THALES

Thales Luna PCIe HSM 7

FM SDK GUIDE

Document Information

Last Updated	2024-05-03 15:49:56 GMT-04:00

Trademarks, Copyrights, and Third-Party Software

Copyright 2001-2024 Thales Group. All rights reserved. Thales and the Thales logo are trademarks and service marks of Thales and/or its subsidiaries and are registered in certain countries. All other trademarks and service marks, whether registered or not in specific countries, are the property of their respective owners.

Disclaimer

All information herein is either public information or is the property of and owned solely by Thales Group and/or its subsidiaries who shall have and keep the sole right to file patent applications or any other kind of intellectual property protection in connection with such information.

Nothing herein shall be construed as implying or granting to you any rights, by license, grant or otherwise, under any intellectual and/or industrial property rights of or concerning any of Thales Group's information.

This document can be used for informational, non-commercial, internal, and personal use only provided that:

> The copyright notice, the confidentiality and proprietary legend and this full warning notice appear in all copies.

> This document shall not be posted on any publicly accessible network computer or broadcast in any media, and no modification of any part of this document shall be made.

Use for any other purpose is expressly prohibited and may result in severe civil and criminal liabilities.

The information contained in this document is provided "AS IS" without any warranty of any kind. Unless otherwise expressly agreed in writing, Thales Group makes no warranty as to the value or accuracy of information contained herein.

The document could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Furthermore, Thales Group reserves the right to make any change or improvement in the specifications data, information, and the like described herein, at any time.

Thales Group hereby disclaims all warranties and conditions with regard to the information contained herein, including all implied warranties of merchantability, fitness for a particular purpose, title and non-infringement. In no event shall Thales Group be liable, whether in contract, tort or otherwise, for any indirect, special or consequential damages or any damages whatsoever including but not limited to damages resulting from loss of use, data, profits, revenues, or customers, arising out of or in connection with the use or performance of information contained in this document.

Thales Group does not and shall not warrant that this product will be resistant to all possible attacks and shall not incur, and disclaims, any liability in this respect. Even if each product is compliant with current security standards in force on the date of their design, security mechanisms' resistance necessarily evolves according to the state of the art in security and notably under the emergence of new attacks. Under no circumstances, shall Thales Group be held liable for any third party actions and in particular in case of any successful attack against systems or equipment incorporating Thales products. Thales Group disclaims any liability with respect to security for direct, indirect, incidental or consequential damages that result from any use of its products. It is further stressed

that independent testing and verification by the person using the product is particularly encouraged, especially in any application in which defective, incorrect or insecure functioning could result in damage to persons or property, denial of service, or loss of privacy.

All intellectual property is protected by copyright. All trademarks and product names used or referred to are the copyright of their respective owners. No part of this document may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, chemical, photocopy, recording or otherwise without the prior written permission of Thales Group.

Regulatory Compliance

This product complies with the following regulatory regulations. To ensure compliancy, ensure that you install the products as specified in the installation instructions and use only Thales-supplied or approved accessories.

USA, FCC

This equipment has been tested and found to comply with the limits for a “Class B” digital device, pursuant to part 15 of the FCC rules.

Canada

This class B digital apparatus meets all requirements of the Canadian interference-causing equipment regulations.

Europe

This product is in conformity with the protection requirements of EC Council Directive 2014/30/EU. This product satisfies the CLASS B limits of EN55032.

CONTENTS

PREFACE: About the FM SDK Programming Guide

A Functionality Module (FM) is a custom-developed, customer-specific code that operates within the secure confines of a Hardware Security Module (HSM).

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. It also describes the tools and requirements for the management of FMs on compliant HSMs.

FMs allow application developers to design security sensitive program code that can be loaded into the HSM to operate as part of the HSM firmware.

The FM concept allows developers to place their most sensitive algorithms within the logical and physical security perimeter of the HSM. A HSM is the pinnacle of a systems trust pyramid and ultimate solution to the threats of malicious tampering and secret exposure.

FMs can make extensive use of the HSM functionality, which is provided using a PKCS#11 compliant Application Programming Interface (API) and a rich set of commands available just to FMs.

The FM has access to tamper protected persistent storage so it can manage its own keys and critical parameters independently of the PKCS#11 objects.

The FM also has direct access to a RS232 interface (using a USB dongle) of the HSM and can use this port to implement a physically trusted path to an external device.

The Luna FM SDK package allows developers an extensive opportunity to create a large range of customized high security applications.

> **NOTE** This feature requires minimum Luna HSM Firmware 7.4.0 and Luna HSM Client 7.4.0.

> **NOTE** For Luna Network HSM 7s, the Luna HSM Client accesses application partitions via NTLS or STC connection, causing the registered application partitions to appear as slots in the LunaCM slot list, just as if they were slots on Luna PCIe HSM 7 cards installed locally in the Luna HSM Client host computer.
>
> > For local Luna PCIe HSM 7s, the HSM Admin (SO) partition (a.k.a. the HSM Admin "**Token**" in deference to Cryptoki terminology) *also* appears in the slot list and is directly accessible.
> >
> > Be aware that for Luna Network HSM 7s the HSM Admin partition (HSM Admin **token**) *must be accessed over SSH via the appliance's LunaSH administrative interface*, and is not visible or accessible via the Client.

This document describes how to use the FM SDK to write, test, install, and use functionality modules to provide custom functions on the HSM. It contains the following chapters:

> "Setup" on page 14

> "FM Architecture" on page 16

The preface includes the following information about this document:

For information regarding the document status and revision history, see "Document Information" on page 2.

Customer Release Notes

The Customer Release Notes (CRN) provide important information about specific releases. Read the CRN to fully understand the capabilities, limitations, and known issues for each release. You can view the latest version of the CRN at www.thalesdocs.com.

Audience

This document is intended for personnel responsible for maintaining your organization's security infrastructure. This includes Luna HSM users and security officers, key manager administrators, and network administrators.

All products manufactured and distributed by Thales are designed to be installed, operated, and maintained by personnel who have the knowledge, training, and qualifications required to safely perform the tasks assigned to them. The information, processes, and procedures contained in this document are intended for use by trained and qualified personnel only.

It is assumed that the users of this document are proficient with security concepts.

Document Conventions

This document uses standard conventions for describing the user interface and for alerting you to important information.

Notes

Notes are used to alert you to important or helpful information. They use the following format:

> **NOTE** Take note. Contains important or helpful information.

Cautions

Cautions are used to alert you to important information that may help prevent unexpected results or data loss. They use the following format:

> **CAUTION!** Exercise caution. Contains important information that may help prevent unexpected results or data loss.

Warnings

Warnings are used to alert you to the potential for catastrophic data loss or personal injury. They use the following format:

> ****WARNING** Be extremely careful and obey all safety and security measures. In this situation you might do something that could result in catastrophic data loss or personal injury.**

Command syntax and typeface conventions

Format	Convention
bold	The bold attribute is used to indicate the following: > Command-line commands and options (Type **dir /p**.) > Button names (Click **Save As**.) > Check box and radio button names (Select the **Print Duplex** check box.) > Dialog box titles (On the **Protect Document** dialog box, click **Yes**.) > Field names (**User Name**: Enter the name of the user.) > Menu names (On the **File** menu, click **Save**.) (Click **Menu** > **Go To** > **Folders**.) > User input (In the **Date** box, type **April 1**.)
italics	In type, the italic attribute is used for emphasis or to indicate a related document. (See the *Installation Guide* for more information.)

Format	Convention
<variable>	In command descriptions, angle brackets represent variables. You must substitute a value for command line arguments that are enclosed in angle brackets.
[**optional**] [<optional>]	Represent optional **keywords** or <variables> in a command line description. Optionally enter the keyword or <variable> that is enclosed in square brackets, if it is necessary or desirable to complete the task.
{**a**|**b**|**c**} {<a>|<b>|<c>}	Represent required alternate **keywords** or <variables> in a command line description. You must choose one command line argument enclosed within the braces. Choices are separated by vertical (OR) bars.
[**a**|**b**|**c**] [<a>|<b>|<c>]	Represent optional alternate keywords or variables in a command line description. Choose one command line argument enclosed within the braces, if desired. Choices are separated by vertical (OR) bars.

 Copyright 2001-2024 Thales Group

Support Contacts

If you encounter a problem while installing, registering, or operating this product, please refer to the documentation before contacting support. If you cannot resolve the issue, contact your supplier or Thales Customer Support. Thales Customer Support operates 24 hours a day, 7 days a week. Your level of access is governed by the support plan negotiated between Thales and your organization. Please consult this plan for details regarding your entitlements, including the hours when telephone support is available to you.

Customer Support Portal

The Customer Support Portal, at https://supportportal.thalesgroup.com, is where you can find solutions for most common problems and create and manage support cases. It offers a comprehensive, fully searchable database of support resources, including software and firmware downloads, release notes listing known problems and workarounds, a knowledge base, FAQs, product documentation, technical notes, and more.

> **NOTE** You require an account to access the Customer Support Portal. To create a new account, go to the portal and click on the **REGISTER** link.

Telephone

The support portal also lists telephone numbers for voice contact (Contact Us).

 Copyright 2001-2024 Thales Group

CHAPTER 1: Setup

FM developers should ensure that their development environment is configured correctly and that all required files and library locations are set. This section 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:

FMSDK	Specifies the installed location of the Luna FM SDK package if it is not installed in the default location.

Software Installation

Refer to the installation guide for hardware and software installation instructions. See Luna HSM Client Software Installation.

Install all device drivers and Luna HSM Client software.

If the server is to be used for FM creation then you need to install these:

> eldk-5.6.fm package

> Luna FM SDK package

All servers need to also install:

> Luna FM Tools package

Requirements

The Luna FM SDK package provides the tools and sources to allow a developer to create and sign a FM and to load that FM into a compliant HSM.

Creating an FM:

A Linux operating system is required to perform the FM build.

For a list of supported platforms see the Customer Release Notes.

Signing an FM:

To sign a FM you can use **mkfm**. This tool requires a PKCS#11 implementation capable of 2048 bit CKM_SHA512_RSA_PKCS signature operation. Any Luna HSM would be suitable for this purpose. However, a smart card or other type of HSM would suffice.

Compliant HSM:

You can use the Luna FM SDK package to develop FMs for the Luna HSMs that were introduced in Luna HSM Firmware 7.4.0. A Luna Network HSM 7 or Luna PCIe HSM 7 with capability to host FM is required.

Before any FM can be loaded the HSM must have the FM capability configured. The **ctfm** utility will report if the HSM is not configured for FMs.

HSM Recovery:

If an HSM becomes unresponsive due to a malformed or buggy FM being loaded, then the HSM needs to be restored by erasing the FM.

For Luna PCIe HSM 7s, see "fmrecover" on page 53

 Copyright 2001-2024 Thales Group

CHAPTER 2: FM Architecture

FM Support within the HSM Hardware

> **NOTE** For Luna Network HSM 7s, the Luna HSM Client accesses application partitions via NTLS or STC connection, causing the registered application partitions to appear as slots in the LunaCM slot list, just as if they were slots on Luna PCIe HSM 7 cards installed locally in the Luna HSM Client host computer.
>
> > For local Luna PCIe HSM 7s, the HSM Admin (SO) partition (a.k.a. the HSM Admin "**Token**" in deference to Cryptoki terminology) *also* appears in the slot list and is directly accessible.
> >
> > Be aware that for Luna Network HSM 7s the HSM Admin partition (HSM Admin **token**) *must be accessed over SSH via the appliance's LunaSH administrative interface*, and is not visible or accessible via the Client.

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

The FM may contain custom-designed functions that are in addition to the native command set of the HSM.

The following diagram shows the various components of the FM system, relevant to the host and HSM.

The 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 Luna FM SDK package.

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

```
API (Function Type)
```

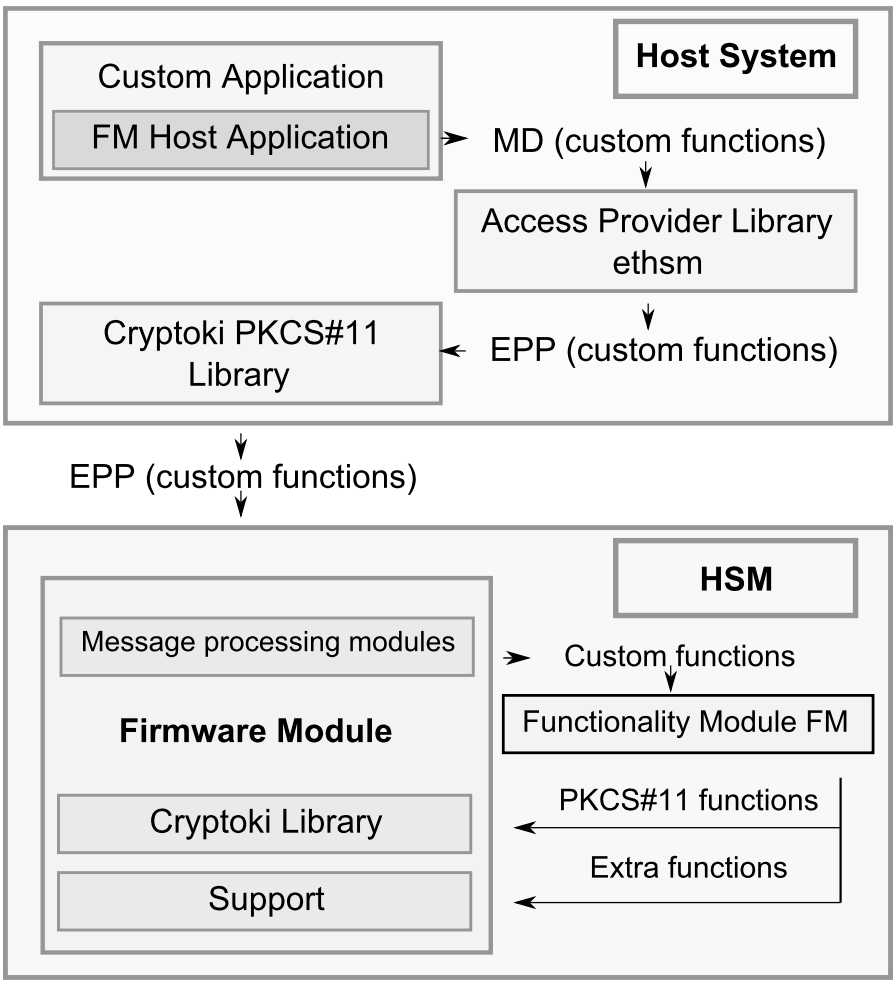
API refers to the API used to interface between the two components and function type indicates the type of function.

> **NOTE** Functionality Modules (FMs) in the HSM run in single-thread mode. multi-threading multi-process

Custom Functions

The following figure 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 Access Provider (**ethsm** library), via the Message Dispatch API.

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 native PKCS#11 function calls plus the extra commands specified in this document.

FM Support in Emulation Mode

At this stage no emulation mode is supported.

Multiple FMs

The Luna HSMs can support more than one FM loaded at the same time.

Concept

A developer will use more than one FM because each FM is implementing a different solution and operates with different applications.

Features

The total number of FMs that the HSM can hold depends on the storage limits. Each FM image is placed in a reserved area of persistent memory which has a fixed size. So the HSM can only hold as many FM images as will fit in this space. The current size of the FM store is 8 megabytes.

Each FM loaded into a HSM must have a unique ID number. The FM ID is an integer value that is set in the FM image through the use of the **DEFINE_FM_HEADER** macro.

The FMs behaviour is dependent on multiple factors:

> If a new FM is loaded with the same ID value as a FM already loaded then the new FM will replace the existing FM.

> If a new FM is loaded with a ID value different to any FM already loaded then the new FM is stored and all existing FMs are untouched.

The order in which the FM images are initialized is set by the FM ID values. The HSM will call the Startup() in the FMs starting with the lowest FM ID value and finishing with the highest ID number.

Memory for FMs

The Luna HSM provides the FM developer with several memory types to use.

> Fixed Read Only memory including executable – 8 MB

> SMFS persistent tamper protected memory – 4 MB

> Partition storage using Cryptoki function to manage objects on a partition

> RAM available with malloc() – about 100 MB

Message Handling

Custom commands are always directed to a particular FM identified by the unique FM ID.

The **MD_SendReceive** function on the host allows the caller to specify the HSM index and the FM ID values. The host side software and HSM FW will ensure that the message is sent to the correct FM

CHAPTER 3: FM Development

The following chapter describes the recommended development cycle to be undertaken when developing custom functionality modules. The recommended development lifecycle contains the following stages:

> "Initial Development" on the next page

> "Adapter Build" on the next page

> "Adapter Test" on the next page

> "Production Build" on page 21

> "Acceptance Test" on page 21

> "Key Management" on page 21

> **NOTE** File paths in the examples in this chapter are Linux specific, as development of FMs is done on Linux, only, for this release. Development of host-side FM applications (that call an FM on your HSM) can be done on Windows, as well as Linux.
>
> TERMINOLOGY NOTE - "Token" is the Cryptoki term for what is in a crypto slot. "Partition" is the Luna term for a Cryptoki token. The Admin Token is the HSM Admin or SO partition and exists while the HSM is in initialized state, distinct from user/application partitions that you can create and delete and reinitialize at will.

> **NOTE** For Luna Network HSM 7s, the Luna HSM Client accesses application partitions via NTLS or STC connection, causing the registered application partitions to appear as slots in the LunaCM slot list, just as if they were slots on Luna PCIe HSM 7 cards installed locally in the Luna HSM Client host computer.
>
> > For local Luna PCIe HSM 7s, the HSM Admin (SO) partition (a.k.a. the HSM Admin "**Token**" in deference to Cryptoki terminology) *also* appears in the slot list and is directly accessible.
>
> > Be aware that for Luna Network HSM 7s the HSM Admin partition (HSM Admin **token**) *must be accessed over SSH via the appliance's LunaSH administrative interface*, and is not visible or accessible via the Client.

Lifecycle Overview

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

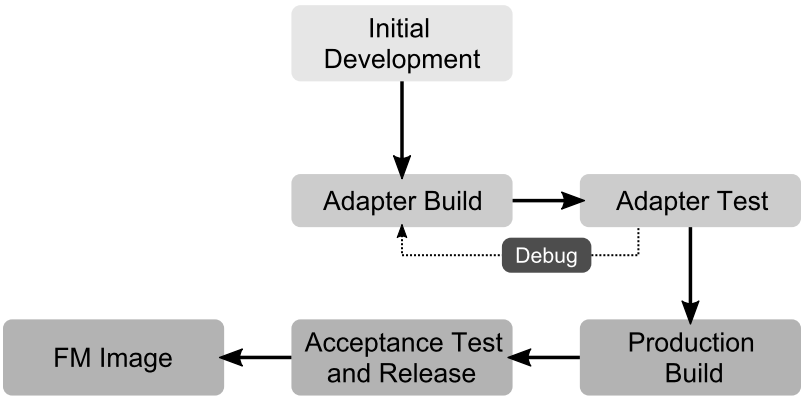


Figure 2: Figure 1 – FM Development Lifecycle

The development process consists of the following stages:

Initial Development:

Includes the design and development of the functionality application code for initial testing.

Adapter Build:

The functionality module should now be built for the HSM environment. The FM is loaded 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.

Adapter Build

The FM must be tested in the HSM.

In this phase, the developer generates the binary FM image in DEBUG mode, and signs it using either a temporary or a permanent development key. Once the image is signed, it can be loaded to the HSM for the next stage of testing. Refer to the "FM Architecture" on page 16 for further details.

Adapter Test

In the HSM test stage, the debug build of the FM is tested in its production environment.

> **NOTE** If problems are detected during this stage, the developer should use trace messages from the FM to resolve the problem.

Production Build

When the implementation of the FM and the host-side code is correct, a production build of the system is performed. Refer to the "FM Architecture" on page 16 for further 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 loaded into the HSM must have an assigned signature. FMs are executed inside the HSM only 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. Thales 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[1] 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 (the HSM SO) will be required to install the certificate in the Admin Token. The verification and loading of the FM requires the HSM Administrator to provide the Admin Token password, enforcing the presence of the HSM Administrator during the loading operation.

> **NOTE** The Admin partition (a.k.a. Admin Token) is not directly available to client servers using the Luna Network HSM 7. Operators of Luna Network HSM 7s must use the LunaSH **hsm fm** commands to load FM Certificates into the HSM Admin Token.

As previously advised, there is no pre-defined key management scheme for the private key and the certificate. 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 the FM 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

[1] the Cryptoki term for what is in a slot; in Luna HSMs, a partition

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 on a Luna PCIe HSM 7 is to generate a new self-signed key/certificate pair in the Admin token of the target HSM. This can be done using the Thales CMU tool.

For the Luna Network HSM 7 a normal User Token should be used to hold the signing key and create the FM Certificate. The certificate can then be extracted and used with the LunaSH **hsm fm load** command.

There should be a production grade HSM to hold the production key/certificate pair.

The raw FM binary image must 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 loaded into the HSM using the **ctfm** utility (on Luna PCIe HSM) or the LunaSH **hsm fm load** command (on Luna Network HSM 7).

Contents of the Luna FM SDK package directory

When installed, the Luna FM SDK package directory (/usr/safenet/lunafmsdk Directory [Linux] or C:\Program Files\SafeNet\LunaClient\samples\fmsdk Directory [Windows]) contains the following:

Directory	File Description
include/	Header files specific to FM and FM Host Application development. These are used together with headers provided by Luna HSM SDK.
lib/libfmsupt.a	Static libraries used for building an FM that will run in an HSM.
samples/	Sample FMs and FM Host Applications
samples/fmconfig.mk	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.

> **NOTE** Samples for Functionality Modules rely on the ELDK library that is installed only for Linux. The host samples work for both Linux and Windows.

SDK Installation Tips

Set the Environment

The tools are all located in the **/usr/safenet/lunaclient/bin** folder so it will be convenient to add this folder to the PATH. For example:

export PATH=$PATH:**/usr/safenet/lunaclient/bin**

The libraries to support the tools are all located in the **/usr/safenet/lunaclient/lib** folder.

The Luna tools will use **/etc/Chrystoki.conf** to obtain the location of the libraries they need. However, the FM Test applications only use the standard system library search methods. As a result, when running FM Test applications it is convenient to add the lunaclient lib folder to the LD_LIBRARY_PATH. e.g.:

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:**/usr/safenet/lunaclient/lib**

> **NOTE** For Windows environments, set a search path to the libraries.
>
> **Example:** for a default installation, set:
>
> `PATH=C:\Program Files\SafeNet\LunaClient\`
>
> and
>
> `LD_LIBRARY_PATH=C:\Program Files\SafeNet\LunaClient\`
>
> Adjust your path statement if you selected a non-default install path during Luna HSM Client software installation.

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 Key Object in one of the tokens (partitions). 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 two possible solutions to this problem:

Using Privilege Level

The **CT_SetPrivilegeLevel** function allows a simple solution to the problem stated above. The FM can make a call to temporarily obtain the rights to read sensitive object attributes.

This allows the FM designer to create and manage their keys using the tools provided with the HSM. This increases security by allowing the user access from the trusted FM, while reducing the risk from external programs.

Using the SMFS

The Secure Memory File System (**SMFS**) provides access to key storage area that is exclusively for the use of FM developers, 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 that 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.

The **SMFS** system uses a Open read/write Close paradigm.

NOTE Functionality Modules that open an **SMFS** file and keep the handle open should take into account the following:

> 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 do backup and restore only.
> Keep the keys in normal memory while the FM is running.
> Restore the keys from 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.

Similar to the way a Luna partition cannot be used by an application until an operator has activated it by logging on to it, the SMFS cannot be used until it is Activated. SMFS Activation occurs when the HSM Security Officer or Administator issues the SMFS Activation command - **"ctfm" on page 48** (specifically, the "a" option) for Luna PCIe HSM 7.

The HSM can be configured to automatically Activate on each startup, by setting the appropriate Security Policy. Other Security Policy flags affect the behavior of FMs – please refer to HSM Capabilities and Policies.

Because the HSM does not pass control to any FMs until after the SMFS is activated the FM designer can assume that the SMFS is always Activated.

Scatter Gather FM Message Dispatching

FM Message dispatching (from client to HSM) support allows for more than one request buffer, and more than one reply buffer to be presented to the HSM in one command. 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 - 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 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.

This behaviour is handled via two possible cases:

> 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.

> 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 IDs

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

The functions **FM_GetCurrentAppId**, and **FM_SetCurrentAppId** allow you to know what process is sending the current message.

If your FM supports the concept of a user login then you will need to track which host processes have logged in.

You can remember which process has logged in by storing the AppId 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 AppId 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 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 AppId to a constant value that the underlying Cryptoki sees by using the **FM_SetCurrentAppId** calls prior to making any Cryptoki calls.

> **NOTE** The value –1 for AppId is a suitable choice for this purpose.

C_CloseAllSessions - Notes

Special consideration should be made of the C_CloseAllSesions function call.

Because the FM and the calling process share the same AppID (see previous section) then HSM considers any session opened using C_OpenSession() by either the Host side application or the FM (on behalf of the calling process), to be owned by the same Application.

So if either the FM or the Application calls C_CloseAllSessions then all sessions owned by that AppId are closed.

An example illustrates this:-

> The Application opens a session and gets session handle 1.

> The Application calls the FM and the FM opens a session it gets session 2.

> The HSM now thinks this application owns two sessions.

> If either the Application or the FM calls C_CloseAllSession then both sessions will be closed.

A well designed solution will clean up its own sessions with C_CloseSession() and avoid the use of C_CloseAllSessions().

Memory Alignment Issues

The PowerPC processor in the Luna PCIe HSM 7 does not require fully aligned memory access, however unaligned access incurs a performance cost.

Memory Endian Issues

The processor in the Luna PCIe HSM 7 is big endian, where the processors in PTK based 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, such as **fm_htobe16**, are provided in the header file **fm_byteorder.h**.

Compiling FMs

The sample FMs provided in the Luna FM SDK package include makefiles to script the compiling and linking of the sample FMs. These makefiles are written to be compatible with the GNU make utility.

When you write your own FM you can start by copying one of the sample FMs. You can remove the unnecessary code and substitute in new code on an as-needed basis. The FM makefile will also need to be modified to match the set of source files for the new FM.

The compiler is available at this location:

/opt/eldk-5.6/powerpc-4xx/sysroots/i686-eldk-linux/usr/bin/powerpc-linux/powerpw-linux-gcc

However, if you do not want to use the GNU make utility, the following sections will give basic instructions on what actions are required for compiling and linking FMs.

Include Path

The -I option is required to tell the compiler where to get the Luna FM SDK package header files.

To build a Luna FM

Use these folders in the following order:

1. /usr/safenet/lunaclient/include
2. /usr/safenet/lunafmsdk/include
3. /usr/safenet/lunafmsdk/include/fm/hsm
4. /usr/safenet/lunafmsdk/include/fm/host

PPO Compatibility INCLUDE Files

FMs written for the PPO SDK will have specific Thales ProtectToolkit (PTK) specific identifiers which are not part of the Luna FM SDK package. The Luna FM SDK package includes compatibility headers which will ease the porting of PPO source to a Luna FM environment.

To build a Luna FM from PPO FM source

Use these folders in the following order:

1. /usr/safenet/lunaclient/include
2. /usr/safenet/lunafmsdk/include
3. /usr/safenet/lunafmsdk/include/fm/ppo-compat
4. /usr/safenet/lunafmsdk/include/fm/ptk-compat

DEFINES

These are the minimum –D Flags required when compiling a FM with Eldk.

```
#     OS_LINUX - needed by cryptoki header files (cryptoki_v2.h)
#     _GNU_SOURCE   -- required to specify correct c runtime lib support
#     DISABLE_CA_EXT -- tell cryptoki_v2.h not to include Luna Extension header (not used in
FM)
#     IS_BIG_ENDIAN -- defines the hsm endian is big - only required if using PPO compat
headers
DEFINES +=  -DOS_LINUX -DIS_BIG_ENDIAN -D_GNU_SOURCE -DDISABLE_CA_EXT
```

C_Flags

The following are the minimum C FLags required when compiling a FM:

```
-fPIC -ffreestanding -std=c99
```

L_Flags

The FM is linked as a shared object which exports some symbols but imports none (including the standard C runtime library).

Only two libraries are required to link the FM: the libfmsupt.a static library from Luna FM SDK package and the libgcc.a from /opt/eldk-5.6. As a result, these linkage flags must be passed to the GNU compiler.

```
L_FLAGS = -shared -zdefs -nostdlib -WI, -static -WI, --gc -sections -L
/usr/safenet/lunafmsdk/lib -lfmsupt -lgcc
```

Building Applications that Talk to FMs

The Sample FMs provided in the FM SDK include makefiles to script the compiling and linking of the test applications that communicate with the sample FMs.

These makefiles are written to be compatible with the GNU make utility on Linux.

When you write your applications to communicate with your own FM you can start the design by copying one of the sample FMs test application source and strip out the code not needed and add new code as appropriate.

The host/makefile will also need to be modified to match the set of source files of the new FM.

However, if you do not want to use the GNU make utility the the following sections will give basic instructions on what options are required for compiling and linking applications to communicate to FMs.

INCLUDE PATH

The -I option is required to tell the compiler where to get the Luna Cryptoki and MD API header files.

To build a Luna FM Application

Include this folder:

> /usr/safenet/lunafmsdk/include

PPO Compatibity INCLUDE Files

Applications written for PTK and/or the PPO SDK will reference specific identifiers which are not part of the Luna FM SDK package. The Luna FM SDK package includes compatibility headers which will ease the porting of PPO/PTK source to the Luna FM SDK environment.

To build an application from PPO/PTK source

Use these folders in the following order:

1. /usr/safenet/lunafmsdk/include/fm/ppo-compat
2. /usr/safenet/lunafmsdk/include/fm/ptk-compat
3. /usr/safenet/lunaclient/include
4. <other_folders>

L_FLAGS

There are two libraries which give an application access to the HSM:

> /usr/safenet/lunaclient/lib/libCryptoki_64.so

> /usr/safenet/lunaclient/lib/libethsm.so

You can design the application to load the shared libraries at runtime or load time.

Runtime

Use the **dlopen()** and **dlsym()** to load libCryptoki2_64.so and/or libethsm.so and fetch entry points as required.

The path to the libraries can be controlled by configuration implemented in the application.

Load Time

Link the application against the shared libraries when linking the application.

When running the application you need to use the LD_LIBRARY_PATH environment variable to point to /usr/safenet/lunaclient/lib or use that -rpath option to tell the application where to find the libraries.

 Copyright 2001-2024 Thales Group

Troubleshooting

SMFS Activation Delay

On FM-enabled HSMs with SMFS auto-activation enabled, there is a delay of approximately 2 seconds for the SMFS to activate after an HSM or driver reboot. For scripted operations, this could cause a host app to fail if it queries the FM immediately after a reboot.

If you encounter this issue, include a 2-second sleep in the script between the reboot and the first FM query.

 Copyright 2001-2024 Thales Group

CHAPTER 4: Comparing PTK to Luna FM SDK, and Porting FMs

This chapter describes how to compare the Luna FM SDK and PTK FM SDK to determine what solution is best for your application and how you would port a PTK FM to Luna.

The actual differences between the FM SDK products will change as new versions of the respective products are released.

In this chapter we compare :

Luna FM SDK	7.4
PTK PPO	5.7

FM source code designed for the PTK PPO SDK will need to be changed before it can be successfully compiled and linked with the Luna FM SDK.

The Luna FM SDK has an overlapping feature set with the PPO feature set.

There are functions available in Luna FM SDK but not in PPO and vice-versa.

The amount of effort to convert your FM source from PPO to Luna FM SDK will depend on which features your FM uses.

The following sections describe these differences.

Summary

This section summarizes the features that differ between PTK and Luna FM SDK and some suggestions on how to migrate FM code that uses them.

Feature	Overview	Details
HSM Management and Security Features	Luna has optional multifactor quorum authentication and tool set is different.	
Programming APIs	Luna has new FMCE Api and limited CipherObj support, Cryptoki object and mechanism lists differ.	See "FMCE API and CipherObj" on page 33
Software Emulation	Only PTK has an emulation for FM development	
Cryptoki function Patching	Only the PTK can allow the FM to intercept and replace PKCS#11 function calls.	See "PTK Function Patching" on page 35

Feature	Overview	Details
Smart card key backup	Luna HSM Firmware 7.4.0 and newer supports Cloning only to another FM Enabled HSM as a backup scheme. PTK can use smartcard as a PKCS#11 slot for data storage	No Luna Backup HSM G5 supported for either Luna or PTK.
Net server support for allow network access to a PCI HSM	Luna has no equivalent	
Enter keys from components by keyboard + Verifone PinPad	Keyboard entry on Luna is possible with special tool	No pinpad support on Luna unless an FM implements that support itself.

HSM Management and Security Features

Here is a description of roles and services, key management policy and authentication techniques for each HSM.

Configuration

The PTK HSM may be ordered with different performance levels but otherwise all configuration is done by the Administrator setting security policy flags.

The Luna HSM may be ordered with different Capabilities for performance, key extraction policy, maximum user tokens and, in addition, the Security Officer can change Security Policy settings.

Roles

HSM	Administration Partition	User Partition
PTK	Admin SO Administrator Auditor	Security Officer User
Luna	HSM Security Officer Administrator Auditor	Partition Security Officer Crypto Officer Crypto User

Authentication and Activation

Luna

Luna HSMs employ a model wherein all material on the HSM remains encrypted except when items are temporarily decrypted into volatile HSM memory for use. Thus any power loss or tamper event causes decrypted material to evaporate, with no overt action needed. Luna requires authentication to each partition before any

cryptographic operation can be performed on that partition.

For a password-authenticated partition, that authentication is done by means of a password (something you know), so the FM can provide that password during C_Login.

If the Luna HSM is multifactor quorum-authenticated, then authentication is layered, requiring authentication with a physical token (something you have) before authentication with a challenge password (something you know), that can be provided by the FM during C_Login.

The first layer is called Activation and requires that the relevant PED key(s) be presented via the Luna PED. A multifactor quorum-authenticated HSM or partition requires appropriate PED keys for each of the HSM Security Officer, Administrator, Partition Security Officer, and Crypto Officer and Crypto User Roles.

For convenience of operation, Activation remains in force until interrupted or until logout is performed. Activation can be made persistent by enabling Auto-activation, which allows Activation to automatically reassert following interruptions or outages up to two hours in duration.

So your FM will not be able to access any keys on a Cryptoki partition until that partition is activated, but as long as the partition remains activated, the process of authentication by the FM is identical to the authentication process on a password-authenticated partition.

Activation occurs manually when a user is authenticated to the Token (logs in), or Automatically if Auto-Activation Policy is enabled.

In addition to the requirement for a physical token to place the partition in a state receptive to password login, some very high-security regimes can choose to employ additional factors for authentication:

> The PED keys can optionally be initialized with MofN split, or shared, secret (also called quorum), thus requiring that more than one trusted person must be present, with their portion of the split secret, in order to perform that operation. So, for example, if the HSM SO secret was initialized as M=2 and N=5, then 5 people would each be given a blue PED key containing a 1/5 portion of the secret, and 2 of those people (any two) would be needed whenever the HSM SO wanted to log in.

> Each PED key can optionally be assigned a multi-digit PIN that must be entered on the Luna PED keypad when that PED key is presented. A PIN is an optional (until invoked) Luna PED function, and is completely unrelated to the challenge password that is presented by applications and FMs.

PTK

PTK requires no Activation – it uses a Tamper Response model – so keys are always available unless the HSM has detected a Tamper event.

PTK can optionally require an OTP Token for Administrator and/or User Roles.

Both HSMs require a user to login before they can access private keys.

The FM can login with C_Login on PTK HSMs to access Private Objects in a partition.

What Does It Mean?

If your Luna partition uses a Luna PED then your client application needs to log in before calling FM functions. Once the SO has activated the partition with the appropriate PED key, authentication process is the same as for a password authenticated partition.

If your Luna partition uses passwords then you can perform login from the client or the FM.

Tool Set

The PTK and Luna have these tools :

PTK Tool	Luna Tool
Hsmreset	LunaSH hsm reset lunareset /dev/k7kp0
Hsmstate	LunaCM, LunaSH
Ctconf	LunaCM, LunaSH
Ctkmu	CMU, ckdemo
Ctcert	CMU
Ctstat	LunaCM, LunaSH
Ctcheck	LunaCM, LunaSH
Graphical KMU	-
Windows ctbrowse	-
gctAdmin	-
-	Salogin

Per Partition SO introduced by Admin

PTK Requires Administrator role to create user partitions/Tokens, while Luna uses the Security Officer Role.

PTK in FIPS mode requires Administrator role to allow a user partition to be initialized with Partition SO (Admin introduces SO) but Luna never requires an SO for this.

FM Programming APIs

FMCE API and CipherObj

Luna has new FMCE API to provide raw key cryptography similar to the CipherObj model used in PTK.

The Luna FM SDK also has a limited CipherObj support to help with porting – refer to Luna FM SDK User Guide for a complete list.

The actual list of mechanism supported in FMCE, CipherObjs and Cryptoki interface overlap but differ between the HSMs. Refer to Luna and PTK User documentation for complete lists.

Public Key Certificate Management

PTK supports CSR and Public Key Certificate creation in firmware using extension mechanisms so these capabilities are available to PTK FMs.

Luna performs these operations in the client side CMU tool and so the FM has no access to these operations and must provide any implementations itself.

Cryptoki Attributes

The PTK Cryptoki provides extension attributes which are not in the Luna Cryptoki.

- CKA_ISSUER_STR
- CKA_SUBJECT_STR
- CKA_SERIAL_NUMBER_INT
- CKA_SERIAL_NUMBER_STR
- CKA_ENUM_ATTRIBUTE
- CKA_EXPORT
- CKA_EXPORTABLE
- CKA_IMPORT
- CKA_KEY_SIZE
- CKA_TIME_STAMP
- CKA_TRUSTED

Client and FM Extension Functions

These functions are not supported. There is no workaround.

- CT_InitToken()
- CT_ResetToken()
- CT_CopyObject
- CKM_DECODE_PKCS7
- FM_SetAppUserData
- FM_SetTokenUserData
- FM_SetTokenAppUserData
- FM_SetSlotUserData
- CT_PresentTicket

These functions are not supported but there is a workaround.

- FM_GetCurrentOid() and FM_GetCurrentPid()
- Use FM_GetCurrentAppId()

JHSM

PTK has JHSM, a Java interface for custom command dispatch i.e. Java version of MD_SendReceive(),

The PTK version should work with the Luna ethsm library.

Compatibility Header Files

In addition to the normal Luna FM SDK header set there are compat headers (detailed in the FM SDK User Guide).

These headers emulate PTK manifest constants and functions using the normal Luna headers. They are provided to ease PTK FM porting to Luna.

PTK Function Patching

OS_GetCprovFuncTable()

This function is not implemented in the Luna HSM.

Because this feature is missing it is not possible to patch/intercept Cryptoki calls to the HSM.

The reason is that Luna HSM Client functions are not equivalent to Cryptoki requests; they make up a custom protocol called PcProt.

PTK FMs developers had two reasons to patch the cryptoki table.

> Since all administration of the PTK HSM is done with Cryptoki calls the management of the HSM could be modified.

> Custom Mechanisms could be implemented.

Administration Patching

If the Cryptoki function patching is intended to enforce a new security policy (for example, controlling the slot on which an application can open a session on) then this type of FM capability cannot be ported to Luna FM SDK.

Read Only Tokens Solution

If the requirement is to have keys in a user slot visible to the Client but is otherwise Read-only, then the following solution can be used:

> Set Usage and extraction Attributes of all key objects to false

> Login the client application as Crypto User Role and pass object handles to the FM using Custom functions.

> The FM can use CT_SetPrivilageLevel to achieve Crypto Officer access to the keys/partition.

Custom Mechanisms

If the Cryptoki patching is used to implement a custom mechanism then you can use the following workaround.

Implement a custom request to the FM to replicate the functionality of the patched Cryptoki function call.

Several issues arise when designing a custom function to replace a Cryptoki call:

> **Object handles:** these must be passed to the Custom function. Users should assume they can be encoded as 32 bit Integers.

> **Slot Numbers:** slot numbers seen by the application must be converted to the equivalent embedded Cryptoki slot number using MD_GetEmbeddedSlotID() and associated HSM index number with MD_GetHsmIndexForSlot().

> **Session Handles:** session handles provided by the host side Cryptoki library to the application cannot be used by the FM because they will not be recognized by the Embedded Cryptoki library used by the FM.

Therefore the FM must open its own Cryptoki sessions.
Depending on the system architecture the developer can either:

- For multi-part commands: Implement a custom command to open a session and return the handle to the application where it can be passed in with each subsequent call or
- For Single Part Commands: find slot number of the Application session (use C_GetSessionInfo) and translate it to the matching embedded slot MD_GetEmbeddedSlotID() then send that slot number to with your custom command request. The FM can use that slot number to temporarily open a session locally to do the single part command.

> Object Attributes: the implementer needs to encode/decode any object attributes that will be passed in or out as part of the Custom command

Porting ProtectServer FMs

FM source code designed for the legacy ProtectServer FM SDK will need to be converted to be successfully compiled and linked with the Luna FM SDK package.

The amount of effort to convert your FM source from PPO to Luna FM SDK package will depend on which features your FM uses.

A large number of PPO features are provided by Luna FM SDK package either directly or through the compatibility headers.

The following section lists features missing from Luna FM SDK package and provides some suggestions on how to migrate the FM code that uses them.

See "Compiling FMs" on page 26 for more information on converting legacy PTK FMs for use with the Luna FM SDK package.

FM_GetCurrentOid() and FM_GetCurrentPid()

See "FM_GetCurrentAppId" on page 150

FM_SetAppUserData, FM_SetTokenUserData, FM_SetTokenAppUserData, FM_SetSlotUserData

These functions are not supported. There is currently no workaround.

OS_GetCprovFuncTable()

This is the largest change. As a result of this feature missing, it is not possible to patch Cryptoki calls to the HSM.

If the Cryptoki function patching is intended to enforce a new security policy (i.e controlling what slot an application can open a session on) then this feature cannot be ported to Luna FM SDK package.

However, if the patching is used to implement a custom mechanism, then the work around is to implement a custom request to the FM to replicate the functionality of the patched Cryptoki function call.

There are several issues which arise when designing a custom function to replace a Cryptoki call:

Object Handles	These need to be passed to the Custom function. Users should assume they can be encoded as 32 bit integers.
Slot Numbers	Slot numbers seen by the application must be converted to the equivalent embedded Cryptoki slot number using **MD_Get_EmbeddedSlotID()** and the associated HSM index number with **MD_GetHsmIndexForSlot()**.
Session Handles	Session handles provided by the host side Cryptoki library to the application cannot be used by the FM because they will not be recognisable by the embedded Cryptoki library. The FM must open its own Cryptoki sessions to make Cryptoki calls that need a session handle. Depending on the system architecture the developer can: > For multi-part commands: implement a custom command to open a session and return the handle to the application where it can be passed in with each subsequent call. > For single part commands: find the slot number of the Application session (use **C_GetSessionInfo**) and translate it to the matching embedded slot **MD_GetEmbeddedSlotId()** then send that slot number to the HSM with your custom command request. The FM can use that slot number to temporarily open a session locally due to the single part command.
Object Attributes	The implementer needs to encode/decode any object attributes that will be passed in or out of the custom command.

CHAPTER 5: FM Samples

There are three sample FMs provided with the Luna FM SDK package:

> "Sample: skeleton" on page 41

> "Sample: pinenc:" on page 42

> "Sample: wrap-comp:" on page 45

NOTE Sample FMs are distributed with the Luna FM SDK package. They have a similar file layout.

Each of the FM samples is structured in a similar way. Each sample directory contains:

makefile	makefile to build host and HSM side code
fm	directory holding HSM side source
host	directory holding host (server) 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/decoder) (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. (For Windows, consider **nmake**.)

> Production build, no debug information in binaries:
make

> Production build, with debug information in binaries and optimization turned off:
make DEBUG=1

Binary files generated by the above variants are placed in different directories. The directory names used are:

obj-ppc	FM Object files
bin-ppc	FM Binary (FM image)

Host Binary files generated by the above variants are placed in different directories. The directory names used are:

output/obj	Host side test application Object files
output/bin	Host side test application executable

The binaries generated from each variant can be deleted using the target 'clean'.

make DEBUG=1 clean

Signing FM Images

The build scripts generate the unsigned FM binary image when the HSM builds are performed. The binary images are named '<samplename>.bin'. Since these images are not signed yet, it is not possible to load them into the HSM. To use the key management scheme (using self-signed FM certificates), follow the steps listed below:

For PCIe HSM

1. Generate the key pair on a user partition/slot. Execute:

   cmu generatekeypair -slot <slot> **-password** <myuserpin> **-label** <fmsign> **-keytype** <rsa> **-sign** <true> **-verify** <true> **-modulusbits** <2048>

   This will generate a 2048 bit RSA key pair. The minimum key size for FM signing should be 2048 bits.

2. To make a self signed certificate, execute:

   cmu selfsigncertificate -slot <slot> **-password** <myuserpin> **-publichandle** <pubkeyhd1> **-privatehandle** <prikeyhd1> **-label** <fmcert> **-cn** <fmcert>

3. Now, the binary image can be signed using mkfm. In the directory where the binary image is generated, execute:

   mkfm -k SLOTID=<slot>**/**<fmsign> **-f**<sampleN.bin> **-osampleN.fm**

   where "<slot>" is the slot id where the signing key is located and <fmsign> 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 HSM partition to be entered.

4. Export the self-signed certificate to a file:

   ```
   /usr/safenet/lunaclient/bin/cmu export -slot <slot> -password <myuserpin> -label <fmcert
   label> -outputfile=<fmcert.cert>
   ```

5. Exit from all cryptoki applications that are still active, and load the FM image into the HSM.

6. Execute this command:

   ctfm i -a <HSM device #> **-f**<fmfilename>.fm **-c** <certificatefile>.cer

   where certificatefile.cer is the name of the certificate in Admin Token used to verify the FM binary image integrity, and device # is the HSM number

   - if you have one Luna PCIe HSM 7 card it is device 0
   - a second HSM card would be device 1, and so on
   - if you want to load the FM on all FM-enabled HSM cards in the system, you can specify –A (as in **ctfm i –A**).

   or, to load the FM and verify it using the public key in the certificate file, AND save the certificate object in the admin partition with the set label:

   ctfm i -a <HSM device #> **-f**<fmfilename>.fm **-c** <certificatefile>.cer **–l**<CertObjectLabel>

   but, if the certificate is already on the HSM slot/partition you can use

   ctfm i -a <HSM device #> **-f**<fmfilename>.fm **-l** <CertObjectLabel>

7. The load operation can be checked by executing the command:

   ctfm q

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

For LLuna Network HSM 7

1. From the Client, generate the key pair on the slot. Execute:

   cmu generatekeypair -slot <slot> **-password** <myuserpin> **-label** <fmsign> **-keytype** <rsa> **-sign** <true> **-verify** <true> **-modulusbits** <2048>

   This will generate a 2048 bit RSA key pair. The minimum key size for FM signing should be 2048 bits.

2. To obtain the handles of the new key objects. Execute:

   cmu list -slot <slot> **-password** <myuserpin> **-handle -class -label** <fmsign>

3. To make a self signed certificate, execute:

   cmu selfsigncertificate -slot <slot> **-password** <myuserpin> **-publichandle** <pubkeyhd1> **-privatehandle** <prikeyhd1> **-label** <fmcert> **-cn** <fmcert>

4. Now, the binary image can be signed using mkfm. In the directory where the binary image is generated, execute:

   mkfm -k SLOTID=<slot>**/**<fmsign> **-f**<sampleN.bin> **-osampleN.fm**

   where “<slot>” is the slot id where the signing key is located and <fmsign> 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 HSM partition to be entered.

5. Export the self-signed certificate to a file:

   ```
   /usr/safenet/lunaclient/bin/cmu export -slot <slot> -password <myuserpin> -label <fmcert
   label> -outputfile=<fmcert.cert>
   ```

6. Copy the exported certificate file to the host of the HSM that is to use the FM, and copy the FM there as well. If the destination is a Luna Network HSM 7, use **pscp** or **scp**:

```
scp <fmcert.cert> admin@<hostname-or-ip-of-appliance>:
```

7. On the Luna Network HSM 7, ensure that policy 51 is set for AutoActivation of Secure Memory File System (SMFS).
8. Exit from all cryptoki applications that are still active, and load the FM image into the HSM.

   Log in via Luna Shell, with **hsm login** and execute:

   hsm fm load -certFile <filename> **-fmFile** <filename>

   The load operation can be checked by executing the command:

   hsm fm status
9. To enable the newly signed FM you must restart the HSM.

   hsm restart

> **NOTE** For Windows environments, set a search path to the libraries.
>
> **Example:** for a default installation, set:
>
> `PATH=C:\Program Files\SafeNet\LunaClient\`
>
> and
>
> `LD_LIBRARY_PATH=C:\Program Files\SafeNet\LunaClient\`
>
> Adjust your path statement if you selected a non-default install path during Luna HSM Client software installation.

Sample: skeleton

This sample FM is very simple and small and can be used as an empty skeleton to start your FM development.

Make a copy of this FM source and modify it for your own needs.

The other sample FMs are large and complex and are intended to illustrate the various programming techniques used in FM development. You can use them as a reference and copy and paste code fragments into your own FM but it is unlikely that you would want to start your own FM development by taking a copy of the **pinenc** or **wrapcomp** FMs.

The ‘e’ FM provides a simple message echo service. The message communication method selected is the random access style. It also opens a session on the Embedded cryptoki to illustrate embedded slot mapping.

skeleton has code samples for the following functionality:

> Registering a random access message handler

> Parsing request messages and performing integer endian conversions

> Constructing and returning a response message

The FM implements one custom command

There is no need to specify a command code as it is implicit.

Description

The skeleton test application is used to exercise the skeleton sample FM.

skeleton Test Application

```
skeleton[-h] [-?] -s<slotnum> -t <text>
```

-s<slotnum>	use slot slotnum - (default 1) e.g. -s3
-t<text>	text to echo

e.g. skeleton -s6 -t “My message”

To access the slot number and determine if the HSM supports FMs:

1. Launch lunacm and execute the command **slot list**

   slot list
2. Record the slot number for the device.
3. Exit lunacm.
4. For Luna PCIe HSM 7, use **ctfm q** command to list available FM-capable HSMs.

   For Luna Network HSM 7, use **hsm fm status** command.

Sample: pinenc:

Demonstrates how custom functionality can be implemented. The only use of the external Cryptoki interface is to login the operator.

The FM provides a simple pin encryption facility. User pins that are encrypted under a RSA public key (perhaps in a Web Browser) can be sent to the HSM to be re-encrypted under a Symmetric Pin Encryption key.

pinenc has code samples for the following functionality:

> Registering a message handler
> Parsing request messages and switching between different command codes
> Using the internal Cryptoki implementation to get services from the Luna Core.
> Using **CT_SetPrivilegeLevel** to override Cryptoki rules
> Using FMCE API to get raw AES and RSA crypto services
> Using the **SMFS** to store sensitive keys.
> Generating Debug trace messages
> Generating Secure Audit entries
> Constructing and returning a response message

The FM implements four custom commands:

PE_CMD_GEN_KEYS:

Description:

Generates an RSA key pair and an AES key and stores them in the **SMFS**

> **NOTE** The FM opens a cryptoki session inside the HSM and relies on that session having the *same* login status as the client process calling the custom command.

Input:

zone, slot_num

Output:

status

Process:

```
C_OpenSession,
C_GenerateKeyPair( 2048 bit RSA key )
CT_SetPrivilegeLevel(1)
C_GetAttributeValue to Read private key attribute
C_Finalize
AES key = FM_GetNDRandom
If cannot open SmFs file then Create SmFs file.
Store RSA and AES key into SmFs file
FM_AddToExt( audit entry )
Return status
```

PE_CMD_GET_PUBKEY:

Description:

Returns the previously generated RSA public key

Input:

zone

Output:

status, encoded pubkey

Process:

```
If ( pub key is not in cache ) open and read SmFs file into cache
Encode rsa pubkey for response
Return status, encoded pubkey
```

PE_CMD_CLR_PIN_ENCRYPT:

Description:

Uses stored RSA public key to encrypt a clear pin block

Input:

zone, clear pinblock

Output:

status, encrypted pinblock

Process:

```
If ( pub key is not in cache ) open and read SmFs file into cache
Use FMCE Api to OAEP encrypt the pinblock
Return status, encrypted pinblock
```

PE_CMD_TRANSLATE_PIN:

Description:

Re-encrypts the pinblock from RSA to AES

Input:

zone, encrypted pinblock

Output:

status, encrypted pinblock

Process:

```
If ( pri key is not in cache ) open and read SmFs file into cache
Use FMCE Api and RSA pri to OAEP decrypt the pinblock
Use FMCE Api and AES key to ECB encrypt the pinblock
Return status, encrypted pinblock
```

pinenc Test Application

```
pinenctest [-z<zone#>] [-s<slot> -p<pin> gen ] | [-d<hsm> test ]
```

-z<key zone number>	Use key zone # - (default 1).e.g. -z123
-s<slot number>	Use slot # - (default 1) e.g. -s3
-d<device number>	Use HSM device # - (default 3) e.g. -d3
-p<pin>	Use pin to log into slot
gen	Perform key generate operation
test	Perform pin translate tests (default)

Description:

The pinencetest application is used to exercise the pinenc sample FM. The FM operates in two modes. Either it is generating a key set or it is using a key set. The pinenctest application allows the user to specify whether to generate a key set (**gen**) or to test a key set (**test**).

When generating a key set, you must determine the Cryptoki slot number on which you want to login and generate a key set. The Test application requires a Cryptoki token to generate key sets. So when you ask the FM to generate keys it needs to know which slot number to use. The test mode uses the keys already generated and requires you to specify only the device number. In order to handle multiple HSM instances you must specify which HSM is to be used for the test. The device number specifies the HSM instance.

To access the slot number and determine if the HSM supports FMs:

1. Launch lunacm and execute the command **slot list**

   slot list
2. Record the slot number for the device.
3. Exit lunacm.
4. For Luna PCIe HSM 7, use **ctfm q** command to list available FM-capable HSMs.

   For Luna Network HSM 7, use **hsm fm status** command.

Process:

```
C_Initialize, Find Admin Token, C_OpenSession, C_Login(Admin Password)
C_GenerateKeyPair( 2048 bit RSA key )
CT_SetPrivilegeLevel(1)
C_GetAttributeValue to Read private key attribute
C_Finalize
AES key = FM_GetNDRandom
If cannot open SmFs file then Create SmFs file.
Store RSA and AES key into SmFs file
FM_AddToExt( audit entry )
Return status
```

Sample: wrap-comp:

Description: This sample demonstrates how to implement an extension to Cryptoki. In this sample a new **C_WrapKey** mechanism is defined.

wrap-comp has code samples for the following functionality:

- Registering a message handler
- Parsing request messages and switching between different commands codes
- Using the internal Cryptoki implementation to get services from the Luna Core.
- Generating Debug trace messages
- Constructing and returning a response message

The FM implements one command:

WC_CMD_GET_RSA_COMP:

Description:

Extracts a specific attribute from a RSA Private key, wrap it with a symmetric key and return the cryptogramme. The schematics of this function are the same as the Cryptoki **C_WrapKey** command.

Input:

Slot_num, hRSAObj, hDESObj, attribute_type

Output:

status, encrypted Component

Process:

```
Call C_OpenSession(slot_num)
Verify that hRSAObj is valid handle to a RSA Private key object with CKA_EXTRACTABLE=1
Verify hDESObj is valid handle to a CKK_DES3 with CKA_WRAP=1
CT_SetPrivilegeLevel(1)
Read selected attribute from hRSAPri object
Use hDESKey to CBC encrypt the component
CT_SetPrivilegeLevel(0)
Return status, encrypted component
```

wrap-comp Test Application

```
wrapcomptest [-sSlot] [-p<pin>]
```

-p<pin>	Specify CKU_USER pin of slot (used for batch mode).
-s#	Use slot # - (default 1) e.g. -s3

Description

The wrapcomptest application is used to exercise the wrapcomp sample FM.

The application logs into the HSM and generates a temporary RSA key pair. It then uses the FM to wrap and (partially display) each component.

The FM uses Cryptoki operations and requires a slot number.

To access the slot number and determine if the HSM supports FMs:

1. Launch lunacm and execute the command **slot list**

   slot list

2. Record the slot number for the device.
3. Exit lunacm.

4. For Luna PCIe HSM 7, use **ctfm q** command to list available FM-capable HSMs.

   For Luna Network HSM 7, use **hsm fm status** command.

CHAPTER 6: Utilities Reference

The section contains information pertaining to the following utilities:

> "cmu" below

> "ctfm" below

> "mkfm" on page 52

> "fmrecover" on page 53

cmu

The CMU utility (Certificate Management Utility) referred to in this manual is provided as a part of the Luna FM SDK package. Refer to the cmu for further information.

ctfm

Functionality Module Management utility.

SYNTAX

```
ctfm d [-a<device> | -A | -s#]-i<fmid> [-p<password> | -e<PED>]
ctfm i [-a<device> | -A | -s#] [-c<certFile>] [-l<certLabel>] -f<fmFile> [-p<password> | -
e<PED>]
ctfm q [-a<device> | -A | -s#]
ctfm v [-a<device> | -A | -s#] [-c<certFile>]  -l<certLabel> -f<fmFile> [-p<password> | -
e<PED>]
ctfm a [-a<device> | -A | -s#] [-n] [-p<password> | -e<PED>]
```

DESCRIPTION

The **ctfm** utility is designed for the HSM administrator and is used to manage functionality modules on the HSMs.

> **NOTE** This tool is for use on a computer that hosts a Luna PCIe HSM 7 locally. For Luna Network HSM 7s, use the LunaSH **hsm fm** commands.

With this tool it is possible to:

> Load a new FM

> Delete an FM

> Query the status of any FMs

> Verify an FM file is correctly signed

> Activate the SMFS on an HSM.

In each case the operation may apply to all HSMs or an individually specified HSM. By default, **ctfm** reports the FM state for all devices found.

The device Security Officer password must be initialized in order for these commands to run. When the commands are executed they might require the SO password of the HSM. When it is required, the utility prompts the operator for the values (unless the values have already been entered during the execution of the same command, or provided by the **-p** option).

Audit trail entries are created when FMs are loaded or disabled. In order to create event logs correctly the HSM real time clock should be initialized.

> **NOTE** To set the real time clock, and enable audit entries, initialize the Auditor Role.

In order to load an FM, a certificate must be present in the Admin Token of the HSM. Usually a PEM-encoded certificate file is provided with the FM image file that you can load to the HSM Admin slot with the **cmu** tool.

If the utility detects that the certificate is not already present in the Admin token, the utility imports the certificate from the cert file.

If no Certificate label is specified then a Certificate file name must be specified and the **ctfm** utility creates a temporary cert object from the file contents.

FM STATES

Each FM in the HSM has a certain state.

Enabled	The FM is in memory and has been started. On HSM restart the FM will return back to the Enabled state.
Zombie	The FM is in memory and has been started. On HSM restart the FM will disappear
Loaded	The FM is not in memory and has not been started. After next HSM restart the FM will go to the Enabled state.
InActive	The FM is loaded but has not been started because it is waiting for the SMFS Activation.
Not started	The FM is loaded but failed to start – this could be due to the HSM FW version being too old for the FM.

COMMANDS

d Disable/Delete FM

```
ctfm d [-a<device> | -A] [-i <fmid>] [-p <password | -e<PED>]
```

This command is used to remove an FM.

The exact behaviour depends on the state of the FM.

Enabled	FM changed to zombie state

Zombie	No action
Loaded	FM will be deleted

> **NOTE** The HSM SO PIN will be required.

i Import FM

ctfm i [-**a**<device> | -**A**] [-**c**<certFile>] [-**p** <password> | -**e**<PED>] [-**l**<certLabel>] -**f**<fmFile>
Load a new FM onto an HSM.

The FM image file contains an FM image and a digital signature. The import operation directs the image and signature to the appropriate Public Key Certificate file which is used to verify the signature. The command looks on the Admin Token of the device for a certificate label equal to the <certLabel> parameter.

If the certificate object is not present then the utility will attempt to create a certificate object from the contents of the certFile i.e. import the certificate

If the <certFile> parameter is not provided the utility will assume the filename is the <certLabel> with .cert or .crt appended. For example, if the certificate label is myfm then the utility will search for a file named myfm.cert and then myfm.crt.

The exact behavior of the command depends on the state of other FMs in the HSM:

- The FM will be installed in the HSM in a Loaded state.
- If a FM with the same ID and in a Loaded state is already in the HSM then the new FM will replace the old FM, and the old FM will enter a Zombie state.
- If a FM with the same ID and in a Enabled/Zombie state is already in the HSM then the old FM changes to or remains in Zombie state.

The exact behavior depends on the state of the HSM:

Enabled	HSM restart will restore all Enabled FMs
Zombie	HSM restart will delete all Zombie FMs
Loaded	HSM restart will enable all Loaded FMs

> **NOTE** The HSM SO PIN will be required.

q Query FM Status

ctfm q [-**a**<device> | -**A**]
Queries the status of an FM (if any) on all or an individual HSM.

Use this command to obtain the name, version information and disable status of an FM or to see if an FM is loaded at all.

> **NOTE** If the FM state is Enabled but the Status shows an error then it might not be possible to communicate with any FM. You should use **ctfm** to delete the failing FM.

No PINs are required to perform this operation.

v Verify an FM Signature

```
ctfm v [-a<device> | -A] [-c<certFile>] [-p <password> | -e<PED>] -l<certLabel> -f<fmFile>
```

This command is used to verify that an FM file has been signed correctly without attempting to load the FM.

The HSM SO PIN is required.

The behaviour of the <certLabel> and <certFile> parameters is the same as is described for the Import FM command above.

a Activate SMFS

```
ctfm a [-a<device> | -A] [-p <password> | -e<PED>] [-n]
```

This command activates the Secure Memory File system.

Either the HSM SO or Administrator user can activate the SMFS. Default is SO. Specify administrator with "-n".

OPTIONS

The following options are supported:

Parameter	Shortcut	Description
--device-number=<device>	**-a**	Use the admin token on the specified device. The first device is numbered 0. If this parameter is absent then the first device is implied. This option is useful in case of more than one HSM in the host.
--all-devices	**-A**	Apply command to all available devices.
--slot=<slotNum>	**-s**	Apply command to Device with Cryptoki slot number.
--fmid=<id>	**-i**	Specify ID (in HEX) of FM (default 0x100).
--fm-cert-file=<name>	**-c**	FM validation certificate filename.
--fm-file=<name>	**-f**	Name of file holding a new FM.
-help	**-h, -?**	Display usage information.
--fm-cert-label=<name>	**-l**	FM validation certificate object label.
--password=<password>	**-p**	SO or Admin password – if this option is not included in the command, then **ctfm** may prompt for the password. Not valid where a PED prompt is required.

Parameter	Shortcut	Description
--administrator	**-n**	Use Administrator role instead of SO (SMFS activation only).
--no-banner	**-b**	Do not show program banner during startup.
--ped=<PED>	**-e**	Remote PED ID. Default is 0 (zero). Check lunacm to find the value (usually 100) to insert here.

mkfm

Synopsis

```
mkfm -f <filename> -k <key> -o <filename> [-c][-b][-e <PED> | -p <password>] [-u <user>]
```

Description

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

> **NOTE** At time of initial release for use with Luna HSMs, MKFM supports only RSA private keys that reside on a Luna token. The signing mechanism uses RSA-SHA512.

Options

The following options are supported:

Parameter	Shortcut	Description
--input-file=<filename>	**-f**<filename>	Specifies the relative, or full, path to the FM binary image.
--signer-key=<key>	**-k**<key>	This is the name of the private key that is going to be used to sign the FM image. The format of the key is <TokenName (PIN) /KeyName>, or <TokenName/KeyName>. TokenName is the label of the token or you can use SLOTID=x, where x is the slot id number.
--output-file=<filename>	**-o**<filename>	This specifies the relative or full path to the loadable FM image.
--password=<Password>	**-p**<password>	Optional parameter to specify user password when performing ctfm operations in batch mode.

Parameter	Shortcut	Description
--user=<user>	**-u** <user>	Optional parameter to specify which user role to login as default CO User : slot user role name. Default is USER Role: • 'ad' on Admin partition • 'co' on User partition • 'cu' (specify slot number in key spec and use -u? to get a list)
--no-banner	**-b**	Do not show program banner during startup
--ped=<PED>	**-e**<PED>	Remote PED ID. Default is 0 (zero). Check lunacm to find the value (usually 100) to insert here.
--compat	**-c**	Compatibility mode – inhibit the use of Luna custom extension functions that would stop the tool from working with a standard Cryptoki provider. If the tool displays error messages referring to missing functions then these may be suppressed by adding FunctionBindLevel=2 to the misc section of /etc/Chrystoki.conf

> **NOTE** The long forms require two leading dashes for each option. The short forms take a single leading dash, and an optional space.

fmrecover

Synopsis:

```
fmrecover [--smfs] [--fm] <path>
```

--smfs	Erase the SMFS
--fm	Erase all FMs
<path>	Path to HSM device node. An example path is: **/dev/k7pf0**

Description:

The FM recovery utility is used to force either all the FMs and/or the entire Secure Memory in the event that the HSM has been rendered non-responsive due to a badly configured FM being loaded.

> **NOTE** This tool is for use on a computer that hosts a Luna PCIe HSM 7 locally. For Luna Network HSM 7s, use the LunaSH **hsm fm** commands.

Examples:

To delete all FMs execute:

./fmrecover --fm /dev/k7pf0

To reformat SMFS execute:

./fmrecover --smfs /dev/k7pf0

To delete all FMs and reformat SMFS execute:

./fmrecover --fm --smfs /dev/k7pf0

The FM part of the command can be tested by loading an FM and then running **./fmrecover --fm /dev/k7pf0**. Ensure that the FM is deleted.

The SMFS part of the command can be tested by running **/fmrecover --smfs /dev/k7pf0**. Ensure that the message "Formatting SMFS..." appears in the dmesg log.

 Copyright 2001-2024 Thales Group

CHAPTER 7: Cryptographic Engine

The Cryptoki interface provides a standardized way of performing cryptographic operations. However a considerable amount of overhead is introduced.

Luna FM SDK package provides internal APIs that bypass the PKCS #11 subsystem to provide high performance cryptographic functionality. This chapter describes the functions in this API. It contains the following functions:

Parameters

The Crypto Engine API uses PKCS#11 parameters for all parameters except for keys.

For a full list of supported mechanisms refer to Supported Mechanisms.

The <FMCE_KEY_PTR> parameter is a pointer to a union of all supported key types. For more details and a list of supported key types refer to the **fmcrypto.h** header file.

Single part sign operation

```
CK_RV FMCE_Sign(
               CK_MECHANISM_PTR pMech,  // IN: mechanism type and parameters
               FMCE_KEY_PTR     pKey,   // IN: key value

               CK_ULONG         ulDataInLen,  // len of data to sign
               CK_BYTE_PTR      pDataIn,      // IN: data to sign
               CK_ULONG_PTR     pulSigOutLen, // IN: len of sig buf OUT: len of signature
               CK_BYTE_PTR      pSigOut       // OUT: signature
                      );
```

Single part verify operation

```
extern CK_RV FMCE_Verify(
               CK_MECHANISM_PTR pMech,  // IN: mechanism type and parameters
               FMCE_KEY_PTR     pKey,   // IN: key value

               CK_ULONG         ulDataInLen,  // len of signed data
               CK_BYTE_PTR      pDataIn,      // IN: signed data
               CK_ULONG         ulSigLen,     // len of signature
               CK_BYTE_PTR      pSig);        // IN: signature
```

Single part encrypt operation

```
CK_RV FMCE_Encrypt(
               CK_MECHANISM_PTR pMech,  // mechanism type and parameters
               FMCE_KEY_PTR     pKey,   // key value

               CK_ULONG         ulDataInLen,
               CK_BYTE_PTR      pDataIn,
               CK_ULONG_PTR     pulOutLen,
               CK_BYTE_PTR      pOut);
```

Single part decrypt operation

```
CK_RV FMCE_Decrypt(
               CK_MECHANISM_PTR pMech,  // mechanism type and parameters
               FMCE_KEY_PTR     pKey,   // key value

               CK_ULONG         ulDataInLen,
               CK_BYTE_PTR      pDataIn,
               CK_ULONG_PTR     pulOutLen,
               CK_BYTE_PTR      pOut);
```

Single part digest operation

```
CK_RV FMCE_Digest(
               CK_MECHANISM_PTR pMech,  // IN: mechanism type and parameters

               CK_ULONG         ulDataInLen,  // len of data to digest
               CK_BYTE_PTR      pDataIn,      // IN: data to digest
```

```
                CK_ULONG_PTR     pulDigOutLen, // IN: len of digest uffer OUT: len of digest
                CK_BYTE_PTR      pDigOut       // OUT: digest
                         );
```

Multi-part sign operation init

```
CK_RV FMCE_SignInit(
                CK_MECHANISM_PTR pMech,  // IN: mechanism type and parameters
                FMCE_KEY_PTR     pKey,   // IN: key value

                FMCE_HANDLE_PTR  pCtxHdl // OUT: handle for following calls
                          );
```

Multi-part encrypt operation init

```
CK_RV FMCE_EncInit(
                CK_MECHANISM_PTR pMech,  // mechanism type and parameters
                FMCE_KEY_PTR     pKey,   // key value

                FMCE_HANDLE_PTR  pCtxHdl // OUT: handle for following calls
                          );
```

Multi-part verify operation init

```
CK_RV FMCE_VerifyInit(
                CK_MECHANISM_PTR pMech,  // mechanism type and parameters
                FMCE_KEY_PTR     pKey,   // key value

                FMCE_HANDLE_PTR  pCtxHdl // OUT: handle for following calls
                          );
```

Multi-part decrypt operation init

```
CK_RV FMCE_DecInit(
                CK_MECHANISM_PTR pMech,  // mechanism type and parameters
                FMCE_KEY_PTR     pKey,   // key value

                FMCE_HANDLE_PTR  pCtxHdl // OUT: handle for following calls
                          );
```

Multi-part digest operation init

```
CK_RV FMCE_DigestInit(
                CK_MECHANISM_PTR pMech,  // mechanism type and parameters

                FMCE_HANDLE_PTR  pCtxHdl // OUT: handle for following calls
                          );
```

Multi-part sign operation update

```
CK_RV FMCE_SignUpdate(
                FMCE_HANDLE     ulCtxHdl,    // handle from init operation
                CK_ULONG        ulDataInLen, // len of data to sign
                CK_BYTE_PTR     pDataIn      // IN: data to sign
                             );
```

Multi-part verify operation update

```
CK_RV FMCE_VerifyUpdate(
               FMCE_HANDLE     ulCtxHdl,    // handle from init operation
               CK_ULONG        ulDataInLen, // len of signed data
               CK_BYTE_PTR     pDataIn      // IN: signed data
                              );
```

Multi-part decrypt operation update

```
CK_RV FMCE_DecUpdate(
               FMCE_HANDLE     ulCtxHdl,    // handle from init operation
               CK_ULONG        ulDataInLen, // len of cipher data
               CK_BYTE_PTR     pDataIn,     // IN: cipher data
               CK_ULONG_PTR    pulDataOutLen, // IN: o/p data len OUT: len clear text
               CK_BYTE_PTR     pDataOut     // OUT: clear text
                          );
```

Multi-part encrypt operation update

```
CK_RV FMCE_EncUpdate(
               FMCE_HANDLE     ulCtxHdl,     // handle from init operation
               CK_ULONG        ulDataInLen,  // len clear text
               CK_BYTE_PTR     pDataIn,      // IN: clear text
               CK_ULONG_PTR    pulDataOutLen,// IN: len cipher text buffer OUT: len cipher text
               CK_BYTE_PTR     pDataOut      // OUT: cipher text
                           );
```

Multi-part digest operation update

```
CK_RV FMCE_DigestUpdate(
               FMCE_HANDLE     ulCtxHdl,     // handle from init operation
               CK_ULONG        ulDataInLen,  // len of data
               CK_BYTE_PTR     pDataIn       // IN: data to digest
                              );
```

Multi-part sign operation final

```
CK_RV FMCE_SignFinal(
               FMCE_HANDLE     ulCtxHdl,     // handle from init operation
               CK_ULONG        ulMacLen,     // len (in bytes) of MAC (only for MAC operations)
else ignored
               CK_ULONG_PTR    pulSigOutLen, // IN: len of signature buffer OUT: len of
signature
               CK_BYTE_PTR     pSigOut       // OUT: signature
                          );
```

Multi-part verify operation final

```
CK_RV FMCE_VerifyFinal(
               FMCE_HANDLE     ulCtxHdl,     // handle from init operation
               CK_ULONG        ulSigInLen,   // len of signature
               CK_BYTE_PTR     pSigIn        // IN: signature
                             );
```

Multi-part decrypt operation final

```
CK_RV FMCE_DecFinal(
               FMCE_HANDLE     ulCtxHdl,     // handle from init operation
               CK_ULONG_PTR    pulDataOutLen,// IN: len of clear text buffer OUT: len of clear
text
               CK_BYTE_PTR     pDataOut      // OUT: clear text
                          );
```

Multi-part encrypt operation final

```
CK_RV FMCE_EncFinal(
               FMCE_HANDLE     ulCtxHdl,     // handle from init operation
               CK_ULONG_PTR    pulDataOutLen,// IN: len of buffer OUT: len cipher text
               CK_BYTE_PTR     pDataOut      // OUT: cipher text
                         );
```

Multi-part digest operation final

```
CK_RV FMCE_DigestFinal(
               FMCE_HANDLE     ulCtxHdl,     // handle from init operation
               CK_ULONG_PTR    pulDigOutLen, // IN: len of buffer OUT: len of digest
               CK_BYTE_PTR     pDigOut       // OUT: digest
```

Supported Mechanisms

As of Luna HSM Firmware 7.7.0, the FM cryptographic engine supports all mechanisms that the host Thales Cryptoki library accepts for the following PKCS#11 functions:

- C_EncryptInit()
- C_DecryptInit()
- C_SignInit()
- C_VerifyINit()
- C_DigestInit()

For an exhaustive list of supported mechanisms, refer to Firmware 7.7.0 Mechanisms.

CHAPTER 8: Cipher Objects

Some limited legacy cipher objects are provided in the Luna FM SDK package to assist developers porting FM designs to the Luna HSM.

Supported Cipher Objects

A subset of the ProtectToolkit cipher objects and modes are supported in Luna:

Cipher Object	Mode
FMCO_IDX_AES	ECB, CBC, MAC_3, MAC_GEN
FMCO_IDX_DES	ECB, CBC, MAC_3, MAC_GEN
FMCO_IDX_TRIPLEDES	ECB, CBC, MAC_3, MAC_GEN
FMCO_IDX_DSA	0 (Sign/Verify)
FMCO_IDX_RSA (Sign/Verify)	RSA_MODE_PKCS
FMCO_IDX_RSA (Enc/Dec)	RSA_MODE_X509, RSA_MODE_PKCS, RSA_MODE_OAEP
FMCO_IDX_RSA_MD5	0
FMCO_IDX_RSA_SHA1	0
FMCO_IDX_RSA_SHA224	0
FMCO_IDX_RSA_SHA256	0
FMCO_IDX_RSA_SHA384	0
FMCO_IDX_RSA_SHA512	0
FMCO_IDX_HMACMD5	0
FMCO_IDX_HMACSHA1	0
FMCO_IDX_HMACRMD160	0

Cipher Object	Mode
FMCO_IDX_CAST	ECB, CBC
FMCO_IDX_RC2	ECB, CBC, MAC_3, MAC_GEN

Mechanisms Supported by Cipher Objects

As of Luna HSM Firmware 7.7.0, FM cipher objects support the following limited set of mechanisms:

> CKM_AES_ECB
> CKM_AES_CBC
> CKM_AES_CBC_PAD
> CKM_AES_MAC
> CKM_AES_MAC_GENERAL
> CKM_DES_ECB
> CKM_DES_CBC
> CKM_DES_CBC_PAD
> CKM_DES_MAC
> CKM_DES_MAC_GENERAL
> CKM_DES3_ECB
> CKM_DES3_CBC
> CKM_DES3_CBC_PAD
> CKM_DES3_MAC
> CKM_DES3_MAC_GENERAL
> CKM_RC2_ECB
> CKM_RC2_CBC
> CKM_RC2_CBC_PAD
> CKM_RC2_MAC
> CKM_RC2_MAC_GENERAL
> CKM_MD5_HMAC
> CKM_MD5_HMAC_GENERAL
> CKM_SHA_1_HMAC
> CKM_SHA_1_HMAC_GENERAL
> CKM_RSA_X_509
> CKM_RSA_PKCS
> CKM_RSA_PKCS_OAEP

 Copyright 2001-2024 Thales Group

- CKM_SHA1_RSA_PKCS
- CKM_SHA224_RSA_PKCS
- CKM_SHA256_RSA_PKCS
- CKM_SHA384_RSA_PKCS
- CKM_SHA512_RSA_PKCS
- CKM_DSA
- CKM_SHA_1
- CKM_SHA224
- CKM_SHA256
- CKM_SHA384
- CKM_SHA512

 Copyright 2001-2024 Thales Group

CHAPTER 9: Hash Objects

Some limited legacy hash objects are provided in the Luna FM SDK package to assist developers porting FM designs to the Luna HSM.

Supported Hash Objects

A subset of the ProtectToolkit hash objects are supported in Luna:

- FMCO_IDX_MD2
- FMCO_IDX_MD5
- FMCO_IDX_RMD160
- FMCO_IDX_SHA1
- FMCO_IDX_SHA224
- FMCO_IDX_SHA256
- FMCO_IDX_SHA384
- FMCO_IDX_SHA512

Mechanisms Supported by Hash Objects

As of Luna HSM Firmware 7.7.0, FM hash objects support the following limited set of mechanisms:

- CKM_AES_ECB
- CKM_AES_CBC
- CKM_AES_CBC_PAD
- CKM_AES_MAC
- CKM_AES_MAC_GENERAL
- CKM_DES_ECB
- CKM_DES_CBC
- CKM_DES_CBC_PAD
- CKM_DES_MAC
- CKM_DES_MAC_GENERAL
- CKM_DES3_ECB
- CKM_DES3_CBC
- CKM_DES3_CBC_PAD

 Copyright 2001-2024 Thales Group

- CKM_DES3_MAC
- CKM_DES3_MAC_GENERAL
- CKM_RC2_ECB
- CKM_RC2_CBC
- CKM_RC2_CBC_PAD
- CKM_RC2_MAC
- CKM_RC2_MAC_GENERAL
- CKM_MD5_HMAC
- CKM_MD5_HMAC_GENERAL
- CKM_SHA_1_HMAC
- CKM_SHA_1_HMAC_GENERAL
- CKM_RSA_X_509
- CKM_RSA_PKCS
- CKM_RSA_PKCS_OAEP
- CKM_SHA1_RSA_PKCS
- CKM_SHA224_RSA_PKCS
- CKM_SHA256_RSA_PKCS
- CKM_SHA384_RSA_PKCS
- CKM_SHA512_RSA_PKCS
- CKM_DSA
- CKM_SHA_1
- CKM_SHA224
- CKM_SHA256
- CKM_SHA384
- CKM_SHA512

 Copyright 2001-2024 Thales Group

CHAPTER 10: Setting Privilege Level

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

The **CT_SetPrivilege** command is only available to FMs – it cannot be called from outside the HSM.

SetPrivilegeLevel

Synopsis

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

Description

This function is a Luna 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 cannot be called from outside the HSM (only from inside 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.

The HSM destructive policy HSM_CONFIG_ALLOW_DISABLING_FM_PRIVILEGE_LEVEL may be set to disable the use of the CL_SetPrivilegeLevel().

PRIVILEGE_OVERRIDE mode allows the FM to do the following:

> Read Sensitive attributes

> Perform Cryptographic Initialization calls that contradict the usage attributes. That is, you can call **C_EncryptInit** with an object that has CKA_ENCRYPT set to FALSE.

> Use **C_CreateObject()** to create secret keys and private keys (CKO_SECRET_KEY and CKO_PRIVATE_KEY).

> Use **C_SetAttributeValue()** to change an attribute of an object when CKA_MODIFIABLE=false. Applies only to attributes that could be changed when the CKA_MODIFIABLE is true.

> Create objects as a Crypto Officer while only logged on as a Crypto User role.

Arguments

level - desired privilege.

CHAPTER 11: SMFS Reference

SMFS is a Secure Memory File System (as exported to FMs). It allows FMs to store keys and objects in the HSM's Flash memory. Objects are always encrypted by an HSM-controled key, *before* being stored in Flash. It becomes unrecoverable upon tampering of the HSM, when HSM Policy (40) Decommission on Tamper is enabled.

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

This chapter contains the following sections:

Important Constants

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

Error Codes

SMFS_ERR_ERROR	A general error has occurred
SMFS_ERR_NOT_INITED	The SMFS has not been initialized
SMFS_ERR_MEMORY	The SMFS has run out of memory
SMFS_ERR_NAME_TOO_LONG	The name given for a file is too long
SMFS_ERR_RESOURCES	The SMFS has run out of resources

SMFS_ERR_PARAMETER	An invalid parameter was passed to SMFS
SMFS_ERR_ACCESS	User does not have request access to file
SMFS_ERR_NOT_FOUND	Requested file was not found
SMFS_ERR_BUSY	Operation is being attempted on an open file
SMFS_ERR_EXIST	A file being created already exists
SMFS_ERR_FILE_TYPE	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

The **SMFS** reference section contains the following functions:

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 in this directory.

Return Value

Returns 0 or an error condition.

 Copyright 2001-2024 Thales Group

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.

 Copyright 2001-2024 Thales Group

SmFsCalcFree

Synopsis

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

Return Value

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

 Copyright 2001-2024 Thales Group

SmFsCreateFile

Synopsis

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

Description

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

> **NOTE** These files are encrypted. If the HSM experiences a tamper event it will respond by erasing the encryption key. This means that the confidentiality of the file contents is protected by both Tamper Resistance and Tamper Response security Features.

Parameters

name	Pointer to the absolute path of file to create.
len	Size of file to create (in bytes).

Return Value

Returns 0 or an error condition.

SmFsCreateFileClr

Synopsis

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

Description

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

> **NOTE** Clear files are not encrypted. This means that they are faster, compared to encrypted files, when reading and especially, writing.

The confidentiality of the file contents is protected by Tamper Resistance only.

Clear files are suitable for logs.

The advantage of these files is they are quicker to update than an encrypted file.

 Copyright 2001-2024 Thales Group

Parameters

name	Pointer to the absolute path of file to create.
len	Size of file to create (in bytes).

Return Value

Returns 0 or an error condition.

 Copyright 2001-2024 Thales Group

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.

 Copyright 2001-2024 Thales Group

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.

 Copyright 2001-2024 Thales Group

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.

 Copyright 2001-2024 Thales Group

SmFsFindFileInit

Synopsis

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

Description

Creates a file iteration context.

Wild card parameters include:

?	match any character
*	match many characters

Parameters

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

Return Value

Returns 0 or an error condition.

 Copyright 2001-2024 Thales Group

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.

 Copyright 2001-2024 Thales Group

SmFsGetOpenFileAttr

Synopsis

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

Description

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

Parameters

fh	Pointer to the file handle.
a	Pointer to the returned attributes structure.

Return Value

Returns 0 or an error condition.

 Copyright 2001-2024 Thales Group

SmFsOpenFile

Synopsis

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

Description

Finds the file and creates an 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.

 Copyright 2001-2024 Thales Group

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.

 Copyright 2001-2024 Thales Group

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.

 Copyright 2001-2024 Thales Group

SmFsWriteFile

Synopsis

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

Description

Writes 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.

 Copyright 2001-2024 Thales Group

CHAPTER 12: FMDebug Reference

FM SDK provides trace functions to FM writers. Debug information is readable via the **dmesg** utility on the host. On Linux, these debug messages are also written to **/var/log/messages**.

Function Descriptions

This section contains the following function descriptions:

> "printf/vprintf" on the next page

> "debug_MACRO" on the next page

> "dump" on the next page

printf/vprintf

FM SDK supports the C standard **printf()** and **vprintf()** functions. These functions can be called at any time and accept all standard C99 formatting specifiers.

In FMs, these functions do not print to **stdout**, but instead send log messages to the syslog system.

Since these are formatting messages for a log rather than **stdout**, there are two differences from the standard C implementations:

> Each line of output from **printf()/vprintf()** is prefixed with a log header that includes the FM's ID.

> Each call to **printf** will create a new log entry whether there is a terminating newline character on the end of the string or not.

debug_MACRO

The debug macro is defined in **fmdebug.h**.

Developers may use it to enable or disable code at build time. The debug macro will wrap a piece of code and either enable it or hide it depending on DEBUG=1 being defined during compile.

For example:

```
debug(printf("%s: Check pri key is RSA\n", _func_)
```

dump

Synopsis

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

Description

This function dumps the hex values of each byte in a buffer to the HSM log stream using **printf()**.

It is intended as a convenience function for FM developers debugging their FM code.

Parameters

Parameter	Description
desc	Describes the buffer being displayed. This string is printed immediately before the buffer.
data	This is a pointer to the buffer to be displayed.
len	The length of the buffer to be displayed.

CHAPTER 13: Message Dispatch API Reference

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. The following functions labeled by the **MD_** prefix form the Message Dispatch (MD) API. The function prototypes are defined in the header file **md.h**. The library **ethsm** provides the following functions:

MD_Initialize

Initializes the message dispatch library. Until this function is called, all other functions will return error code MDR_NOT_INITIALIZED.

The MD_Initialize function is not re-enterant. Do not call this function twice at the same time from two different threads.

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 the following codes:

Function Code	Qualification
MDR_OK	No error
MDR_UNSUCCESSFUL	Error in operation. Operation failed.

 Copyright 2001-2024 Thales Group

MD_Finalize

Finalizes 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.

The MD_Finalize function is not re-enterant. Do not call this function twice at the same time from two different threads.

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

Retrieves the number of accessible HSMs at the time the message dispatch library was initialized - 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	No error
MDR_INVALID_PARAMETER	If pHSMCount was NULL
MDR_NOT_INITIALIZE	The message dispatch library was not previously initialized successfully

 Copyright 2001-2024 Thales Group

MD_GetHsmState

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

Parameter	Description
hsmIndex	Zero based index of the HSM to query. 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	Not used: always return zero. The pointer may be NULL.

Output Requirements

Parameter	Description
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_NORMAL_OPERATION	0x8000	The HSM is operational.
S_HSM_DISCONNECTED	2	No HSM detected.
S_HSM_ERASED	1	HSM Decommissioned.
S_TAMPER_RESPOND	3	HSM is in Tampered State.

> **NOTE** Any other value indicates a non-operational HSM

pErrorCode	Error code can be ignored.

The function returns the following codes:

Function Code	Qualification
MDR_OK	No error
MDR_UNSUCCESSFUL	Error in operation. Operation failed.
MDR_NOT_INITIALIZE	The message dispatch library was not previously initialized successfully
MDR_INVALID_HSM_INDEX	HSM index was not in the range of accessible HSMs

 Copyright 2001-2024 Thales Group

MD_ResetHsm

Resets 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 state is not S_NORMAL_OPERATION.

Input Parameters

hsmIndex	Zero based index of the HSM to query. For remote HSMs (such as the Luna Network HSM), the HSM indices are numbered according to the order that the HSM's IP addresses were entered in the ET_HSM_NETCLIENT_SERVERLIST registry key.

> **NOTE** When **MD_Initialize**() is invoked the message dispatch library assigns an index to each available HSM.

Refer to "MD_GetHsmState" on page 90 for further details.

Output Requirements

The function returns the following codes:

Function Code	Qualification
MDR_OK	No error
MDR_UNSUCCESSFUL	Error in operation. Operation failed.
MDR_NOT_INITIALIZE	The message dispatch library was not previously initialized successfully
MDR_INVALID_HSM_INDEX	HSM index was not in the range of accessible HSMs

MD_SendRecieve

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

Parameter	Description
hsmIndex	Zero based index of the HSM to query. For remote HSMs (such as the Luna Network HSM), HSMs are numbered according to the order that the HSM's 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.
pRef	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:javahasmstate	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;
```

```
unit32length;
} 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.

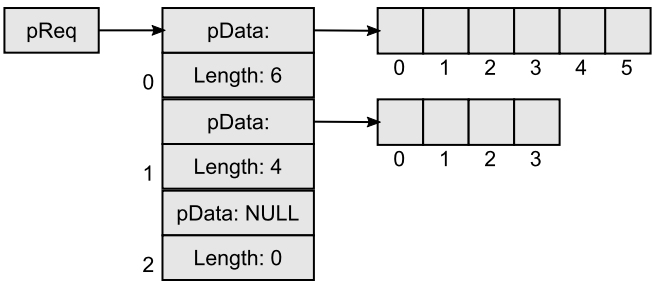


Figure 3: An example of a request buffers data type for function MD_SendReceive

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.
pRecievedLen	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	No error.
MDR_ UNSUCCESSFUL	Error in operation. Operation failed.
MDR_INVALID_ PARAMETER	The pointer supplied for pReq is NULL. The request requires a single response and the pointer supplied for pResp os NULL, or pReserved is not zero.
MDR_NOT_ INITIALIZE	The message dispatch library was not previously initialized successfully.
MDR_INVALID_ HSM_INDEX	HSM index was not in the range of accessible HSMs.
MDR_ INSUFFICIENT_ RESOURCE	There is an insufficient memory on either the host or adapter.
MDR_OPERATION_ CANCELLED	The operation was cancelled in the HSM. This code will not be returned.
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.
MDR_FM_NOT_ AVAILABLE	An invalid FM number was used.

 Copyright 2001-2024 Thales Group

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

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

Parameter	Meaning
MDP_MAX_BUFFER_LENGTH	The recommended maximum buffer size, in number of bytes, for messages that can be sent using the MD library. While messages larger than this buffer size can 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	Meaning
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	No error.

Function Code	Qualification
MDR_ UNSUCCESSFUL	The pointer supplied for pReq is NULL. The request requires a single response and the pointer supplied for pResp os NULL, or pReserved is not zero.

MD_GetEmbeddedSlotID

Synopsis:

```
#include <md.h>
MD_RV MD_GetEmbeddedSlotID(CK_SLOT_ID hostP11SlotId,
                           uint32_t *pHsmIndex);
```

Input Requirements

None

Input Parameters

hostP11SlotID	Host side slot ID of a P11 slot.
pHsmIndex	Pointer to where embedded slot number is returned.

Output Requirements:

The function returns the following codes:

Function Code	Qualification
MDR_OK	No error
MDR_NO_EMBEDDED_SLOT	Indicates the host slot does not have its peer in the HSM
Any other MD_RV error code	Error in operation. Operation failed.

MD_FmIdFromName

Synopsis

```
#include <md.h>
MD_RV MD_GetFmIdFromName (uint32_t hsmIndex,
                          char *pName,
                          uint32_t len,
                          uint32_t *pFMID);
```

For HSMs with FMs enabled this function finds the FMID value for a FM based on the FM name.

Input Requirements

None

Input Parameters

hsmIndex	Zero based index of the HSM to query.
pName	FM name.
len	Length of FM name
pFMID	Pointer to where to store the resulting FMID.

Output Requirements

The function returns the following codes:

Function Code	Qualification
MDR_OK	No error
MDR_UNSUCCESSFUL	If an FM not found
Any other MD_RV error code to indicate error condition.	

MD_GetHsmInfo

Fetch information about an HSM .

Synopsis

```
#include <md.h>
MD_RV MD_GetHsmInfo(uint32 hsmIndex,
                    MD_Info_t infotype,
                    void *pValue,
                    uint32 valueLen);
```

Input Requirements

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

Input Parameters

hsmIndex	Which HSM to query. When looping through the HSM count to search for the HSM having the desired MDI_HSM_LABEL using MD_GetHsmInfo, the hsmIndex starts from 0 and goes to HSMCount-1. See also Configuration File Summary.
infotype	Which value to query.
pValue	Where to store a null-terminated string.
valueLen	Size of buffer pointed at by pValue.

Output Requirements

Information Types

Each type of information is returned as a null terminated string.

Result is always NULL-terminated and might be truncated.

Type	Description
MDI_HSM_DESCRIPTION	Luna HSM Description.
MDI_HSM_MODEL	HSM Model Name.
MDI_HSM_LABEL	HSM Label.
MDI_HSM_SERIAL_NUMBER	Serial Number as a decimal string.
MDI_HSM_FIRMWARE_VERSION	HSM FW Version “mm.nn.ss”
MDI_HSM_CONNECTED	Returns no value. This may be used to query if HSM is connected.

Function Codes

Function Code	Qualification
MDR_NOT_INITIALIZED	The message dispatch library was not previously initialized successfully.
MDR_INVALID_HSM_INDEX	HSM index was not in the range of accessible HSMs.
MDR_HSM_NOT_AVAILABLE	HSM disconnected.

MD_GetHsmIndexForSlot

Synopsis

```
#include <md.h>
MD_RV MD_GetHsmIndexForSlot(CK_SLOT_ID hostP11SlotId,
                            uint32_t *pHsmIndex);
```

For HSMs with FMs enabled, this function translates host PKCS#11 slot ID to the HSM index. Using this function, FM developers can direct FM custom commands to a respective HSM. This function should be used by the host ethsm, only.

Input Requirements

None

Input Parameters

hostP11SlotId	Host side slot ID of a PKCS#11 slot.
pHsmIndex	Pointer to where embedded slot number is returned.

Output Requirements

The function returns the following codes:

Function Code	Qualification
MDR_OK	For successful execution.
MDR_HSM_NOT_AVAILABLE	If a host slot does not have its peer in any HSM with FMs enabled.
Any other MD_RV error code to indicate error condition.	

 Copyright 2001-2024 Thales Group

CHAPTER 14: HSM Functions Reference

The Luna FM SDK package 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.

The HSM Functions section contains reference material for the following functions:

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 redirected 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	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,, realloc, strtol, strtoll, strtoull, stroul
string.h	memchr, memcmp, memcpy, memmove, memset, strcat, strchr, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strncpy, strrchr, strspn, strstr, strerror, strtok
time.h	asctime, clock, ctime, gmtime, localtime, mktime, strftime, time, difftime, clock, gettime

Unsupported Standard C APIs

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

- **comple.h**
- **fenv.h**
- **float.h**
- **locale.h**
- **math.h**
- **signal.h**
- **tgmath.h**
- **wchar.h**
- **wctype.h**

Request/Reply Message management functions

The following functions are provided by **libfmsupt.a**:

- "SVC_GetReplyBuffer" on page 114
- "SVC_ConvertReqToReply" on page 115
- "SVC_SendReply" on page 116
- "SVC_ResizeReplyBuffer" on page 117
- "SVC_DiscardReplyBuffer" on page 118
- "SVC_GetUserReplyBuffLen" on page 119
- "SVC_GetRequest" on page 120
- "SVC_GetRequestLength" on page 121
- "SVC_GetReply" on page 122
- "SVC_GetReplyLength" on page 123

High Precision Timers

- "THR_UpdateTiming" on page 148

> "THR_BeginTiming" on page 147

Register Functionality module Custom handler function

The following functions are provided by **libfmsupt.a** :

> "FMSW_RegisterRandomDispatch" on page 125
> "FMSW_RegisterStreamDispatch" on page 126

Serial communication functions

The following functions are provided by **libfmsupt.a** :

> "SERIAL_GetNumPorts" on page 139
> "SERIAL_Open" on page 144
> "SERIAL_Close" on page 145
> "SERIAL_InitPort" on page 140
> "SERIAL_SendData" on page 135
> "SERIAL_WaitReply" on page 137
> "SERIAL_RecieveData" on page 136
> "SERIAL_FlushRX" on page 138
> "SERIAL_GetControlLines" on page 141
> "SERIAL_SetControlLines" on page 142
> "SERIAL_SetMode" on page 143

USB Access functions

NOTE There is no direct USB support at this time.

Support Functions

> "FM_GetNDRandom" on page 129
> "FM_AddToExtLog" on page 130
> "FM_GetCurrentAppId" on page 150
> "FM_SetCurrentAppId" on page 151
> "FM_GetHsmInfo" on page 130

Message Streaming Functions

This section contains descriptions of functions that the FM can use to access the contents of the request message and to build a corresponding reply message.

Each command that is sent to the FM has a token associated with it that links to the request and reply buffers.

These functions read the request and write the response using a stream model. This scheme can improve performance by providing lower latency compared to a scheme that waits until all the request is available before starting processing.

The Message Streaming section contains the following functions:

SVC_IO_Read

Reads up to the 'size' bytes to the user destination buffer from the I/O input buffer (also known as the "command" buffer). Returns the size actually read (if the end of the data is reached the returned size can be smaller than the requested one).

Synopsis

```
unsigned int SVC_IO_Read(FmMsgHandle token,
                         void *destination,
                         int size);
```

Syntax options

SVC_IO_Read8

SVC_IO_Read16

SVC_IO_Read32

These functions are like **SVC_IO_Read()** except that the size of the data is assumed by the input data type and endian conversion is performed.

The implementation may be able to make these functions faster than the generic **SVC_IO_Read()**.

```
unsigned int SVC_IO_Read8(FmMsgHandle token,
                          uint8_t *v);
unsigned int SVC_IO_Read16(FmMsgHandle token,
                           uint16_t *v);
unsigned int SVC_IO_Read32(FmMsgHandle token,
                           uint32_t *v);
```

 Copyright 2001-2024 Thales Group

SVC_IO_Write

Writes up to the size bytes from the user source buffer to the I/O output buffer (a.k.a. "reply" or "response" buffer). Returns the size actually written (if the capacity of the output buffer is reached the returned size can be smaller than the requested size).

Synopsis

```
unsigned int SVC_IO_Write(FmMsgHandle token,
                          void *source,
                          int size);
```

Syntax options

SVC_IO_Write8

SVC_IO_Write16

SVC_IO_Write32

Similar to **SVC_IO_Read8/16/32()**, but for writes to the reply buffer.

```
unsigned int SVC_IO_Write8(FmMsgHandle token,
                           uint8_t v);
unsigned int SVC_IO_Write16(FmMsgHandle token,
                            uint16_t v);
unsigned int SVC_IO_Write32(FmMsgHandle token,
                            uint32_t v);
```

 Copyright 2001-2024 Thales Group

SVC_IO_GetReadPointer

Returns the pointer to the input buffer and its size. If the buffer is internally scattered or chunked in any other way, the pointer and the size relate to the current chunk only. The user can then directly access the buffer via the pointer, but only within the limits of the returned size.

Synopsis

```
unsigned int SVC_IO_GetReadPointer(FmMsgHandle token,
                                   void **read_pointer);
```

Input requirements

read_pointer must be the address of a void * variable that will be assigned the read buffer address

Output parameters

The function returns the number of bytes available in the read buffer.

 Copyright 2001-2024 Thales Group

SVC_IO_GetReadBuffer

Returns the pointer to consecutive input buffer, chunks will be consolidated if required.

The user can then directly access the full input buffer via the pointer.

Synopsis

```
unsigned int SVC_IO_GetReadBuffer(FmMsgHandle token,
                                  void **read_pointer);
```

Input requirements

read_pointer must be the address of a void * variable that will be assigned the read buffer address.

Output parameters

The function returns the number of bytes available in the read buffer.

 Copyright 2001-2024 Thales Group

SVC_IO_UpdateReadPointer

Tells the I/O subsystem that the 'size' amount has been consumed ("read") from the current chunk of the input buffer. Next **SVC_IO_GetReadPointer()** call will return the pointer to the remaining portion of the chunk, or to the new chunk altogether if the 'size' consumes all the remaining portion of the current input buffer chunk.

Synopsis

```
void SVC_IO_UpdateReadPointer(FmMsgHandle token,
                              int size);
```

Input parameters

This function assumes that the 'size' parameter does _not_ exceed the return value of the previous **SVC_IO_GetReadPointer()** call.

 Copyright 2001-2024 Thales Group

SVC_IO_GetWritePointer

Returns the pointer to the output buffer and its size. If the buffer is internally scattered or chunked in any other way, the pointer and the size relate to the current chunk only. The user can then directly access the buffer via the pointer, but only within the limits of the returned size.

Synopsis

```
unsigned int SVC_IO_GetWritePointer(FmMsgHandle token,
                                    void **write_pointer);
```

Input requirements

write_pointer must be the address of a void * variable that will be assigned the output buffer address.

Output parameters

The function returns the number of bytes available in the output buffer.

 Copyright 2001-2024 Thales Group

SVC_IO_GetWriteBuffer

Returns the pointer to the output buffer and its size. If the buffer is internally scattered or chunked in any other way, chunks will be consolidated if required. The user can then directly access the buffer via the pointer, but only within the limits of the returned size.

Synopsis

```
unsigned int SVC_IO_GetWriteBuffer(FmMsgHandle token,
                                   void **write_pointer);
```

Input requirements

write_pointer must be the address of a void * variable that will be assigned the output buffer address

Output parameters

The function returns the number of bytes available in the read buffer.

SVC_IO_UpdateWritePointer

These two functions do the same for write buffer as the previous ones does for reads from the command buffer.

Synopsis

```
unsigned int SVC_IO_GetWritePointer(FmMsgHandle token,
                                    void **write_pointer);
void SVC_IO_UpdateWritePointer(FmMsgHandle token,
                               int size);
```

Input requirements

write_pointer must be the address of a void * variable that will be assigned the output buffer address.

HIFACE Reply Management Functions

This section contains the legacy reply buffer management functions which are based on random access to the request and reply buffers. In this scheme the FM has access to the whole request and reply buffers and can read and write at any location within these buffers.

Each command that is sent to the FM has a token associated with it that links to the request buffer and, optionally, a reply buffer. When the command presented to the FM the token has a request buffer tied to it, but there is no reply buffer, it is the responsibility of the FM to allocate a reply buffer appropriate for the specific command. Reply buffers are optional, if the command only needs to return a status then no reply buffer is required. Typically a Functionality Module command will not send back a reply message if the status is non zero. The entire reply buffer is always returned to the caller, so the FM must set up the length *before* calling **SVC_SendReply**.

The Message Random Access section contains the following functions:

SVC_GetReplyBuffer

Allocates a reply buffer of a specified length and associates it with a 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>
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

Treats 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

```
#include <csa8hiface.h>
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.

 Copyright 2001-2024 Thales Group

SVC_SendReply

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.

Each call to a custom message handler MUST call **SVC_SendReply** once for each message processed.

Synopsis

```
#include <csa8hiface.h>
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 value of this parameter does not effect the reply delivery in any way.

Output Requirements

None.

 Copyright 2001-2024 Thales Group

SVC_ResizeReplyBuffer

Resizes the reply buffer associated with the specified token. The returned address need not be equal to the previous address of the reply buffer. The contents of the matching parts of the old and new reply buffers will always be the same.

Synopsis

```
#include <csa8hiface.h>
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.

 Copyright 2001-2024 Thales Group

SVC_DiscardReplyBuffer

Discards the current reply buffer. This function is usually called when a processing error is detected after the reply has been allocated.

Synopsis

```
#include <csa8hiface.h>
void SVC_DiscardReplyBuffer( HI_MsgHandle token );
```

Input Parameters

token	The token identifying the request

Output Requirements

None.

SVC_GetUserReplyBuffLen

Retrieves the length of the 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

```
#include <csa8hiface.h>
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.

 Copyright 2001-2024 Thales Group

SVC_GetRequest

Retrieves the address of request data.

Synopsis

```
#include <csa8hiface.h>
void *SVC_GetRequest( HI_MsgHandle token );
```

Input Parameters

token	The token identifying the request

Output Requirements

The request buffer address is returned.

 Copyright 2001-2024 Thales Group

SVC_GetRequestLength

Retrieves the length of request data in number of bytes.

Synopsis

```
#include <csa8hiface.h>
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.

 Copyright 2001-2024 Thales Group

SVC_GetReply

Retrieves the address of current reply buffer.

Synopsis

```
#include <csa8hiface.h>
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.

 Copyright 2001-2024 Thales Group

SVC_GetReplyLength

Retrieves the length of reply data in number of bytes.

Synopsis

```
#include <csa8hiface.h>
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.

 Copyright 2001-2024 Thales Group

Functionality Module Dispatch Switcher Functions

This section describes the firmware message dispatch management functions. There are two types of custom message dispatch functions and a FM designer needs to choose one and only one for their own FM. FM designers should use the type of dispatch function that best suits their source code.

When the FM registers its dispatch function it needs to specify the FMID of the current FM. The function **GetFMID()** is provided to allow the FM writer to easily get this value.

The Functionality Module Dispatch Switcher section contains the following functions:

> "FMSW_RegisterRandomDispatch" on the next page

> "FMSW_RegisterStreamDispatch" on page 126

Random Access Dispatch Function:

This is the older schema best suited for PTK style FMs. It allows the FM designer full random access to the contents of the request buffer and the response buffer.

Streaming Access Dispatch Function:

This is the newer schema that is used by the sample FMs. It allows the FM designer to read from the request buffer and write to the response buffer without needing to keep track of the current read and write points.

The read and write functions will provide automatic Endian conversion to integer values.

 Copyright 2001-2024 Thales Group

FMSW_RegisterRandomDispatch

Registers a Custom Command handler routine to the HSM. When a custom request is sent to the HSM with a FM ID equal to the fmNumber, the Dispatch function is called.

The type **FMSW_RandomDispatchFn_t** is a pointer to a function such as the following.

```
void RandomDispatchHandler(FmMsgHandle token,
                           void *reqBuffer,
                           uint32_t reqLength);
```

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

The Dispatch function returns void – it is the responsibility of the Dispatch function to call **SVC_SendReply()** to return the request response and to specify the return error code for the command.

The pair (reqBuffer, reqLength) defines the concatenated data that has been received on the request. See "Message Dispatch API Reference" on page 86 function for details on custom 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_RegisterRandomDispatch(
FMSW_FmNumber_t fmNumber,
FMSW_RandomDispatchFn_t dispatch);
```

Input Parameters

fmNumber	The FM identification number
dispatch	Pointer on custom request 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

FMSW_RegisterStreamDispatch

This function registers a Custom Command handler routine to the HSM. When a custom request is sent to the HSM with a FM ID equal to the fmNumber, the Dispatch function is called.

The type **FMSW_StreamDispatchFn_t** is a pointer to a function like this following example:

```
int StreamDispatchHandler(FmMsgHandle token);
```

The token is an opaque handle value identifying the request. The same token must be passed to **CL_Xxx()** functions. After the Dispatch function returns the HSM will take the return value and send the return value and response buffer back to the caller.

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_RegisterStreamDispatch(
FMSW_FmNumber_t fmNumber,
FMSW_StreamDispatchFn_t dispatch);
```

Input Parameters

fmNumber	The FM identification number
dispatch	Pointer on custom request 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

FMSW_GetImage API to validate an FM

This call returns a pointer to a Functionality Module image and a pointer to the size of the image, to assist the verification of FMs in compliance with industry and national standards. This is available once the FM is activated. To check the FM before it is deployed in an operational setting with your important keys and objects, we suggest that you first perform image integrity checks in a closed/offline environment, before deploying to the production environment.

API Specification:

```
/**
 * Obtain a pointer to the FM's image and its size.
 *
 * @param fmNumber
 *   FM Number.
 *
 * @param image
 *   A pointer to receive the pointer to the FM image.
 *
 * @param imageLength
 *   A pointer to receive the size of the FM image.
 *
 * @return
 *   @li FMSW_OK: The function processed the command OK.
 *   @li FMSW_BAD_FM_NUMBER: The FM number is incorrect.
 */

FMSW_STATUS FMSW_GetImage(
   FMSW_FmNumber_t   fmNumber,
   void              **image,
   uint32_t          *imageLength
);
```

Sample Code:

```
{
   void       *pimage;
   uint32_t   imageLength;

   FMSW_STATUS err = FMSW_GetImage(GetFMID(), &pimage, &imageLength);

   if (FMSW_OK != err)
   {
      // error handling
   }
}
```

Usage Notes

Use static IDs for the FMs to ensure that the retrieved image does not change. A dynamically allocated FM ID would not match, and the validation of the image would fail.

FM Support Functions

This section contains some support functions that can be used by the FM developer:

The Functionality Module Support section contains the following functions:

FM_GetNDRandom

Returns cryptographic quality (non-deterministic) random objects.

Synopsis

```
#include <fm.h>
int FM_GetNDRandom( char * out,
                    int len);
```

Input Parameters

out	Pointer to output buffer
len	number of bytes to store in "out"

Output Requirements

Return Value: number of bytes returned (should equal ‘len’)

 Copyright 2001-2024 Thales Group

FM_AddToExtLog

Have the FM add a message to the HSM Audit trail. The Audit trail is a secured stream of messages that are managed by the HSM Audit officer..

> **NOTE** **printf** writes to the HSM message stream and is best suited for debugging FMs.

Synopsis

```
#include <fm.h>
int FM_AddToExtLog(char * format,
                   … );
```

Input Parameters

format	**printf** style format string

Return Code

Function Code	Qualification
0	The function was successful.

FM_GetHsmInfo

Fetch information about the CORE.

Synopsis

```
#include <fm.h>
MD_RV FM_GetHsmInfo(MD_Info_t infotype,
                    void *pValue,
                    uint32 valueLen);
```

See "MD_GetHsmInfo" on page 98 for more details of types of information that can be fetched.

Extensions to the Standard C Library

This section describes two functions that work similar to standard C library's memset() and memcmp() functions but guarantee properties that the standard functions may not provide.

> "fm_memisequal" on the next page

> "fm_memzero" on the next page

> **NOTE** The functions described in this section are only available if you are using Luna HSM Firmware 7.7.0 or newer.

fm_memisequal

This function is similar to the standard C library's memcmp() function but differs in the following ways:

> It guarantees that the time comparison takes is strictly proportional to the length of the comparison. In other words, comparison takes the same time regardless of whether the memory areas being compared are different or equal. This property is also known as "constant time comparison".

> It is not a lexicographic comparator.Unlike memcmp, if the compared memory areas differ, this function does not report which one is greater or smaller in mathematical sense; it only reports whether the memory areas are equal or not.

Synopsis

```
#include <fmstring.h>
int fm_memisequal(const void *s1, const void *s2, size_t n);
```

Input Parameters

s1	Pointer to the first memory area.
s2	Pointer to the second memory area.
n	Number of bytes in s1 and s2 to compare.

Output Requirements

Return value: zero if the s1 and s2 memory areas are identical, not zero otherwise.

fm_memzero

This function is similar to a call memset(s, 0, n) of the standard C library’s memset() function. Unlike memset(), fm_memzero() guarantees that its call will never be optimized out by the compiler; that is, it guarantees that the first *n* bytes of the memory area *s* will be set to zero even if the compiler determines it is not necessary. For example, the compiler can determine that the memory area *s* is not used by the execution flow of the code after memset(s, 0, n) call. It may then consider the memset() call redundant and remove it. This never happens if you use the fm_memzero function call.

Synopsis

```
#include <fmstring.h>
void fm_memzero(void *s, size_t n);
```

Input Parameters

S	Pointer to the first memory area.
N	Number of bytes in s to set to zero.

Output Requirements

None

Extensions to PKCS#11

This section lists functions that are extensions to the PKCS#11 standard. These functions are provided by the Thales implementation of the PKCS#11 Cryptoki library; that is, the PKCS#11 standard itself does not specify them. FM code must include **fmsupt.h** to use any of them.

> **NOTE** Functions that are identical to the ones listed below exist in the Luna HSM Client, with a 'CA_' prefix in their names instead of an 'FM_' prefix. The descriptions of the Luna HSM Client PKCS#11 extensions apply to all of the functions listed below. For descriptions of any of the functions listed in this section, refer to Luna Extensions to PKCS#11.
>
> V2 functions pass in a 16-byte application accessID while non-V2 functions pass in an 8-byte application accessID (4 bytes high and 4 bytes low).

Operational concepts of Luna HSMs related to the functions below are described in the HSM and partition administration guides. For more information, refer to About the HSM Administration Guide and About the Partition Administration Guide.

Luna HSM Firmware 7.7.0 and Newer

The following extensions were introduced with Luna HSM Firmware 7.7.0:

- FM_SMKRollover
- FM_OpenSessionWithAppIDV2
- FM_CloseApplicationIDV2
- FM_CloseApplicationIDForContainerV2
- FM_OpenApplicationIDV2
- FM_OpenApplicationIDForContainerV2
- FM_SetApplicationIDV2
- FM_RandomizeApplicationID
- FM_GetApplicationID
- FM_SIMExtract
- FM_SIMInsert
- FM_SIMMultiSign
- FM_AuthorizeKey
- FM_SetAuthorizationData
- FM_ResetAuthorizationData
- FM_AssignKey
- FM_IncrementFailedAuthCount
- FM_DeriveKeyAndWrap
- FM_GetSessionInfoV2
- FM_CloneAsSourceInit

- FM_CloneAsTargetInit
- FM_CloneAsSource
- FM_CloneAsTarget
- FM_GetConfigurationElementDescription
- FM_GetHSMCapabilities (calls CA_GetHSMCapabilitiesSet)
- FM_GetHSMCapabilitySetting
- FM_GetHSMPolicies (calls CA_GetHSMPoliciesSet)
- FM_GetHSMPolicySetting
- FM_GetTokenCapabilities
- FM_GetTokenCapabilitySetting (calls CA_GetContainerCapabilitySetting)
- FM_GetTokenPolicies
- FM_GetTokenPolicySetting (calls CA_GetContainerPolicySetting)
- FM_FindAdminSlotForSlot

Luna HSM Firmware 7.4.0 and Newer

The following extensions were introduced with Luna HSM Firmware 7.4.0:

- FM_Bip32ExportPublicKey
- FM_Bip32ImportPublicKey

Serial Communication Functions

This section contains functions for using the serial ports on the HSM.

The Serial Communications section contains the following functions:

 Copyright 2001-2024 Thales Group

SERIAL_SendData

Synopsis

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

Description

Sends a character array over a serial port.

Parameters

port	Serial port number (0 based). Specify port 0 to redirect the output to the HSM trace log
buf	Pointer to an array of bytes to be sent
bufLen	length of the buffer, in bytes
timeout	Number of milliseconds to wait for a character to be sent. A timeout of -1 will use the default timeout. Default timeout is 2000 ms. **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

Function Code	Qualification
0	The characters were sent successfully.
-1	There was an error in operation.

 Copyright 2001-2024 Thales Group

SERIAL_RecieveData

Retrieves an arbitrary length of characters from the serial port.

Synopsis

```
#include <serial.h>
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 HSM trace log
buf	Pointer to an array of bytes to be sent
len	Pointer to an integer which will hold the actual number of characters received
bufLen	length of the buffer, in bytes
timeout	Number of milliseconds to wait for a character to appear. A timeout of -1 will use the default timeout. The default timeout is 4000ms + (10ms * number of characters) Example, reading 25 characters: 4000 + (10 * 25) = 4250ms = 4.25s

Return Code

Function Code	Qualification
0	Requested number of bytes has been received.
-1	Less than the requested number of bytes has been received.

 Copyright 2001-2024 Thales Group

SERIAL_WaitReply

Waits for a character to appear on the serial port.

Synopsis

```
#include <serial.h>
int SERIAL_WaitReply( int port );
```

Parameters

port	Serial port number (0 based). Specify port 0 to redirect the output to the HSM trace log

Return Code

Function Code	Qualification
0	There is a character at the serial port.
-1	Timeout occurred and no data appeared.

 Copyright 2001-2024 Thales Group

SERIAL_FlushRX

Flushes the receive buffer of the specified serial port.

Synopsis

```
#include <serial.h>
void SERIAL_FlushRX( int port );
```

Parameters

port	Serial port number (0 based). Specify port 0 to redirect the output to the HSM trace log

 Copyright 2001-2024 Thales Group

SERIAL_GetNumPorts

Returns the number of serial ports available.

Synopsis

```
#include <serial.h>
int SERIAL_GetNumPorts(void);
```

Parameters

None.

Return Value

The number of serial ports available.

 Copyright 2001-2024 Thales Group

SERIAL_InitPort

Initializes the specified serial port to the parameters “9600 8N1” with no handshake.

Synopsis

```
#include <serial.h>
int SERIAL_InitPort(int port);
```

Parameters

port	Serial port number (0 based). Specify port 0 to redirect the output to the HSM trace log

Return Code:

Function Code	Qualification
0	The serial port was initialized successfully
-1	There was an error initializing the port.

 Copyright 2001-2024 Thales Group

SERIAL_GetControlLines

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

```
#include <serial.h>
int SERIAL_GetControlLines( int port,
                            unsigned char *bitmap);
```

Parameters

port	Serial port number (0 based). Specify port 0 to redirect the output to the HSM trace log
bitmap	Pointer to a character, which will have the resulting bitmap

Return Code:

Function Code	Qualification
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

Modifies the control lines (DTR/RTS).

Synopsis

```
#include <serial.h>
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 HSM trace log
bitmap	Bitmap of control lines to be modified . Input control lines are silently ignored
op	One of MCL_OP_SET/MCL_OP_CLEAR control lines specified in the bitmap parameter

Return Code

Function Code	Qualification
0	The function was successful.
-1	The function failed.

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
```

 Copyright 2001-2024 Thales Group

SERIAL_SetMode

Used to set the serial port communication parameters.

Synopsis

```
#include <serial.h>
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 HSM trace 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

Function Code	Qualification
0	Mode changed successfully.
-1	There was an error in operation.

 Copyright 2001-2024 Thales Group

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

```
#include <serial.h>
int SERIAL_Open( int port );
```

Parameters

port	Serial port number (0 based). Specify port 0 to redirect the output to the HSM trace log

Return Code

Function Code	Qualification
0	Port opened successfully.
-1	An error prevented the serial port from opening.

Comments

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

 Copyright 2001-2024 Thales Group

SERIAL_Close

Releases ownership of the serial port.

Synopsis

```
#include <serial.h>
void SERIAL_Close(int port);
```

Parameters

port	Serial port number (0 based). Specify port 0 to redirect the output to the HSM trace log

Comments

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

 Copyright 2001-2024 Thales Group

High Resolution Timer Functions

The High Resolution Timer section contains the following functions:

> "THR_BeginTiming" on the next page

> "THR_UpdateTiming" on page 148

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 parameters

Parameter	Value
secs	Time value in seconds
ns	Time value in nanoseconds. The nanoseconds value must always be less than 109 (which is equal to 1 second)

THR_BeginTiming

Starts a high-resolution timing operation. The timing resolution is 20ns, and the accuracy of the timer is about 1 microsecond.

Synopsis

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

Input Parameters

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

 Copyright 2001-2024 Thales Group

THR_UpdateTiming

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

Synopsis

```
#include <timing.h>
void THR_UpdateTiming(const THR_TIME *start,
                      THR_TIME*elapsed);
```

Input Parameters

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

 Copyright 2001-2024 Thales Group

Current Application ID functions

These functions can be used to obtain and manipulate the Application ID of the calling application.

The AppID is used by the HSM core to identify the application making the call. It identifies which Cryptoki sessions are valid for the specified caller.

The Current Applications ID section contains the following functions:

> "FM_GetCurrentAppId" on the next page

> "FM_SetCurrentAppId" on page 151

FM_GetCurrentAppId

Returns the AppID 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>
Uint64_t FM_GetCurrentAppId(void);
```

Return Code

The AppId of the application which originated the current request to the FM.

 Copyright 2001-2024 Thales Group

FM_SetCurrentAppId

Overrides the AppId recorded in the current request originated from the host side. If there is no active request the function does nothing.

Synopsis

```
#include <fmappid.h>
void FM_SetCurrentPid(uint64_t appid);
```

Parameters

appid	The new AppId to be recorded in the request he new A

Return Code

none

 Copyright 2001-2024 Thales Group

PKCS#11 State Management Functions

The functions listed in this section allow 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.

The PKCS#11 State Management section includes the following functions:

> "FM_SetSessionUserData" on the next page

> "FM_GetSessionUserData" on page 154

FM_SetSessionUserData

Associates 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()** call), 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 the 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 the 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. Value must not be NULL if userData is not NULL

Return Code

The function returns the following codes:

Function Code	Qualification
CKR_OK	The operation was successful
CKR_ARGUMENTS_BAD	Free user date 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

Obtains 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. Value must not be NULL.

Return Code

The function returns the following codes:

Function Code	Qualification
CKR_OK	The operation was successful. The associated user data is placed in the variable specified by the ppUserData.
CKR_ARGUMENTS_BAD	ppUserData 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

 Copyright 2001-2024 Thales Group

FM Header Definition Macro

DEFINE_FM_HEADER

The FM header contains information which is used at runtime and must be present in all functionality modules (FMs),

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

Synopsis

```
#include <mkfmhdr.h>
```

Usage

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

FM_NUMBER	Must be a manifest constant. Refer to "FM_NUMBER" below for more information.
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 **ctfm**)
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.

FM_NUMBER

This field is used to define the FM identifier number (FMID). Note the following regarding FMIDs:

> The FMID is used by **MD_SendReceive()** to identify which FM receives the request.

> Following a reboot, the HSM starts up FMs in ascending order of their FMID values.

> If more than one FM is loaded onto the HSM, each FM must have a different ID value.

> The FMID can be returned to the client in the following ways:

- The client may call **MD_GetFmIdFromName()** to return the FMID with a matching PRODUCT_ID.
- The client may call **GetFMID(void)** from the FM itself to get the FMID.

> The FM_NUMBER value can be any of the following:

- A number from FM_ID_CUSTOM_MIN to FM_ID_CUSTOM_MAX.

 Copyright 2001-2024 Thales Group

- One of the special values FMID_ALLOCATE_HIGH, FMID_ALLOCATE_NORM, or FMID_ALLOCATE_LOW.

Normally, select a number between FM_ID_CUSTOM_MIN and FM_ID_CUSTOM_MAX for each type of FM you may want to load onto the HSM. If you only have one FM to manage in your HSM, you should specify the standard static ID value of FM_NUMBER_CUSTOM_FM (=FM_ID_CUSTOM_MIN).

Because these ID values are fixed, the client application can have the FMID of its FM fixed in the source code. However, if there are a lot of FMs , then FMID values can clash. Use the special values FMID_ALLOCATE_HIGH, FMID_ALLOCATE_NORM, or FMID_ALLOCATE_LOW to request that the HSM dynamically assign an unused ID value to the FM when it is loaded onto the HSM. This process ensures FMID values do not clash. The three different ranges allow the FM designer some control over the initialization order of their FMs, where FMID_ALLOCATE_LOW FMs are started up first. If the startup order does not matter, then you should specify FMID_ALLOCATE_NORM.

FM PKCS#11 EXTENSION FUNCTIONS - updated for post-7.7.0 HSM version

Functions in the list below are called PKCS#11 extensions. They are provided by the Thales's implementation of the PKCS#11 Cryptoki library. The PKCS#11 standard itself does not specify them.

Identical functions also exist in the Luna HSM client software. The only difference is that the client's versions of the functions have a 'CA_' prefix in their names instead of the 'FM_' prefix. 'CA_' equivalents are fully described in Luna HSM SDK Reference [see Extensions to PKCS#11]. By extension, their descriptions fully apply to the functions listed in this section.

Operation concepts of Luna HSMs related to the functions are described in About the HSM Administration Guide and About the Partition Administration Guide.

FM code must include "fmsupt.h" to use any of the functions below.

"FM_SMKRollover"

"FM_Bip32ExportPublicKey"

"FM_Bip32ImportPublicKey"

"FM_OpenSessionWithAppIDV2"

"FM_CloseApplicationIDV2"

"FM_CloseApplicationIDForContainerV2"

"FM_OpenApplicationIDV2"

"FM_OpenApplicationIDForContainerV2"

"FM_SetApplicationIDV2"

"FM_RandomizeApplicationID"

"FM_GetApplicationID"

"FM_SIMExtract"

"FM_SIMInsert"

"FM_SIMMultiSign"

"FM_AuthorizeKey"

"FM_SetAuthorizationData"

"FM_ResetAuthorizationData"

"FM_AssignKey"

"FM_IncrementFailedAuthCount"

"FM_DeriveKeyAndWrap"

"FM_GetSessionInfoV2"

"FM_CloneAsSourceInit"

"FM_CloneAsTargetInit"

"FM_CloneAsSource"

"FM_CloneAsTarget"

"FM_GetConfigurationElementDescription"

"FM_GetHSMCapabilities"

"FM_GetHSMCapabilitySetting"

"FM_GetHSMPolicies"

"FM_GetHSMPolicySetting"

"FM_GetTokenCapabilities"

"FM_GetTokenCapabilitySetting"

"FM_GetTokenPolicies"

"FM_GetTokenPolicySetting"

"FM_FindAdminSlotForSlot"

FM EXTENSIONS TO THE STANDARD C LIBRARY

This section describes two functions that work similar to standard `memset()` and `memcmp()`, but that guarantee properties that standard functions might not provide.

fm_memisequal()

Similar to the standard `memcmp()` function, except it guarantees that the time taken by comparison is strictly proportional to the length of the comparison. In other words, comparison takes the same time regardless if the memory areas being compared are different or equal. This property is sometimes called constant time comparison.

Another difference is that this function is not a lexicographic comparator. Unlike `memcmp`, if the compared memory areas differ this function does not indicate which one is greater or smaller in a mathematical sense. It just indicates if the memory areas are equal or not.

Synopsis

```
#include <fmstring.h>
int fm_memisequal(const void *s1, const void *s2, size_t n);
```

Input Parameters

s1	Pointer to the first memory area
s2	Pointer to the second memory area
n	number of bytes in s1 and s2 to compare

Output Requirements

Return value: zero if the s1 and s2 memory areas are identical, not zero otherwise.

fm_memzero()

Similar to a call `memset(s, 0, n)` of the standard C library's `memset()` function. Unlike `memset()`, `fm_memzero()` guarantees that its call will never be optimized out by the compiler. In other words, it guarantees that the first `n` bytes of the memory area `s` will be set to zero even if the compiler considers it not necessary. For example, the compiler can determine that the memory area `s` is not used by the execution flow of the code after a `memset(s, 0, n)` call; thus it may consider the `memset()` call redundant and remove it. This can never happen if `fm_memzero` function call is used.

Synopsis

```
#include <fmstring.h>
void fm_memzero(void *s, size_t n);
```

Input Parameters

S	Pointer to the memory area
N	number of bytes in s to set to zero

Output Requirements

None

CRYPTOGRAPHIC MECHANISMS SUPPORTED BY FM CRYPTO ENGINES

FM Crypto Engines support all mechanisms that the host Thales Cryptoki library accepts for PKCS#11 functions `C_EncryptInit()`, `C_DecryptInit()`, `C_SignInit()`, `C_VerifyInit()` and `C_DigestInit()`. The exhaustive list of mechanisms can be found in the Luna HSM SDK Reference [see Supported Mechanisms].

CRYPTOGRAPHIC MECHANISMS SUPPORTED BY FM CIPHER AND HASH OBJECTS

FM Cipher Objects and Hash Objects support the following limited set of mechanisms:

- CKM_AES_ECB
- CKM_AES_CBC

- CKM_AES_CBC_PAD
- CKM_AES_MAC
- CKM_AES_MAC_GENERAL
- CKM_DES_ECB
- CKM_DES_CBC
- CKM_DES_CBC_PAD,
- CKM_DES_MAC
- CKM_DES_MAC_GENERAL
- CKM_DES3_ECB
- CKM_DES3_CBC
- CKM_DES3_CBC_PAD
- CKM_DES3_MAC
- CKM_DES3_MAC_GENERAL
- CKM_CAST128_ECB
- CKM_CAST128_CBC
- CKM_CAST128_CBC_PAD
- CKM_CAST128_MAC
- CKM_CAST128_MAC_GENERAL
- CKM_RC2_ECB
- CKM_RC2_CBC
- CKM_RC2_CBC_PAD
- CKM_RC2_MAC
- CKM_RC2_MAC_GENERAL
- CKM_MD5_HMAC
- CKM_MD5_HMAC_GENERAL
- CKM_SHA_1_HMAC
- CKM_SHA_1_HMAC_GENERAL
- CKM_RIPEMD160_HMAC
- CKM_RIPEMD160_HMAC_GENERAL
- CKM_RSA_X_509
- CKM_RSA_PKCS
- CKM_RSA_PKCS_OAEP
- CKM_RSA_9796
- CKM_MD5_RSA_PKCS
- CKM_SHA1_RSA_PKCS

- CKM_SHA224_RSA_PKCS
- CKM_SHA256_RSA_PKCS
- CKM_SHA384_RSA_PKCS
- CKM_SHA512_RSA_PKCS
- CKM_DSA
- CKM_MD5
- CKM_RIPEMD160
- CKM_SHA_1
- CKM_SHA224
- CKM_SHA256
- CKM_SHA384
- CKM_SHA512