>
CK_RV FM_SetSessionUserData(FmNumber_t fmNo,
                            CK_SESSION_HANDLE hSession,
                            CK_VOID_PTR userData,
                            CK_VOID (*freeUserData)(CK_VOID_PTR));
```

Parameters

fmNo:	The fm number of the caller. It must be FM_NUMBER_CUSTOM_FM in this release of the software.
hSession:	A session handle, which was obtained from an C_OpenSession() call. The validity of this parameter is checked.
userData:	Address of the memory block that will be associated with the session handle. If it is NULL, the current associated buffer is freed.
freeUserData:	Address of a function that will be called to free the userData, if the library decides that it should be freed. It must be non-NULL if userData is not NULL.

Return Code

CKR_OK	The operation was successful.
CKR_ARGUMENTS_BAD	freeUserData was NULL or fmNo was not FM_NUMBER_CUSTOM_FM.
CKR_SESSION_HANDLE_INVALID	The specified session handle is invalid.
CKR_CRYPTOKI_NOT_INITIALIZED	Cryptoki is not yet initialized.

FM_GetSessionUserData

This function is used to obtain the userData associated with the specified session handle. If there are no associated buffers, NULL is returned in ppUserData.

Synopsis

```
#include <objstate.h>
CK_RV FM_GetSessionUserData(FmNumber_t fmNo,
                            CK_SESSION_HANDLE hSession,
                            CK_VOID_PTR_PTR ppUserData);
```

Parameters

fmNo:	The fm number of the caller. It must be FM_NUMBER_CUSTOM_FM in this release of the software.
hSession:	A session handle, which was obtained from an C_OpenSession() call. The validity of this parameter is checked.
ppUserData:	Address of a variable (of type CK_VOID_PTR) which will contain the address of the user data if this function returns CKR_OK. It must be non-NULL.

Return Code

CKR_OK	The operation was successful. The associated user data is placed in the variable specified by ppUserData.
CKR_ARGUMENTS_BAD	ppUserData was NULL or fmNo was not FM_NUMBER_CUSTOM_FM.
CKR_SESSION_HANDLE_INVALID	hSession is not a valid session handle.
CKR_CRYPTOKI_NOT_INITIALIZED	Cryptoki is not yet initialized.

FM Header Definition Macro

The FM header contains information which is used at runtime and must be present in all FMs.

The use of the DEFINE_FM_HEADER macro simplifies the definition of FM header structure and also ensures that the header is placed in the appropriate location in the FM binary image.

#include <mkfmhdr.h>

Usage

DEFINE_FM_HEADER(FM_NUMBER, FM_VERSION, FM_SERIAL_NO, MANUFACTURER_ID, PRODUCT_ID);

FM_NUMBER:	Must be the manifest constant FM_NUMBER_CUSTOM_FM in this software version.
FM_VERSION:	A 16 bit integer, of the form 0xmmMM, where mm is the minor number, and MM is the major number (It is displayed as VMM.mm in ctconf). Example: V1.0f . is encoded as 0x0f01.
SERIAL_NO:	An integer representing the serial number of the FM.
MANUFACTURER_ID:	A string of at most 32 characters, which contains the manufacturer name. This does not need to be NULL terminated.
PRODUCT_ID:	A string consisting of a maximum of 16 characters, which contains the FM name. This does not need to be NULL terminated.

THIS PAGE INTENTIONALLY LEFT BLANK

Glossary

Adapter	The printed circuit board responsible for cryptographic processing in a HSM
API	Application Programming Interface
CA	Certification Authority
DLL	Dynamically Linked Library. A library which is linked to application programs when they are loaded or run rather than as the final phase of compilation.
FM	Functionality Module. A segment of custom program code operating inside the CSA800 HSM to provide additional or changed functionality of the hardware.
FMSW	Functionality Module Dispatch Switcher
HIFACE	Host Interface. It is used to communicate with the host system.
HSM	Hardware Security Module
PKCS#11	Also referred to as Cryptoki. A cryptographic token interface standard developed by RSA laboratories.
SDK	Software Development Kit
SO	Security Officer

END OF DOCUMENT