THALES

ProtectToolkit 5.9.1
ProtectToolkit Functionality Module SDK

PROGRAMMING GUIDE

Document Information

Last Updated	2025-09-15 18:12:32-05:00

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.

CONTENTS

Preface: About the FM SDK Programming Guide 10
- Document Conventions 10
- Support Contacts 13

Chapter 1: Overview 14
- Features 14
- Constraints 14

Chapter 2: Configuring the FM SDK Environment 16
- Setting the Environment Variables 16
- Windows Environment Variable Paths 17

Chapter 3: FM Architecture 18
- FM Support within the HSM Hardware 18
- FM Support in Emulation Mode 21

Chapter 4: FM Development 24
- Lifecycle Outline 24
- Initial Development 25
- Emulation Build 25
- Adapter Build 25
- Production Build 26
- Key Management 26
- Contents of the $(FMSDK) Directory 27
- SDK Installation Tips 28
- Protecting Data Storage of FM 28
- Cprov Function Patching 29
- FM Message Dispatching 29
- Handling Host Processes 30
- Memory Alignment Issues 30
- Memory Endian Issues 30

Chapter 5: Setting up an MSYS environment and cross-compiler 31
- Download MinGW and the toolchain source code 31
- Build and install the cross-compiler 32
  - Automated cross-compiler build 32
  - Manual cross-compiler build 32
- Set the MSYS environment to include the FMDIR and CPROVDIR directories 33

Chapter 6: FM Samples 35

Included Samples ... 35
RSAENC ... 35
XORSign ... 36
restrict ... 36
safedebug ... 36
cipherobj ... 36
smfs ... 36
secfmenc ... 36
ssldemo ... 36
usbdemo ... 36
javahsmreset ... 37
javahsmstate ... 37
FM Sample Directories ... 37
Signing Sample FMs ... 38
FMs in Emulation Mode ... 39

Chapter 7: Building sample FMs in emulation mode on Windows ... 41
Copy the samples and emulation source folders ... 41
Set the environment variables ... 41
Compile the binaries ... 41

Chapter 8: FM Deployment ... 43
Deploy a Single FM ... 43
Deploy Multiple FMs ... 44
Configuring a unique FMID ... 44
Calling FMID ... 45

Chapter 9: Utilities Reference ... 46
ctcert ... 46
ctconf ... 46
ctfm ... 46
mkfm ... 46

Chapter 10: Cipher Object ... 48
The Cipher Object Access API ... 48
FmCreateCipherObject ... 49
Cipher Object Functions ... 50
New ... 51
Free ... 51
GetInfo ... 51
EncInit ... 52
EncryptUpdate ... 53
EncryptFinal ... 54
DecInit ... 55
DecryptUpdate ... 56
DecryptFinal ... 56
SignInit ... 57

SignUpdate 58
SignFinal 58
SignRecover 59
VerifyInit 60
VerifyUpdate 61
VerifyFinal 61
VerifyRecover 62
Verify 63
LoadParam 64
UnloadParam 65
Config (Obsolete) 65
Status (Obsolete) 66
EncodeState (Obsolete) 66
DecodeState (Obsolete) 67
Cipher Object Function Error Codes 67
Algorithm-Specific Cipher Information 67
AES Cipher Object 68
DES Cipher Object 69
Triple DES Cipher Object 70
ECDSA Cipher Object 72
IDEA Cipher Object 73
RC2 Cipher Object 74
RC4 Cipher Object 75
RSA Cipher Object 75

Chapter 11: Hash Object 79
FmCreateHashObject 79
Hash Object Functions 80
Free 80
Init 81
Update 81
Final 82
GetInfo 82
LoadParam 83
UnloadParam 84

Chapter 12: Setting Privilege Level 85
SetPrivilegeLevel 85

Chapter 13: SMFS Reference 86
Important Constants 86
Error Codes 86
File Attributes Structure (SmFsAttr) 87
Function Descriptions 87
SmFsCreateDir 88
SmFsCloseFile 88
SmFsCalcFree 88

SmFsCreateFile ... 89
SmFsDeleteFile ... 89
SmFsFindFile ... 89
SmFsFindFileClose ... 90
SmFsFindFileInit ... 90
SmFsGetFileAttr ... 91
SmFsGetOpenFileAttr ... 91
SmFsOpenFile ... 91
SmFsReadFile ... 92
SmFsRenameFile ... 92
SmFsWriteFile ... 93

Chapter 14: FMDEBUG Reference ... 94
debug (macro) ... 94
printf/vprintf ... 95
DBG_INIT ... 95
DBG ... 95
DBG_PRINT ... 95
DBG_STR ... 96
DUMP ... 96
DBG_FINAL ... 97

Chapter 15: Message Dispatch API Reference ... 98
Function Descriptions: ... 98
MD_Initialize ... 98
MD_Finalize ... 99
MD_GetHsmCount ... 99
MD_GetHsmState ... 99
MD_ResetHsm ... 101
MD_SendReceive ... 102
MD_GetParameter ... 105
FM Host Legacy Functions API ... 106

Chapter 16: HSM Functions Reference ... 107
Summary ... 107
Subset of ISO C99 standard library ... 107
Extensions to the Standard C API ... 108
Unsupported Standard C APIs ... 108
HIFACE Reply Management Functions ... 108
SVC_GetReplyBuffer ... 109
SVC_ConvertReqToReply ... 109
SVC_SendReply ... 109
SVC_ResizeReplyBuffer ... 110
SVC_DiscardReplyBuffer ... 110
SVC_GetUserReplyBufLen ... 111
SVC_GetPid ... 111
SVC_GetOid ... 111

SVC_GetRequest ... 112
SVC_GetRequestLength ... 112
SVC_GetReply ... 112
SVC_GetReplyLength ... 113
Functionality module dispatch switcher function ... 114
FMSW_RegisterDispatch ... 114
Serial Communication Functions ... 114
SERIAL_GetNumPorts ... 115
SERIAL_Open ... 115
SERIAL_Close ... 116
SERIAL_InitPort ... 116
SERIAL_SendData ... 116
SERIAL_WaitReply ... 117
SERIAL_ReceiveData ... 117
SERIAL_FlushRX ... 118
SERIAL_GetControlLines ... 118
SERIAL_SetControlLines ... 119
SERIAL_SetMode ... 119
High Resolution Timer Functions ... 120
THR_BeginTiming ... 120
THR_UpdateTiming ... 121
Cprov function patching helper function ... 121
OS_GetCprovFuncTable ... 122
Current Application ID functions ... 122
FM_GetCurrentPid ... 122
FM_GetCurrentOid ... 122
FM_SetCurrentPid ... 123
FM_SetCurrentOid ... 123
PKCS#11 State Management Functions ... 123
FM_SetAppUserData ... 124
FM_GetAppUserData ... 124
FM_SetSlotUserData ... 125
FM_GetSlotUserData ... 126
FM_SetTokenUserData ... 126
FM_GetTokenUserData ... 127
FM_SetTokenAppUserData ... 128
FM_GetTokenAppUserData ... 128
FM_SetSessionUserData ... 129
FM_GetSessionUserData ... 130
FM Header Definition Macro ... 130

Chapter 17: USB API Reference ... 132
USB File System ... 132
Data Structures ... 132
Example Usage ... 133
Function Descriptions ... 133
USBFS_Close ... 134

USBFS_Destroy 134
USBFS_Finalize 135
USBFS_GetInfo 136
USBFS_Init 137
USBFS_New 137
USBFS_Open 138
USBFS_ReadData 139
USBFS_WriteData 140
USB API Vendor-Defined Error Codes 140

Glossary 142

PREFACE: About the FM SDK Programming Guide

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

> "Overview" on page 14
> "Configuring the FM SDK Environment" on page 16
> "FM Architecture" on page 18
> "FM Development" on page 24
> "Setting up an MSYS environment and cross-compiler" on page 31
> "FM Samples" on page 35
> "Building sample FMs in emulation mode on Windows" on page 41
> "Utilities Reference" on page 46
> "Cipher Object" on page 48
> "Hash Object" on page 79
> "Setting Privilege Level" on page 85
> "SMFS Reference" on page 86
> "FMDEBUG Reference" on page 94
> "Message Dispatch API Reference" on page 98
> "HSM Functions Reference" on page 107
> "USB API Reference" on page 132

This preface also includes the following information about this document:

> "Document Conventions" below
> "Support Contacts" on page 13

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

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 cross-references to other documents in this documentation set.
<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.

Format	Convention
{**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.

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 to this service is governed by the support plan arrangements made between Thales and your organization. Please consult this support plan for further information about 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. The Customer Support Portal is 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. You can also use the portal to create and manage support cases.

> **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).

CHAPTER 1: Overview

A Functionality Module (FM) is custom-developed, customer-specific code that operates within the secure confines of a Hardware Security Module (HSM). You can use the ProtectToolkit FM SDK to develop FMs for the ProtectServer Network HSM and ProtectServer PCIe HSM, introduced in release 5.0.

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

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

This document is intended for software developers, as a technical reference describing the programming methodologies and functions used for developing FMs and host-side applications.

Features

Host apps are supported on all platforms supporting the ProtectToolkit SDK. FMs have to be cross-compiled on Linux. The FM SDK provides the following components:

- Sample FM code
- Sample host-side code
- Build scripts
- Host-side libraries
- Java classes to access HSMs
- Java docs
- FM binary image generation tools
- FM libraries
- FM emulation libraries
- 8 MB of storage space is available on the HSM to store FMs.

Constraints

The ProtectToolkit FM SDK has the following limitations on FM development:

- FMs compiled using the ProtectToolkit FM SDK 5.4 or newer do not load correctly into HSMs using firmware 5.00.xx.

- Downgrading HSMs from firmware 5.01.00 or newer to 5.00.08 or older will delete any FMs on the device that were compiled using the ProtectToolkit FM SDK 5.4 or newer.
- FMs that have been loaded onto the HSM are not deleted from the HSM after a tamper event. FMs must be deleted by using the **ctfm** utility before tampering the HSM. For more information about tampering the HSM and deleting FMs using the **ctfm** utility, see Tampering or Decommissioning the HSM and CTFM in the "Operational Tasks" and "Command Line Utilities Reference" sections of the *ProtectToolkit-C Administration Guide*.

CHAPTER 2: Configuring the FM SDK Environment

This section describes how to configure your FM SDK environment by setting environment variables. The FM SDK must be installed before proceeding with the instructions in this section. See the following sections for more information about FM SDK installation:

- Installing ProtectToolkit on Windows in the "ProtectToolkit Software Installation" section of the *ProtectServer HSM and ProtectToolkit Installation Guide*.
- Installing ProtectToolkit on Unix/Linux in the "ProtectToolkit Software Installation" section of the *ProtectServer HSM and ProtectToolkit Installation Guide*.
- Installing ProtectToolkit on Linux Manually in the "ProtectToolkit Software Installation" section of the *ProtectServer HSM and ProtectToolkit Installation Guide*.

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

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

- CPROVDIR: Specifies the installed location of the Cprov SDK (ProtectToolkit-C)
- FMDIR: Specifies the installed location of the FM SDK

Setting the Environment Variables

The environment variables are set using scripts.

To set the environment variables on Linux

1. Go to the ProtectToolkit software installation directory:

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

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

   ```
   /setvars.sh
   ```

To set the environment variables on Windows

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

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

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

   ```
   /fmsdkvars.bat
   ```

Windows Environment Variable Paths

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

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

CHAPTER 3: FM Architecture

This chapter describes the basic architecture of your FM SDK, and includes the following sections:

> "FM Support within the HSM Hardware" below
> "FM Support in Emulation Mode" on page 21

FM Support within the HSM Hardware

FMs allow application developers to design security-sensitive program code, which is uploaded into the HSM and operates as part of the HSM firmware. The ProtectToolkit FM SDK also provides application developers with APIs to develop applications on a host to interface to the HSM.

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

Starting with ProtectToolkit version 5.4, you can upload multiple custom FMs to an HSM and use them simultaneously. Only one PKCS#11 patched FM can be loaded and used at a time. If a patched FM is already loaded, it is overwritten by the new FM. Refer to "Custom Functions" on the next pageand "PKCS#11 Patched Functions" on page 20 in the *FM SDK Programming Guide* for descriptions of these FM types.

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

> "The components and interfaces in a system using Functionality Modules for Custom Functions" on the next page
> "The components and interfaces in a system using Functionality Modules for Patched Functions" on page 20

The diagrams are presented this way for ease of illustration, and because FMs tend to be used in environments where either only custom functions or only PKCS#11 functions are utilized. However, there is no constraint that prevents a FM from containing a mixture of both custom and patched PKCS#11 functions.

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

The data flows in the diagrams are depicted using the notation:

API (Function Type)

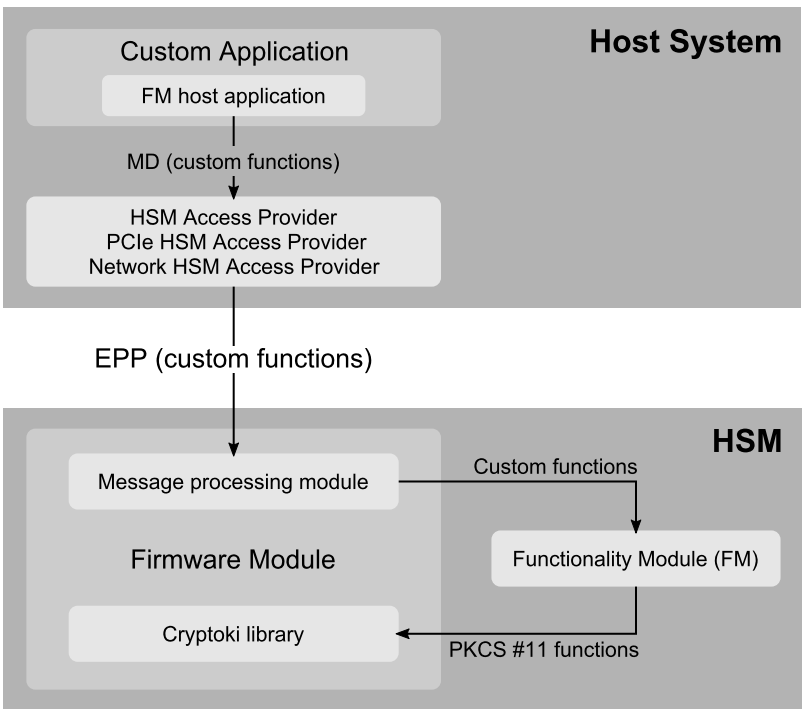
For example, in "The components and interfaces in a system using Functionality Modules for Custom Functions" on the next page, the arrow labeled MD (custom functions) indicates the flow of custom function request packages passed between components via the Message Dispatch (MD) API (refer to "Message

Dispatch API Reference" on page 98 for details). EPP (custom functions) refers to Custom function request packages passed from the host across the PCI bus (in the case of local HSMs) or across a TCP/IP link (in the case of remote HSMs), via an unpublished Thales-proprietary protocol.

Custom Functions

"The components and interfaces in a system using Functionality Modules for Custom Functions" below depicts 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 implements the protocol.

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



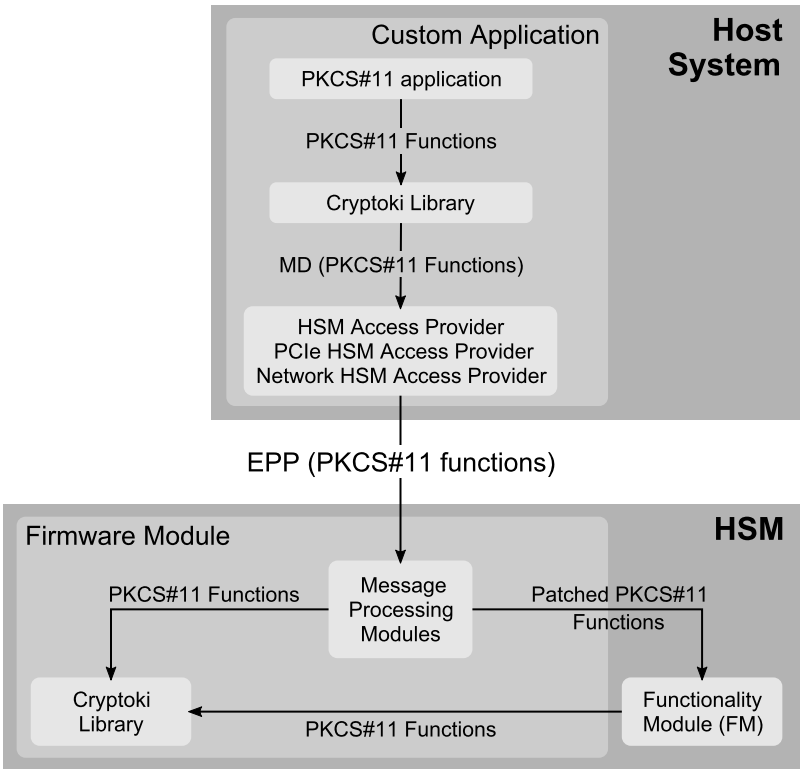
These message response and request packages are transferred between the application and the PCIe or Network HSM Access Provider, via the Message Dispatch (MD) API. The legacy FM Host API has been deprecated since Version 2.0 of ProtectProcessing Orange (legacy name of the FM SDK). As of FM SDK 5, the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

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

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

PKCS#11 Patched Functions

Figure 2: The components and interfaces in a system using Functionality Modules for Patched Functions



"FM Architecture" on page 18 depicts the components contained in the Host system and the HSM when using PKCS#11 functions. The custom application is executed on the host system. The application accesses patched PKCS#11 functions in the FM and the standard PKCS#11 functionality of the Cryptoki library provided in the firmware module via a standard PKCS#11 interface provided by the Cryptoki library on the host system.

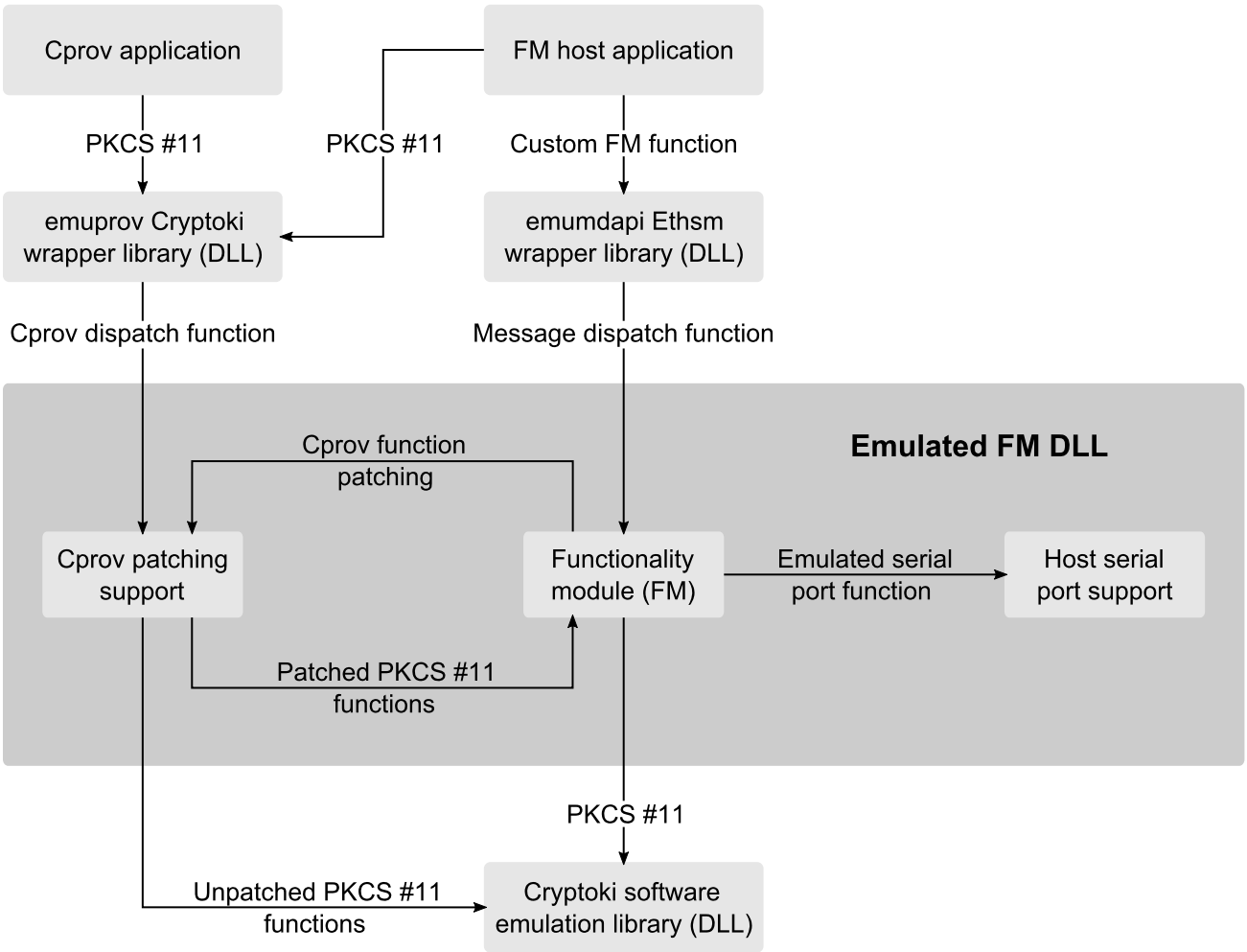
The Message Processing Modules contain a list of patched PKCS#11 functions, to which the incoming function is compared. The Message Processing Modules call the PKCS#11 function from HSM firmware if the function isn't patched or gives control to an FM version of same function if it is patched. FM implementations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

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

FM Support in Emulation Mode

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

Figure 3: The components of a system emulating Functionality Modules



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

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

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

Emulation Mode Limitations

Supported Platforms

FM Emulation using the ProtectToolkit FM SDK is supported on Windows and Linux.

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

Supported C APIs

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

ctype.h
isascii, **toascii**

string.h
strdup, **strsep**

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

If your emulation environment uses **glibc**, such as the standard **gcc** environment on Red Hat, you may define the following to enable these non-standard **libc** functions in emulation mode:

Note that we test for **gcc**, not **glibc**, because the compile time **glibc** flag is set in **features.h**, but OVERRIDE_GNU_SOURCE must be set before **features.h** is included.

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

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

Source Level Debugging

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

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

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

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

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

Random Number Generator

The emulation Cryptoki library does not provide true random numbers. Although the FIPS 140-approved Pseudo Random Number Generator is implemented in the emulation version, there is insufficient entropy to make good quality random numbers.

Do not use the emulation to create production-grade keys.

Tampering the Secure Memory

The emulation Cryptoki library does not support the concept of a hardware tamper event.

You may delete the emulation Cryptoki data directory to simulate a total loss of secure memory. However you should only do this when the Cryptoki library is not running.

The emulation Cryptoki data directory default location is

Windows: **c:\cryptoki**

Linux: **~/.cryptoki/cryptoki**

The emulation data directory can be changed via the ET_PTKC_SW_DATAPATH entry in the ProtectToolkit configuration file. See Storage Location Assignment in the "Configuration Items" section of the *ProtectServer HSM and ProtectToolkit Installation Guide* for more information.

Cryptoki Function Patching

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

ETHSM

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

Cryptoki FM Object

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

CHAPTER 4: FM Development

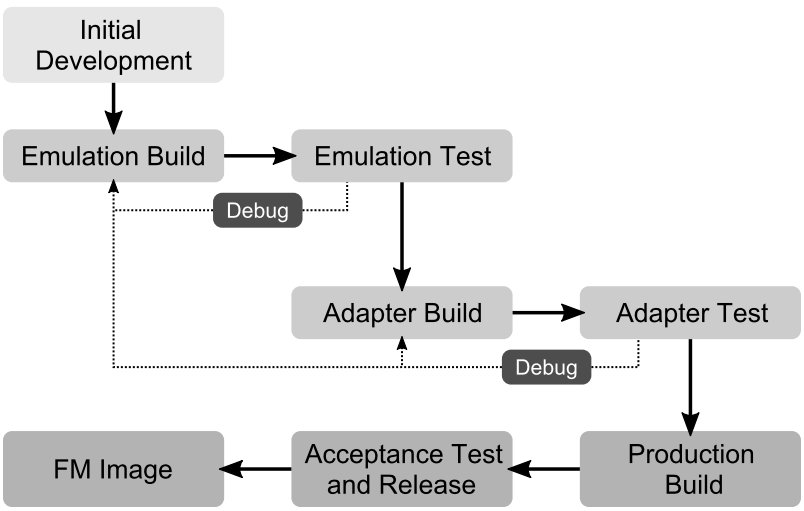
This chapter outlines the development life cycle of your FM SDK, and includes the following:

- "Lifecycle Outline" below
- "Initial Development" on the next page
- "Emulation Build" on the next page
- "Adapter Build" on the next page
- "Production Build" on page 26
- "Key Management" on page 26
- "Contents of the $(FMSDK) Directory" on page 27
- "SDK Installation Tips" on page 28
- "Protecting Data Storage of FM" on page 28
- "Cprov Function Patching" on page 29
- "FM Message Dispatching" on page 29
- "Handling Host Processes" on page 30
- "Memory Alignment Issues" on page 30
- "Memory Endian Issues" on page 30

Lifecycle Outline

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

Figure 4: FM Development Lifecycle



As shown in the diagram, the flow of development activities is comprised of the following stages:

- "Initial Development" below: Includes the design and development of the functionality application code for initial testing.
- "Emulation Build" below This process compiles the FM code for the emulation environment.
- "Emulation Test" below: The FM produced during the previous steps should now be completely tested in emulation mode to ensure correct operation. Should errors be found in the emulation build, the developer should repeat the build/test/debug cycle until successful operation is confirmed.
- "Adapter Build" below: The functionality module should now be built for the HSM environment. The FM is downloaded onto the HSM hardware and again tested to ensure it operates as expected.
- "Production Build" on the next page: Finally, the FM is produced and released for the operational environment it was intended.

Initial Development

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

Emulation Build

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

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

Emulation Test

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

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

Adapter Build

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

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

Adapter Test

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

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

Production Build

When convinced that the implementation of the FM and the host side code is correct, a production build of the system is performed.

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

Acceptance Test

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

Key Management

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

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

As previously advised, there is no pre-defined key management scheme for the private key and the certificate. The FM developer must decide early on the key management scheme to be used in the system.

Example Key-Management Scheme

This sample approach to a key management scheme can be customized and extended.

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

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

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

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

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

Self-Signed certificate

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

Certificate signed by a trusted third party

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

Use of a local CA signing key

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

Contents of the $(FMSDK) Directory

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

Directory or file	Description
bin/	Non-toolchain utilities and tools for creating an FM
Doc/	FM Development documentation
Include/	Header files specific to FM and FM Host Application development. These are used together with headers provided by ProtectToolkit-C in **$(CPROVDIR)/** include.
lib/ppcfm/	Static libraries used for building an FM that will run in a ProtectServer PCIe HSM.

Directory or file	Description
lib/linux-i386/	Static libraries used for building an emulation mode FM that will run on the host computer.
samples/	Sample FMs and FM Host applications described below.
src/emul/	Source files used for building wrapper **cryptoki** and **ethsm** libraries required with FM emulation.
cfgbuild.mak	Common configuration makefile that sets up makefile and toolchain variables and rules required for building an FM. This should be included at the top of any FM's makefile.

SDK Installation Tips

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

Protecting Data Storage of FM

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

There are three possible solutions to this problem:

- Token blocking
- Using Privilege Level
- Using the SMFS

Token Blocking

As shown in the sample FM "restrict" on page 36, the FM can patch the **C_OpenSession() PKCS#11** function, preventing any session from being open to the slot containing the objects used by the FM. In PKCS#11, all functions that can access or export the object need a session handle to the token containing the object, and the FM patches the **C_OpenSession()** to prevent applications from obtaining the required session handle. This method effectively reserves one or more tokens of the HSM for the FM's internal use, and prevents any kind of access to the contents of this token from the host side.

Using Privilege Level

The **CT_SetPrivilegeLevel** function allows a simple solution. As shown in the sample FM "XORSign" on page 36, 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 keys using the tools provided with the HSM, so they are safe from outside programs but still accessible from the trusted FM.

Using the SMFS

The Secure Memory File System provides access to the same low-level key storage facility. By creating a new application directory, the FM designer can store keys without them being visible through the HSM's Cryptoki interface.

The key format is up to the FM designer - they need not have attributes as Cryptoki objects do.

There is no need to call **C_Initialize**, 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.

When creating FMs that open an SMFS file and keep the handle open, developers should note:

When an application calls **C_Initialize** for the first time after the HSM is rebooted, the HSM firmware will close all SMFS handles. So if you open an SMFS file during startup, the next **C_Initialize** call will close the file. Also, the number of SMFS file handles is a limited resource (approx 16).

Therefore, FM designers should not keep SMFS file handles. Instead, only use SMFS to back up the keys.

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, open/write/close the SMFS file to back up the changes.

Cprov Function Patching

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

One technique is to put safety switches in the **startup** function, as seen in the sample FM "safedebug" on page 36.

FM Message Dispatching

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

The scatter-gather operation on the reply buffers can behave in an unintuitive manner when the initial buffers are of 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. This means 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, the first 40-byte buffer will be filled with 32 bytes of data meant for the first buffer and 8 bytes of data meant for the second buffer. The second reply buffer of 40 bytes will only contain 26 bytes of data.

There are two possible ways to handle this:

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

Handling Host Processes

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

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

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

Therefore, you can remember which process has logged in by storing the PID and the OID as the process successfully authenticates. When a process sends a message that requires authentication, you can check to see if the process is in the list of authenticated processes.

The Cryptoki system always uses the PID/OID to determine if a session handle or object handle is valid for the calling process.

Therefore, if the FM makes Cryptoki calls while processing a request, and it is using a session handle obtained earlier from a different request, there is a possibility that the Cryptoki call with fail with a CKR_CRYPTOKI_NOT_INITIALIZED error.

This occurs because Process A calls the FM, which calls **C_Initialize** and opens a Cryptoki session. 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 PID and OID to a constant value that the underlying Cryptoki sees by using the **FM_SetCurrentPid** and **FM_SetCurrentOid** calls prior to making any Cryptoki calls.

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

Memory Alignment Issues

The PowerPC processor in the ProtectServer Network HSM and ProtectServer PCIe HSM does not require fully aligned memory access, but unaligned access incurs a performance cost.

Memory Endian Issues

The processor in ProtectServer PCIe HSM is big endian, and the legacy PSI-E and PSG processors are little endian.

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

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

CHAPTER 5: Setting up an MSYS environment and cross-compiler

This chapter provides instructions on how to set up an MSYS environment with a cross-compiler that is appropriate for the PCIe hardware, and configuring the environment to correctly build the FM binaries.

The process includes these general steps:

1. Download the required binaries and source code (See "Download MinGW and the toolchain source code" below).
2. Build the cross-compiler (See "Build and install the cross-compiler" on the next page).
3. Add the ProtectServer paths to the MSYS environment (See "Set the MSYS environment to include the FMDIR and CPROVDIR directories" on page 33).

Download MinGW and the toolchain source code

Download the following files:

Filename	Source file	Location
binutils-2.26.tar.gz	binutils 2.26	http://ftp.gnu.org/gnu/binutils/
gcc-5.3.0.tar.gz	gcc 5.3.0	http://ftp.gnu.org/gnu/gcc/gcc-5.3.0/
newlib-2.4.0.tar.gz	newlib 2.4.0	https://sourceware.org/newlib/ Follow the "Download" link and find the source snapshots in the newlib ftp directory.

Extract the **binutils**, **gcc** and **newlib** source files into the same folder. This will be referred to as the **%SRC%** folder. Ensure that the **%SRC%** path has no spaces.

Go to: http://www.mingw.org/ or https://sourceforge.net/projects/mingw/ to download MinGW.

Install MinGW in the default directory (**C:\MinGW**) or a directory of your choice. This directory will be referred to as **%MINGW_HOME%**.

> **NOTE** If you install MinGW in another directory, ensure that the path does not contain spaces. For the automated scripts to work, the **%MINGW_HOME%** environment variable must also be set to the install directory.

Build and install the cross-compiler

There are two options for this process:

> run the provided script that builds and installs the cross-compiler and sets the MinGW environment, or

> complete the steps manually by following the instructions below.

Automated cross-compiler build

1. Copy the files **crossc.cmd** and **crossgcc.mak** from **C:\Program Files\SafeNet\Protect Toolkit 5\FM SDK\gcc_fm** into the **%SRC%** folder.
2. Run **crossc.cmd** from a command prompt.

> **NOTE** If you installed MinGW in another directory other than the default (**C:\MinGW**), set the **%MINGW_HOME%** environment variable to that path before running the script.

Manual cross-compiler build

1. Using either the MinGW GUI or the CLI, install the following packages:
   - mingw32-base
   - mingw32-gcc-g++
   - mingw-developer-toolkit
   - mpc-dev
   - mpfr-dev
   - gmp-dev

   Using the CLI:

   ```
   %MINGW_HOME%\bin\mingw-get install mingw32-base mingw32-gcc-g++ mingw-developer-toolkit mpc-dev mpfr-dev gmp-dev
   ```
2. Go to the **%MINGW_HOME%\msys\1.0\bin** directory and run **bash -c 'mount --replace "%MINGW_HOME%" /mingw'**.

   ```
   cd %MINGW_HOME%\msys\1.0\bin
   bash -c 'mount --replace "%MINGW_HOME%" /mingw'
   ```
3. Open an **MSYS** shell. You might want to create a shortcut on the desktop for easy access. This shell will be used later on to compile the FM binaries.

   ```
   %MINGW_HOME%\msys\1.0\msys.bat
   ```
4. Go to the **%SRC%** folder.

   ```
   cd $SRC
   ```
5. Set up the build environment.

   ```
   export TARGET=powerpc-eabi
   export PREFIX=/mingw
   ```
6. Build the binaries by executing the following **msys** commands, *in this order*:

a. Build **binutils**:

```
tar xvf binutils-2.26.tar.gz
mkdir binutils_build
cd binutils_build
../binutils-2.26/configure --prefix=$PREFIX --target=$TARGET --disable-nls --disable-shared --with-gcc --with-gnu-as --with-gnu-ld --with-stabs --disable-multilib --enable-thmake all-gccreads
make all
make install
cd ..
```

b. Build **gcc - C only**:

```
tar xvf gcc-5.3.0.tar.gz
mkdir gcc_build
cd gcc_build
../gcc-5.3.0/configure --target=$TARGET --prefix=$PREFIX --with-cpu=440fp --enable-languages=c,c++ --disable-multilib --with-gcc --with-gnu-ld --with-gnu-as --with-stabs --disable-shared --enable-threads --disable-nls --with-newlib
make all-gcc
make install-gcc
cd ..
```

c. Build **newlib**:

```
tar xvf newlib-2.4.0.tar.gz
mkdir newlib_build
cd newlib_build
../newlib-2.4.0/configure --target=$TARGET --prefix=$PREFIX
make
make install
cd ..
```

d. Build **gcc - complete**:

```
cd gcc_build
make all
make install
cd ..
```

Set the MSYS environment to include the FMDIR and CPROVDIR directories

> **NOTE** If you installed MinGW in another directory other than the default (**C:\MinGW**), set the **%MINGW_HOME%** environment variable to the install directory you selected.

In the **%FMDIR%\bin** directory, run the **setmsysenv.cmd** script. This will add the environment variables **CPROVDIR** and **FMDIR** to the **MSYS** environment, allowing you to compile the FM binaries. This only needs to be done once.

CHAPTER 6: FM Samples

There are 10 sample FMs provided with the SDK:

> "RSAENC" below
> "XORSign" on the next page
> "restrict" on the next page
> "safedebug" on the next page
> "cipherobj" on the next page
> "smfs" on the next page
> "secfmenc" on the next page
> "ssldemo" on the next page
> "usbdemo" on the next page
> "javahsmreset" on page 37
> "javahsmstate" on page 37

Most have similar file layouts.

> "FM Sample Directories" on page 37
> "FMs in Emulation Mode" on page 39

Included Samples

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

> **NOTE** To avoid running into issues, move samples out of the installation directory before modifying, compiling, or running them.

RSAENC

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

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

Note that for running this sample TEST_RSA_KEY RSA key must be created. This may be achieved with the following command:

```
ctkmu c -nTEST_RSA_KEY -trsa -z1024 -aEDSUPT
```

XORSign

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

The new mechanism is bitwise exclusive OR with name CKM_XOR.

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

```
xortest -g
```

restrict

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

safedebug

This sample demonstrates how to use SMFS storage to determine if your FM should abort its startup.

cipherobj

This sample demonstrates how to access primitive cryptographic services by using the Cipher Object system. Note that to run this sample, TEST_DES3_KEY DES3 key must be created with the following command:

```
ctkmu c -nTEST_DES3_KEY -tdes3 -aEDSVT
```

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

smfs

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

secfmenc

This sample demonstrates the use of the **FMSC_SendReceive()** function to call custom FMs using the Cryptoki library (see FMSC_SendReceive in the "PKCS#11 Command Reference" section of the *ProtectToolkit-C Programming Guide*). It can perform RSA as well as DES3 encryption.

To run an RSA test, you must first have an RSA key with the label TEST_RSA_KEY stored on the HSM.

To run a DES3 test, you must first have a DES3 key with the label TEST_DES3_KEY stored on the HSM.

ssldemo

This sample is included as a reference for how to use the Big Number library (**libfmbn**) with your FMs.

usbdemo

This sample allows applications written using the USB API to interact with the HSM. You must load the **usbdemo** FM sample to use this functionality. See "USB API Reference" on page 132 for the available calls.

NOTE

> This sample is included for Linux clients only.

> Warnings appear while compiling this sample. These warnings can be safely ignored.

javahsmreset

This function does not make FMs.

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

javahsmstate

This function does not make FMs.

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

FM Sample Directories

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

> **Makefile**: makefile to build host and HSM side code

> **Fm**: directory holding HSM side source

> **Host**: directory holding host side source

> **Include**: optional directory to hold common header files

Within the FM directory are files like these -

> **hdr.c**: header file for the production build of the FM binary image.

> **sample.c**: HSM side; main source for FM

> **Makefile**: Makefile to build the FM and the application

Within the host directory are files like this -

> **stub_sample.c**: host side stub (request encoder/decoders) (needed only for custom API)

> **sample.c**: main source for host side test application

> **Makefile**: Makefile to build the host-side application for emulation, or production.

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

> Production build, no debug information in binaries:

  make

> Production build, with debug information in binaries:

  make DEBUG=1

> Emulation build, no debug information in binaries:

  make EMUL=1

> Emulation build, with debug information in binaries:

  make EMUL=1 DEBUG=1

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

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

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

make EMUL=1 clean

Signing Sample FMs

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 download them to the HSM.

To sign sample FMs

To use the key management scheme #1 (using self-signed FM download certificates), follow the steps listed below.

1. Generate the key/certificate pair on the first token. From a command prompt, execute:

   ctcert c -s0 -k -trsa -z2048 -lfm

   This will generate a 2048 bit RSA key pair, and a self-signed certificate. The minimum key size for FM signing is 2048 bits. The labels of the generated keys are as shown below:

   Private Key: **fm (Pri)**

   Public Key: **fm (Pub)**

   Certificate: **fm**

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

   ctcert x -lfm -s0 -ffmcert.crt

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

ctcert i -ffmcert.crt -s2 -lfm

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

   ctcert t -lfm -s2

   Please refer to CTCERT in the "Command Line Utilities Reference" section of the *ProtectToolkit-C Administration Guide* for a full account of the **ctcert** utility.

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

   mkfm -k “*<TokenLabel>***/***<fm (Pri)>***” -fsampleN -osampleN.fm**

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

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

   ctconf -b*<fm>* **-jsampleN.fm**

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

   ctconf -s

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

FMs in Emulation Mode

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

1. Initialize the slot (0):

   ctconf -n0

2. Initialize the slot's user PIN:

   ctkmu p -s0

3. List slots:

   ctkmu l

Emulation builds and test steps

```
make EMUL=1
cdfm/obj-linux-i386e
sampleN
```

Adapters builds and test steps

```
make
cd obj-ppcfm
mkfm -k devel_key -f sampleN -o sampleN.fm
ctconf -j sampleN.fm
sampleN
```

CHAPTER 7: Building sample FMs in emulation mode on Windows

This chapter provides instructions on how to compile the sample FM projects in the Windows environment.

The process consists of the following steps:

1. Copy the samples and emulation source folders.
2. Set the environment variables.
3. Compile the binaries.

Copy the samples and emulation source folders

Microsoft recommends against creating and editing files in the Program Files folder. To avoid running into issues with the UAC, copy the samples folder and the emul folder to a separate folder. These will be referred to as the **%SRC%** and **%EMUL%** folders respectively.

Set the environment variables

1. Open a Visual Studio Command Prompt to load the Visual studio compiler.
2. Run the **fmsdkvars.bat** file found in the FM-SDK installation folder to set the *%FMDIR%* and *%CPROVDIR%* environment variables.

   ```
   %FMDIR%\bin\fmsdkvars.bat
   ```

3. Set the **%OUTDIR%** environment variable. This is the folder where the emulation libraries will be created.

   ```
   set OUTDIR=c:\fmdemo
   ```

4. Set the **%FM_BIN%** environment variable. This is the name of the FM emulation library.

   ```
   set FM_BIN=fm-restrict
   ```

Compile the binaries

1. Navigate to the sample folder's FM directory (e.g. **%SRC%\restrict\fm**) and run **nmake -f nt.mak**

   ```
   cd %SRC%\restrict\fm
   ```

   ```
   nmake -f nt.mak
   ```

2. Navigate to the sample folder's host directory (e.g. **%SRC%\restrict\host**) and run **nmake -f nt.mak**

   ```
   cd %SRC%\restrict\host
   ```

   ```
   nmake -f nt.mak
   ```

3. Navigate to the **%EMUL%** folder and run **nmake -f nt.mak**

```
cd %EMUL%

nmake -f nt.mak
```

4. Navigate to the **%OUTDIR%** folder:

```
cd %OUTDIR%
```

5. In there, you should have the following files:
   - **%FM_BIN%.dll**
   - **%CLIENT_BIN%.exe**
   - **cryptoki.dll**
   - **ethsm.dll**
6. Run the executable to test the FM.

CHAPTER 8: FM Deployment

This chapter provides an overview of how to deploy an FM on an HSM device. The ProtectToolkit-C Administrator can deploy FMs using the Functionality Module Manager (**ctfm**). For more information about the **ctfm** utility, see CTFMin the "Command Line Utilities Reference" section of the *ProtectToolkit-C Administration Guide*.

It contains the following sections:

> "Deploy a Single FM" below

> "Deploy Multiple FMs" on the next page

> **NOTE** Only one PKCS#11 patched FM can be loaded and used at a time. If a patched FM is already loaded, it is overwritten by the new FM. Refer to "Custom Functions" on page 19 and "PKCS#11 Patched Functions" on page 20 for more information about these FM types.

Deploy a Single FM

To use an FM you must sign the FM binary and load the signed **.fm** file onto the HSM. The process consists of the following:

1. Compile the FM using **make**.
2. Sign the compiled FM using **mkfm**.
3. Load the FM onto the HSM using **ctfm**.

> **NOTE** Thales recommends deploying FMs in a test environment before delivering to a production environment.

To compile the FM

1. Navigate to the directory where the FM binary components are stored.
2. Use **make** to compile the FM.

To sign a compiled FM

1. Navigate to the directory containing the FM binary image.
2. Use **mkfm** to sign the FM and generate a **.fm** file.

   mkfm -kSLOTID=<HSMslotID>**/**<PrivateKeyLabel> **-f** <fmBinary> **-0** <fmFile>**.fm**

> **NOTE**
> > The output file name must be a **.fm** file.
> > The slot must have a token name implemented or the **mkfm** command will fail.
> > The minimum key size for FM signing is 2048 bits.

3. Use **ctcert** to import the certificate into the Admin slot

   ctcert i -f<certificate_file> **-l**<certificate_label> [**-s**<admin_slot>]

4. Use **ctcert** to make the certificate trusted.

   ctcert t -l<certificate_label> [**-s**<admin_slot>]

To load the FM onto the HSM

1. Navigate to the directory where the **.fm** file is stored. Use **ctfm** to load the **.fm** onto the HSM.

   ctfm i -a<device> **-l** <certLabel> **-f**<.fmFile>

2. Run **ctfm q -a**<device> to confirm that the FM was loaded successfully.

   ctfm q -a<device>

Deploy Multiple FMs

To deploy multiple FMs you must sign the FM binary and load the signed **.fm** file onto the HSM. Note the following:

> Each FM must have a unique FMID.
> The FMID must be compiled into the FM as part of the FM image.
> You must use the FMID when calling the FM.

Configuring a unique FMID

Before you compile your FM, you must assign a unique FMID to it by defining the MY_FM_NUMBER macro in the **hdr.c** file. Subsequently, you compile the FM using **make**, sign the FM binary, and load the **.fm** file onto the HSM.

To configure a unique FMID

1. Open the FM **hdr.c** file in a text editor. Add the following line to define the MY_FM_NUMBER value.

```
#define MY_FM_NUMBER <FMID>
```

> **NOTE** The MY_FM_NUMBER value must be input as a hexadecimal value from 0x100 to 0xffff (values below 0x100 are unavailable). Subsequent calls to the FM must be directed towards the FMID value. For example, an **hdr.c** MY_FM_NUMBER value of 0x100 results in an FMID of 256. The **ctfm** and **ctconf** utilities display FMIDs in hex mode.

Calling FMID

On an HSM device with multiple loaded FMs you must direct calls towards the unique FMID. Adhere to the following:

- You must use the FMID value to register any dispatch using FMSW_RegisterDispatch.
- You must use the FMID value for any host application such as MD_SendReceive.
- You must use the correct FMID when registering a custom command handler in the FM startup.

Examples

This example demonstrates an FM that has been assigned an FMID of 0x300.

- Configuration of the FM **hdr.c** file.

```
#define MY_FM_NUMBER 0x300

DEFINE_FM_HEADER(MY_FM_NUMBER,
FM_VERSION,
FM_SER_NO,
FM_MANUFACTURER,
FM_NAME);
```

- Using the FMID value in the host application MD_SendReceive.

```
rv = MD_SendReceive( adapter,
originatorID,
MY_FM_NUMBER,
request,
RESERVED,
reply,
&recvLen,
&appState);
```

CHAPTER 9: Utilities Reference

This section contains information pertaining to the following utilities:

> "ctcert" below

> "ctconf" below

> "ctfm" below

> "mkfm" below

ctcert

The **ctcert** utility is provided as a part of the ProtectToolkit-C package. For more information about the **ctcert** utility, see CTCERT in the "Command Line Utilities Reference" section of the *ProtectToolkit-C Administration Guide*.

ctconf

The **ctconf** utility is provided as a part of the ProtectToolkit-C package. For more information about the **ctconf** utility, see CTCONF in the "Command Line Utilities Reference" section of the *ProtectToolkit-C Administration Guide*.

ctfm

The **ctfm** utility is provided as a part of the ProtectToolkit-C package. For more information about the **ctfm** utility, see CTFM in the "Command Line Utilities Reference" section of the *ProtectToolkit-C Administration Guide*.

mkfm

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

Synopsis

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

Option	Description
`-f<filename>`	**--input-file=<filename>** This specifies the relative or full path to the FM binary image.

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

CHAPTER 10: Cipher Object

The PKCS #11 API provides a standard method for accessing and managing keys, and performing cryptographic operations. Providing a system-independent layer, however, introduces a considerable amount of overhead.

ProtectToolkit provides an internal API which bypasses the PKCS #11 subsystem to perform high-performance cryptographic functions.

The Cipher Object Access API

Cryptographic operations require that you obtain a pointer to an instance of a *cipher object* or a *hash object*. A cipher object may be used to encrypt, decrypt, sign (or MAC), or verify data. A hash object is used to perform a digest operation. There is a function for obtaining an instance of each of these objects.

This chapter provides details on Cipher Objects:

> "FmCreateCipherObject" on the next page
> "Cipher Object Functions" on page 50
> "Algorithm-Specific Cipher Information" on page 67

See "Hash Object" on page 79 for information on Hash Objects.

FmCreateCipherObject

This function constructs and initializes a Cipher Object of the specified type.

Synopsis

```
#include "fmciphobj.h"
CipherObj * FmCreateCipherObject(FMCO_CipherObjIndex index);
```

Option	Description
`index`	The type of cipher object requested. It can have the following values (defined in **fmciphobj.h**): > **FMCO_IDX_AES**: Implementation of the AES (Rijndael) algorithm > **FMCO_IDX_CAST**: Implementation of the CAST algorithm > **FMCO_IDX_IDEA**: Implementation of the IDEA algorithm > **FMCO_IDX_RC2**: Implementation of the RC2 algorithm > **FMCO_IDX_RC4**: Implementation of the RC4 algorithm > **FMCO_IDX_DES**: Implementation of the single DES algorithm > **FMCO_IDX_TRIPLEDES**: Implementation of the triple DES (with either double or triple length keys) algorithm > **FMCO_IDX_DSA**: Implementation of the DSA algorithm (signing and verification only) > **FMCO_IDX_ECDSA**: Implementation of the ECDSA algorithm (signing and verification only) > **FMCO_IDX_HMACMD2**: Implementation of the HMAC construct using MD2 hash algorithm (signing and verification only) > **FMCO_IDX_HMACMD5**: Implementation of the HMAC construct using MD5 hash algorithm (signing and verification only) > **FMCO_IDX_HMACSHA1**: Implementation of the HMAC construct using SHA-1 hash algorithm (signing and verification only) > **FMCO_IDX_HMACRMD128**: Implementation of the HMAC construct using RIPEMD-128 hash algorithm (signing and verification only) > **FMCO_IDX_HMACRMD160**: Implementation of the HMAC construct using RIPEMD-160 hash algorithm (signing and verification only) > **FMCO_IDX_RSA**: Implementation of the RSA algorithm (only single-part operations supported) This list is correct at time of writing; the actual number of objects supported depends on the HSM firmware version.

Return Value

The address of the cipher object is returned. If the system does not have enough memory to complete the operation, NULL is returned.

Comments

The returned Cipher Object should be freed by calling its **Free()** function (See "Free" on page 51).

The Cipher Objects may have some of the function's addresses set to NULL to indicate that they are not implemented. It is recommended that the function address be checked before using these functions. Please consult the list under "Cipher Object" on page 48 for details.

> **NOTE** It is the Operating System firmware that provides the CipherObject - not the FM SDK. As new versions of OS firmware are developed and released, more Cipher Objects may be added to the list of supported algorithms.

A firmware upgrade may be required to obtain a particular Cipher Algorithm.

Cipher Object Functions

The Cipher Object is a wrapper that provides a common interface for all supported cryptographic algorithms. It is implemented as a structure containing the addresses of functions, as well as a data pointer that keeps the internal state of the instance. The contents of the data field are private, and should not be accessed or modified externally.

In this section, the following functions in the cipher object are specified:

- "New" on the next page
- "Free" on the next page
- "GetInfo" on the next page
- "EncInit" on page 52
- "EncryptUpdate" on page 53
- "EncryptFinal" on page 54
- "DecInit" on page 55
- "DecryptUpdate" on page 56
- "DecryptFinal" on page 56
- "SignInit" on page 57
- "SignUpdate" on page 58
- "SignFinal" on page 58
- "SignRecover" on page 59
- "VerifyInit" on page 60
- "VerifyUpdate" on page 61
- "VerifyFinal" on page 61
- "VerifyRecover" on page 62
- "Verify" on page 63
- "LoadParam" on page 64
- "UnloadParam" on page 65
- "Config (Obsolete)" on page 65

> "Status (Obsolete)" on page 66
> "EncodeState (Obsolete)" on page 66
> "DecodeState (Obsolete)" on page 67

New

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

Synopsis

```
#include "fmciphobj.h"
struct CipherObj * (*New)(struct CipherObj * ctx);
```

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

Return Value

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

Comments

The returned Cipher Object should be freed by calling its **Free()** function (see "Free" below).

Free

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

Synopsis

```
#include "fmciphobj.h"
int (*Free)(struct CipherObj * ctx);
```

Parameter	Description
ctx	The address of the cipher object to be freed.

Return Value

None

GetInfo

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

Synopsis

```
#include "fmciphobj.h"
int (*GetInfo)(struct CipherObj * ctx, struct CipherInfo * info);
```

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

Return Value

0: Operation completed successfully.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

Comments

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

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

The field meanings are:

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

EncInit

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

Synopsis

```
#include "fmciphobj.h"
int (*EncInit)(struct CipherObj * ctx,
int mode,
const void * key, unsigned int klength,
const void * param, unsigned int plength);
```

Parameter	Description
ctx	The address of a cipher object instance.
mode	The encrypt mode. Different algorithms support different values for this parameter. Please consult "Algorithm-Specific Cipher Information" on page 67 for the possible values for a certain algorithm.
key	The address of a buffer containing the key value. The encoding of the key is algorithm-dependent. However, for most block ciphers, this buffer contains the key value. Please consult "Algorithm-Specific Cipher Information" on page 67 for key encoding information.
klength	Number of bytes in the key buffer.
param	The address of the buffer containing various parameters for the encrypt operation. The contents and the encoding of this buffer are algorithm and mode dependent. However, for most block ciphers this buffer contains the IV when one of the CBC modes is used. Please consult "Algorithm-Specific Cipher Information" on page 67 for key encoding information.
plength	Number of bytes in the param buffer.

Return Value

0: The operation was successful.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

EncryptUpdate

This function is used to encrypt some data. It can be used in two ways: discovering the output buffer length, or performing encryption.

If the target buffer address, *tgt*, is NULL, then the variable pointed to by *plen* is updated to contain the length of the output that is required to perform the operation.

If the target buffer address is not NULL, then the currently buffered data and the input buffer contents are combined, and as many blocks as possible are encrypted. The result of the encrypted blocks are placed in the output buffer. If there are any remaining bytes, they are internally buffered. The number of bytes placed in the target buffer is also written to the variable pointed by *plen*.

Synopsis

```
#include "fmciphobj.h"
int (*EncryptUpdate)(struct CipherObj * ctx,
void * tgt, unsigned int tlength, unsigned int * plen,
const void * src, unsigned int length);
```

Parameter	Description
ctx	The address of a cipher object instance.

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

Return Value

0: Operation completed successfully.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

EncryptFinal

This function must be called to finish an encryption operation. It can be used for either discovering the target buffer length, or for actually performing the operation.

If the target buffer address, *tgt*, is NULL, then the variable pointed to by *plen* is updated to contain the length of the output that is required to perform the operation.

If the target buffer address is not NULL, then the currently buffered data is padded (if the mode permits it), and encrypted. The result is placed in the *tgt* buffer. The number of bytes placed in the target buffer is also written to the variable pointed by *plen*. If the current mode does not allow padding, and there is buffered data, then an error is returned.

Synopsis

```
#include "fmciphobj.h"
int (*EncryptFinal)(struct CipherObj * ctx,
void * tgt, unsigned int tlength, unsigned int * plen);
```

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

Return Value

0: Operation completed successfully.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

Comments

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

DecInit

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

Synopsis

```
#include "fmciphobj.h"
int (*DecInit)(struct CipherObj * ctx,
int mode,
const void * key, unsigned int klength,
const void * param, unsigned int plength);
```

Parameter	Description
`ctx`	The address of a cipher object instance.
`mode`	The encrypt mode. Different algorithms support different values for this parameter. Please consult "Algorithm-Specific Cipher Information" on page 67 for the possible values for a certain algorithm.
`key`	The address of a buffer containing the key value. The encoding of the key is algorithm-dependent. However, for most block ciphers, this buffer contains the key value. Please consult "Algorithm-Specific Cipher Information" on page 67 for key encoding information.
`klength`	Number of bytes in the key buffer.
`param`	The address of the buffer containing various parameters for the encrypt operation. The contents and the encoding of this buffer are algorithm and mode dependent. However, for most block ciphers this buffer contains the IV when one of the CBC modes is used. Please consult "Algorithm-Specific Cipher Information" on page 67 for key encoding information.
`plength`	Number of bytes in the param buffer.

Return Value

0: The operation was successful.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

DecryptUpdate

This function is used to decrypt some data. This function can be used in two ways: discovering the output buffer length, or performing decryption.

If the target buffer address, *tgt*, is NULL, then the variable pointed to by *plen* is updated to contain the length of the output that is required to perform the operation.

If the target buffer address is not NULL, then the currently buffered data and the input buffer contents are combined, and as many blocks as possible are decrypted. The result of the decrypted blocks are placed in the output buffer. If there are any remaining bytes, they are internally buffered. The number of bytes placed in the target buffer is also written to the variable pointed by *plen*.

Synopsis

```
#include "fmciphobj.h"
int (*DecryptUpdate)(struct CipherObj * ctx,
void * tgt, unsigned int tlength, unsigned int * plen,
const void * src, unsigned int length);
```

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

Return Value

0: Operation completed successfully.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

DecryptFinal

This function must be called to finish a decryption operation. It can be used for either discovering the target buffer length, or for actually performing the operation.

If the target buffer address, *tgt*, is NULL, then the variable pointed to by *plen* is updated to contain the length of the output that is required to perform the operation.

If the target buffer address is not NULL, and the mode is padded mode, the final block is decrypted and the padding bytes are removed before they are placed in the *tgt* buffer. If there is not exactly one block of data in the buffer, an error is returned.

Synopsis

```
#include "fmciphobj.h"
int (*DecryptFinal)(struct CipherObj * ctx,
void * tgt, unsigned int tlength, unsigned int * plen);
```

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

Return Value

0: Operation completed successfully.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

Comments

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

SignInit

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

Synopsis

```
#include "fmciphobj.h"
int (*SignInit)(struct CipherObj * ctx,
int mode,
const void * key, unsigned int klength,
const void * param, unsigned int plength);
```

Parameter	Description
ctx	The address of a cipher object instance.
mode	The encrypt mode. Different algorithms support different values for this parameter. Please consult "Algorithm-Specific Cipher Information" on page 67 for the possible values for a certain algorithm.
key	The address of a buffer containing the key value. The encoding of the key is algorithm-dependent. However, for most block ciphers, this buffer contains the key value. Please consult "Algorithm-Specific Cipher Information" on page 67 for key encoding information.

Parameter	Description
klength	Number of bytes in the key buffer.
param	The address of the buffer containing various parameters for the encrypt operation. The contents and the encoding of this buffer are algorithm and mode dependent. However, for most block ciphers this buffer contains the IV when one of the CBC modes is used. Please consult "Algorithm-Specific Cipher Information" on page 67 for key encoding information.
plength	Number of bytes in the param buffer.

Return Value

0: The operation was successful.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

SignUpdate

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

Synopsis

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

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

Return Value

0: Operation completed successfully.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

Comments

Usually, only block cipher algorithms implement this function.

SignFinal

This function must be called to finish a signing or MAC operation. It can be used for either discovering the target buffer length, or for actually performing the operation.

If the target buffer address, *tgt*, is NULL, then the variable pointed to by *plen* is updated to contain the length of the output that is required to perform the operation.

If the target buffer address is not NULL, then the signing operation is completed, and the signature is placed in the output buffer.

Synopsis

```
#include "fmciphobj.h"
int (*SignFinal)(struct CipherObj * ctx,
void * tgt, unsigned int tlength, unsigned int * plen);
```

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

Return Value

0: Operation completed successfully.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

Comments

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

SignRecover

This function implements a single-part signing or MAC operation. It can be used for either discovering the target buffer length, or for actually performing the operation.

If the target buffer address, *tgt*, is NULL, then the variable pointed to by *plen* is updated to contain the length of the output that is required to perform the operation.

If the target buffer address is not NULL, then the signing operation is performed, and the signature is placed in the output buffer.

Synopsis

```
#include "fmciphobj.h"
int (*SignRecover)(struct CipherObj * ctx,
void * tgt, unsigned int tlength, unsigned int * plen,
const void * src, unsigned int length);
```

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

Return Value

0: The operation was successful.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

Comments

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

VerifyInit

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

Synopsis

```
#include "fmciphobj.h"
int (*VerifyInit)(struct CipherObj * ctx,
int mode,
const void * key, unsigned int klength,
const void * param, unsigned int plength);
```

Parameter	Description
ctx	The address of a cipher object instance.
mode	The encrypt mode. Different algorithms support different values for this parameter. Please consult "Algorithm-Specific Cipher Information" on page 67 for the possible values for a certain algorithm.
key	The address of a buffer containing the key value. The encoding of the key is algorithm-dependent. However, for most block ciphers, this buffer contains the key value. Please consult "Algorithm-Specific Cipher Information" on page 67 for key encoding information.

Parameter	Description
klength	Number of bytes in the key buffer.
param	The address of the buffer containing various parameters for the encrypt operation. The contents and the encoding of this buffer are algorithm and mode dependent. However, for most block ciphers this buffer contains the IV when one of the CBC modes is used. Please consult "Algorithm-Specific Cipher Information" on page 67 for key encoding information.
plength	Number of bytes in the param buffer.

Return Value

0: The operation was successful.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

VerifyUpdate

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

Synopsis

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

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

Return Value

0: Operation completed successfully.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

Comments

Usually, only block cipher algorithms implement this function.

VerifyFinal

This function must be called to finish a signature or MAC verification operation. It can be used for either discovering the target buffer length, or for actually performing the operation.

If the target buffer address, *tgt*, is NULL, then the variable pointed to by *plen* is updated to contain the length of the output that is required to perform the operation.

If the target buffer address is not NULL, then the verification operation is completed. In addition to the verification, the recovered signature is placed in the output buffer.

Synopsis

```
#include "fmciphobj.h"
int (*VerifyFinal)(struct CipherObj * ctx,
const void * sig, unsigned int slength,
void * tgt, unsigned int tlength, unsigned int * plen);
```

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

Return Value

0: The signature or MAC was correct.

Otherwise, an error occured or the signature or MAC was incorrect. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

Comments

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

VerifyRecover

This function implements a single-part signature or MAC verification operation with recovery. It can be used for either discovering the target buffer length, or for actually performing the operation.

If the target buffer address, *tgt*, is NULL, then the variable pointed to by *plen* is updated to contain the length of the output that is required to perform the operation.

If the target buffer address is not NULL, then the verification operation is performed. In addition to the verification, the recovered signature is placed in the output buffer.

Synopsis

```
#include "fmciphobj.h"
int (*VerifyRecover)(struct CipherObj * ctx,
const void * sig, unsigned int slength,
void * tgt, unsigned int tlength, unsigned int * plen,
const void * src, unsigned int length);
```

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

Return Value

0: The operation was successful.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

Comments

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

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

Verify

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

Synopsis

```
#include "fmciphobj.h"
int (*Verify)(struct CipherObj * ctx,
const void * sig, unsigned int slength,
const void * src, unsigned int length);
```

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

Return Value
0: The operation was successful.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

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

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

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

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

Parameter	Description
ctx	The address of a cipher object instance.
param	The address of the buffer containing various parameters for the encrypt operation. The contents and the encoding of this buffer are algorithm and mode dependent. However, for most block ciphers this buffer contains the IV when one of the CBC modes is used. Please consult "Algorithm-Specific Cipher Information" on page 67 for key encoding information.
length	Number of bytes in the param buffer.

Return Value
0: The operation was successful.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

Comments

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

UnloadParam

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

Synopsis

```
#include "fmciphobj.h"
int (*UnloadParam)(struct CipherObj * ctx,
void * param, unsigned int length, unsigned int *plen);
```

Parameter	Description
`ctx`	The address of a cipher object instance.
`param`	The address of the buffer containing various parameters for the encrypt operation. The contents and the encoding of this buffer are algorithm and mode dependent. However, for most block ciphers this buffer contains the IV when one of the CBC modes is used. Please consult "Algorithm-Specific Cipher Information" on page 67 for key encoding information.
`length`	Number of bytes in the param buffer.
`plen`	Address of a variable that will receive the number of bytes placed in the param buffer. This variable must not be NULL.

Return Value

0: The operation was successful.

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on page 67.

Comments

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

Config (Obsolete)

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

Synopsis

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

Parameter	Description
ctx	The address of a cipher object instance.
parameters	The address of the buffer that contains the information returned from the Status function.
length	Number of bytes in the parameters buffer.

Return Value
0: Operation was successful

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on the next page.

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

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

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

Parameter	Description
ctx	The address of a cipher object instance.
parameters	The address of the buffer that contains the configuration information.
length	Number of bytes in the parameters buffer.

Return Value
0: Operation was successful

Otherwise, an error occured. For descriptions of various error codes generated by cipher object functions, see "Cipher Object Function Error Codes" on the next page.

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

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

DecodeState (Obsolete)

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

Cipher Object Function Error Codes

The table below lists the error codes that may be returned from cipher object functions.

Name	Value	Description
CO_PARAM_INVALID	0x0001	Some input value is invalid or missing.
CO_SIG_INVALID	0x0002	Bad signature during verify operation.
CO_LENGTH_INVALID	0x0003	Invalid length of data or parameter i.e. not multiple of block.
CO_DEVICE_ERROR	0x0004	An unexpected internal error that may be caused by bad input data.
CO_GENERAL_ERROR	0x0005	An unexpected internal error that may be caused by bad input data.
CO_MEMORY_ERROR	0x0006	**malloc()** failed.
CO_BUFFER_TOO_SMALL	0x0007	Output buffer supplied to command is too small.
CO_DATA_INVALID	0x0008	Input data badly formatted.
CO_NEED_IV_UPDATE	0x0009	Not used.
CO_NOT_SUPPORTED	0x000A	CiphObj member function not implemented. For example, HMAC Encrypt.
CO_DUPLICATE_IV_FOUND	0x000B	Only possible with DES3 output feedback (OFB) mode.
CO_FIPSG_ERROR	0x000C	Not used.
CO_FUNCTION_NOT_IMPLEMENTED	0x000D	CiphObj member function not implemented. For example,HMAC Encrypt.
CO_POINT_INVALID	0x000E	Public key is not valid for the ECC curve.

Algorithm-Specific Cipher Information

This section contains the following descriptions:

> "AES Cipher Object" on the next page

> "DES Cipher Object" on page 69

> "Triple DES Cipher Object" on page 70

- "ECDSA Cipher Object" on page 72
- "IDEA Cipher Object" on page 73
- "RC2 Cipher Object" on page 74
- "RC4 Cipher Object" on page 75
- "RSA Cipher Object" on page 75

AES Cipher Object

Operation Supported : Encrypt, Decrypt, Multi-part MAC, Multi-part MAC Verify

Key Encoding

Supports 16, 24 and 32 byte key value.

Encrypt/Decrypt Modes

The least significant nibble (4 bits) determines the operational mode.

Possible values:

- **SYM_MODE_ECB (0)**

  Electronic Code Book (ECB) mode.
- **SYM_MODE_CBC (1)**

  Cipher Block Chaining (CBC) mode. It may be combined with a padding mode (see below).
- **SYM_MODE_CFB (2)**

  Cipher Feedback (64-bit) mode
- **SYM_MODE_BCF (3)**

  Byte Cipher Feedback (8-bit CFB) mode
- **SYM_MODE_OFB (4)**

  Output Feed Back (OFB) mode.
- **SYM_MODE_WRAP_KWP (7)**

  Key Wrap with Padding (KWP) mode
- **SYM_MODE_GCM (9)**

  Galois Counter Mode (GCM)
- **SYM_MODE_CCM (10)**

  Counter with CBC-MAC (CCM) mode

Padding Mode

The most significant nibble defines the padding mode used.

Possible values:

- **SYM_MODE_PADPKCS1 (0x80)**

PKCS#1 padding is applied. This causes 1 to 8 bytes of padding to be added to the data. Note that the padding is applicable to SYM_MODE_CBC only.

> **SYM_MODE_PADCMAC (0x40)**

CMAC padding is applied. This causes 1 to 7 bytes of padding to be added to the data.

MAC modes

For MAC generation and verification, available modes include:

> **SYM_MODE_MAC_3 (0)**

Standard CBC

> **SYM_MODE_MAC_GEN (1)**

Standard CBC with configurable MAC length

Encrypt/Decrypt Parameters

In SYM_MODE_CBC, the parameter buffer must contain the IV (16 bytes). In SYM_MODE_ECB, there are no parameters.

MAC Parameters

When mode is SYM_MODE_MAC_GEN, parameter buffer contains at least 4 bytes, which is the little-endian encoding of an integer. The integer value must contain a value from 1 to 8, indicating the number of bytes of the final IV that will be used as the MAC. This is optionally followed by 8 bytes containing the IV.

DES Cipher Object

Operations Supported

Encrypt, Decrypt, Multi-Part MAC, and Multi-Part Verify.

Key encoding

Every byte contains 7 key bits, and 1 parity bit. The parity bit is the least significant bit in each byte. There is no additional encoding of the key data. The key must be 8 bytes long.

Encrypt/Decrypt Modes

The least significant nibble (4 bits) determines the operational mode.

Possible values:

> **SYM_MODE_ECB (0)**

Electronic Code Book (ECB) mode. It may be combined with a padding mode (see below).

> **SYM_MODE_CBC (1)**

Cipher Block Chaining (CBC) mode. It may be combined with a padding mode (see below).

> **SYM_MODE_CFB (2)**

Cipher Feedback (64-bit) mode

> **SYM_MODE_BCF (3)**

Byte Cipher Feedback (8-bit CFB) mode

> **SYM_MODE_OFB (4)**

Output Feedback (64-bit) mode

> **SYM_MODE_BOF (5)**

Byte Output Feedback (8-bit OFB) mode

The most significant nibble defines the padding mode used.

Possible values:

> **SYM_MODE_PADPKCS1 (0x80)**

PKCS#1 padding is applied. This causes 1 to 8 bytes of padding to be added to the data.

> **SYM_MODE_PADCMAC (0x40)**

CMAC padding is applied. This causes 1 to 7 bytes of padding to be added to the data.

MAC modes

For MAC generation and verification operation the following modes are available:

> **0**: Standard DES CBC

> **1**: Standard DES CBC with configurable MAC length

In both methods, NULL padding is applied to the data.

Encrypt/Decrypt Parameters

In all modes, except SYM_MODE_ECB, the parameter buffer must contain the IV (8 bytes). In SYM_MODE_ECB, there are no parameters.

MAC Parameters

When mode 1 is used, parameter buffer contains 4 bytes, which contain a little-endian encoding of an integer. The integer value must contain a value from 1 to 8, indicating the number of bytes of the final IV that will be used as the MAC.

Triple DES Cipher Object

Operations Supported

Encrypt, Decrypt, Multi-Part MAC, and Multi-Part Verify.

Key encoding

Every byte contains 7 key bits, and 1 parity bit. The parity bit is the least significant bit in each byte. There is no additional encoding of the key data. The key must be 16 or 24 bytes.

Encrypt/Decrypt Modes

The least significant nibble (four bits) determines the operational mode. Possible values:

> **SYM_MODE_ECB (0)**

Electronic Code Book (ECB) mode. It may be combined with a padding mode (see below).

- **SYM_MODE_CBC (1)**

  Cipher Block Chaining (CBC) mode. It may be combined with a padding mode (see below).
- **SYM_MODE_CFB (2)**

  Cipher Feedback (64-bit) mode
- **SYM_MODE_BCF (3)**

  Byte Cipher Feedback (8-bit CFB) mode
- **SYM_MODE_OFB (4)**

  Output Feedback (64-bit) mode
- **SYM_MODE_BOF (5)**

  Byte Output Feedback (8-bit OFB) mode
- **SYM_MODE_WRAP_TKW (8)**

  Triple DES Key Wrap (TKW) mode

The most significant nibble determines the padding mode. Possible values:

- **SYM_MODE_PADPKCS1 (0x80)**

  PKCS#1 padding is applied. This causes 1 to 8 bytes of padding to be added to the data.
- **SYM_MODE_PADCMAC (0x40)**

  CMAC padding is applied. This causes 1 to 7 bytes of padding to be added to the data.

MAC modes

For MAC generation and verification, available modes include:

- **0**: Standard triple DES CBC
- **1**: Standard triple DES CBC with configurable MAC length
- **2**: X9.19 triple DES CBC
- **3**: X9.19 triple DES CBC with configurable MAC length
- **4**: Retail CFB MAC.

In all methods, NULL padding is applied to the data.

Encrypt/Decrypt Parameters

In all modes except SYM_MODE_ECB, the parameter buffer must contain the IV (8 bytes). In SYM_MODE_ECB, there are no parameters.

MAC Parameters

When mode is either 1 or 3, parameter buffer contains at least 4 bytes, which is the little-endian encoding of an integer. The integer value must contain a value from 1 to 8, indicating the number of bytes of the final IV that will be used as the MAC. This is optionally followed by 8 bytes containing the IV. For mode 4 (Retail MAC CFB), the parameter buffer must have 8 bytes containing the encrypted IV. For the remaining two modes, the parameter buffer is either empty, or has 8 bytes containing the IV.

ECDSA Cipher Object

Operations Supported
SignInit(), Sign(), VerifyInit(), and Verify().

Key Encoding
When performing:

- **Sign** operation: the key is specified as a buffer of ECC_Curve_t followed by Private Key ECC_PrivateKey_t.
- **Verify** operation: the key is specified as a buffer of ECC_Curve_t followed by Public Key ECC_PublicKey_t.

See also "ECDSA Key Structures" below.

Modes
None

Sign/Verify Parameters
None

ECDSA Key Structures
```
#define ECC_MAX_MOD_LEN 571
#define ECC_MAX_BUF_LEN ROUND_UP(ECC_MAX_MOD_LEN, 8)/8

typedef enum ECC_FieldType_et {
        ECC_FT_GFP,
        ECC_FT_G2M
        ECC_FT_MON
} ECC_FieldType_t;
```
Where:

- **ECC_FT_GFP:** Identifies a curve over a field with an odd prime number of elements.
- **ECC_FT_G2M:** Identifies a curve over a field of characteristic two (F_2^m).
- **ECC_FT_MON:** Identifies a curve that uses a Montgomery function.

```
typedef struct ECC_Point_st {
        unsigned char x[ECC_MAX_BUF_LEN];
        unsigned char y[ECC_MAX_BUF_LEN];
} ECC_Point_t;
```
Where:

- **x:** The X coordinate of the point. X is an element of the field over which the curve is defined.
- **y:** The Y coordinate of the point. Y is an element of the field over which the curve is defined.

```
typedef struct ECC_Curve_st {
        ECC_FieldType_t fieldType;
        unsigned char modulus[ECC_MAX_BUF_LEN];
        unsigned char a[ECC_MAX_BUF_LEN];
        unsigned char b[ECC_MAX_BUF_LEN];
        ECC_Point_t base;
        unsigned char bpOrder[ECC_MAX_BUF_LEN];
} ECC_Curve_t;
```
Where:

- **fieldType:** The field type, over which this curve is defined.
- **base:** The base point.
- **modulus:** The curve modulus. This value is the field polynomial for ECC_FT_G2M field types.
- **a:** The coefficient 'a' in the elliptic curve equation.
- **b:** The coefficient 'b' in the elliptic curve equation.
- **bpOrder:** The base point order. This buffer contains a big endian large number regardless of the field type.

```
typedef struct ECC_PrivateKey_st {
        unsigned char d[ECC_MAX_BUF_LEN];
} ECC_PrivateKey_t;
```

Where:

- **d:** The buffer containing the private key. The private key is always a big-endian large number, d, regardless of the field type of the curve.

```
typedef struct ECC_PublicKey_st {
        ECC_Point_t p;
} ECC_PublicKey_t;
```

Where:

- **p:** The point P on the curve, which is calculated from the curve base and the private key.

IDEA Cipher Object

Operation Supported

Encrypt, Decrypt, Multi-part MAC, Multi-Part MAC Verify

Key Encoding

Supports 16 byte key value.

Encrypt/Decrypt Modes

The least significant nibble (four bits) is used to determine the operational mode.

Possible values:

- **SYM_MODE_ECB (0)**

  Electronic Code Book (ECB) mode. It may be combined with a padding mode (see "Padding Mode " below).

- **SYM_MODE_CBC (1)**

  Cipher Block Chaining (CBC) mode. It may be combined with a padding mode (see "Padding Mode " below).

Padding Mode

The most significant nibble defines the padding mode used.

Possible value: **SYM_MODE_PADPKCS1 (0x80)**

PKCS#1 padding is applied. This causes 1 to 8 bytes of padding to be added to the data.

MAC modes

For MAC generation and verification, available modes include:

- **SYM_MODE_MAC_3 (0)**

  Standard MAC

- **SYM_MODE_MAC_GEN (1)**

  Standard MAC configurable length

Encrypt/Decrypt Parameter

In SYM_MODE_CBC, the parameter buffer must contain the IV (8 bytes). In SYM_MODE_ECB, there are no parameters.

MAC Parameters

When mode is SYM_MODE_MAC_GEN, parameter buffer contains at least 4 bytes, which is the little-endian encoding of an integer. The integer value must contain a value from 1 to 8, indicating the number of bytes to return.

RC2 Cipher Object

Operations Supported

Encrypt, Decrypt, Multi-Part MAC, Multi-Part MAC Verify

Key Encoding

128 byte (Max)

Encrypt/Decrypt Modes

The least significant nibble (four bits) is used to determine the operational mode. The following list defines the possible values:

- **SYM_MODE_ECB (0)**

  Electronic Code Book (ECB) mode. It may be combined with a padding mode (see below).

- **SYM_MODE_CBC (1)**

  Cipher Block Chaining (CBC) mode. It may be combined with a padding mode (see below).

MAC modes

For MAC generation and verification the following modes are available:

- **SYM_MODE_MAC_3 (0)**

  Standard CBC

- **SYM_MODE_MAC_GEN (1)**

  Standard CBC with configurable MAC length (max 8 bytes).

Encrypt/Decrypt Parameters
In SYM_MODE_CBC, the parameter buffer must contain the IV (8 bytes). In SYM_MODE_ECB, there are no parameters.

MAC Parameters
When mode is SYM_MODE_MAC_GEN, parameter buffer contains at least 4 bytes, which is the little-endian encoding of an integer. The integer value must contain a value from 1 to 8, indicating the number of bytes of the final IV that will be used as the MAC.

RC4 Cipher Object

Operations Supported
Encrypt/Decrypt

Key Encoding
256 byte (Max)

Encrypt/Decrypt Modes
None

Encrypt/Decrypt Parameters
None

RSA Cipher Object

Operations Supported
New, Free, GetInfo, EncInit, DecInit, SignInit, VerifyInit, EncryptUpdate, DecryptUpdate, SignRecover, VerifyRecover and Verify.

To perform an encrypt call

EncInit + EncryptUpdate

To perform a decrypt call

DecInit + DecryptUpdate

To generate a signature call

SignInit + SignRecover

To verify a signature call

> and view the recovered signature:

  VerifyInit + VerifyRecover

> without viewing the signature:

VerifyInit + Verify

Key encoding

The key format depends on whether the operation is expecting a public key or a private key.

Private Keys are used by:

> DecInit

> SignInit

Public Keys are used by:

> EncInit

> VerifyInit

Public Keys are stored in a CtPubRsaKey structure

Private Keys are stored in a CtPriRsaKey structure (see "RSA Key Structures" on page 78).

RSA Modes and Parameters

X509 Mode

```
#define RSA_MODE_X509        0
```

X509 Mode is the RAW uncooked mode. No padding or any other transformations are applied by the Cipher Object.

There is no parameter for this mode.

PKCS Mode

```
#define RSA_MODE_PKCS        1
```

PKCS Mode pads the input data into a specified block format according to the methods described in PKCS #1. The actual block padding method depends on whether encryption or signing operations are being performed.

For Encryption and Decryption, Block Type 2 is used.

For Signing, Block Type 1 is used.

There is no parameter for this mode.

9796 Mode

```
#define RSA_MODE_9796        2
```

ISO 9796 is a signature method only. Encrypt and Decrypt are not supported.

There is no parameter for this mode.

OAEP Mode

```
#define RSA_MODE_OAEP        3
```

OAEP is an Encryption/Decryption method only. Signing and Verification operations are not supported.

The padding is performed using the OAEP block format defined in PKCS #1.

This mode requires a parameter which is a structure of type **CK_RSA_PKCS_OAEP_PARAMS** (see **cryptoki.h**).

Restrictions apply to the values of members of the parameter structure:

- **hashAlg** must be CKM_SHA_1, CKM_SHA_244, CKM_SHA_256, CKM_SHA_384, CKM_SHA_512, CKM_SHA3_256, CKM_SHA3_224, CKM_SHA3_384, or CKM_SHA3_512
- **mgf** must be CKG_MGF1_SHA1, CKG_MGF1_SHA224, CKG_MGF1_SHA256, CKG_MGF1_SHA384, CKG_MGF1_SHA512, CKG_MGF1_SHA3_224, CKG_MGF1_SHA3_256, CKG_MGF1_SHA3_384, or CKG_MGF1_SHA3_512
- **source** must be CKZ_DATA_SPECIFIED

> **NOTE** If you are using firmware 5.06.03 or older, the hash algorithm must be the same in **hashAlg** and **mgf**.

Example

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

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

KEY WRAP OAEP Mode

```
#define RSA_MODE_KW_OAEP        4
```

Key Wrap OAEP is an Encryption/Decryption method only. Signing and Verification operations are not supported.

The padding is performed using the OAEP block format defined in PKCS #1 version 2.0

This mode requires a parameter which is a structure of type **CK_KEY_WRAP_SET_OAEP_PARAMS** (see **cryptoki.h**).

RSA Key Structures

```
#define MAX_RSA_MOD_BYTES (4096/8)
#define MAX_RSA_PRIME_BYTES ((MAX_RSA_MOD_BYTES / 2) + 4)
typedef unsigned char byte;
typedef struct {
    byte bits[2]; /* not used */
    byte mod [MAX_RSA_MOD_BYTES];
    byte exp [MAX_RSA_MOD_BYTES];
}
RSA_PUBLIC_KEY;
struct CtPubRsaKey {
int isPub;/* TRUE */
unsigned int modSz; /* in bytes */
RSA_PUBLIC_KEY key;
};
typedef struct CtPubRsaKey CtPubRsaKey;
typedef struct {
    byte bits[2]; /* not used */
    byte mod [MAX_RSA_MOD_BYTES];
    byte pub [MAX_RSA_MOD_BYTES];
    byte pri [MAX_RSA_MOD_BYTES];
    byte p   [MAX_RSA_PRIME_BYTES];
    byte q   [MAX_RSA_PRIME_BYTES];
    byte e1  [MAX_RSA_PRIME_BYTES];
    byte e2  [MAX_RSA_PRIME_BYTES];
    byte u   [MAX_RSA_PRIME_BYTES];
}
RSA_PRIVATE_KEY_XCRT;
struct CtPriRsaKey {
int isPub;/* FALSE */
int isXcrt;/* TRUE */
unsigned int modSz; /* significant size in bytes */
RSA_PRIVATE_KEY_XCRT key;
};
typedef struct CtPriRsaKey CtPriRsaKey;
```

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

CHAPTER 11: Hash Object

Cryptographic operations require that you obtain a pointer to an instance of a *cipher object* or a *hash object*. A cipher object may be used to encrypt, decrypt, sign (or MAC), or verify data. A hash object is used to perform a digest operation. There is a function for obtaining an instance of each of these objects.

This chapter provides the following details on Hash Objects:

> "FmCreateHashObject" below

> "Hash Object Functions" on the next page

See "Cipher Object" on page 48 for information on Cipher Objects.

FmCreateHashObject

Returns the address of a hash object for digest operations.

Synopsis

```
#include "fmciphobj.h"
HashObj * FmCreateHashObject(FMCO_HashObjIndex index);
```

Option	Description
`index`	The type of hash object requested. It can have the following values (defined in **fmciphobj.h**): > **FMCO_IDX_MD2**: Implementation of the MD-2 algorithm > **FMCO_IDX_MD5**: Implementation of the MD-5 algorithm > **FMCO_IDX_RMD128**: Implementation of the RIPEMD-128 algorithm > **FMCO_IDX_RMD160**: Implementation of the RIPEMD-160 algorithm > **FMCO_IDX_SHA1**: Implementation of the SHA-1 algorithm > **FMCO_IDX_SHA256**: Implementation of the SHA256 algorithm > **FMCO_IDX_SHA384**: Implementation of the SHA384 algorithm > **FMCO_IDX_SHA512**: Implementation of the SHA512 algorithm This list is correct at time of writing; the actual number of objects supported depends on the HSM firmware version.

Operations supported

> "Init" on page 81

> "Update" on page 81

> "Final" on page 82

Data Block Size

Any

Return Value

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

Comments

The returned hash object should be freed by calling its **Free()** function (See "Free" below).

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

Example

```
{
char buf[100];
char hash[100];
int lenOut;
HashObj *o = FmCreateHashObject(FMCO_IDX_SHA1);
If ( o == NULL )
Return error;
o->Init(o);
o->Update(o, data, len);
o->Final(o, hash, sizeof(hash), &lenOut);
o->Free(o);
}
```

Hash Object Functions

The generic Hash Object wraps hashing algorithms into a common interface. In this section, the following Hash Object functions are specified:

- "Free" below
- "Init" on the next page
- "Update" on the next page
- "Final" on page 82
- "GetInfo" on page 82
- "LoadParam" on page 83
- "UnloadParam" on page 84

Free

HashObj destructor

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

Synopsis

```
#include "fmciphobj.h"
int (*Free)(struct HashObj * ctx);
```

Parameter	Description
ctx	pointer to object to destroy

Return Value

See **CiphObjStat** in **cipherr.h**

Init

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

Synopsis

```
#include "fmciphobj.h"
int (*Init)(struct HashObj * ctx);
```

Parameter	Description
ctx	IN/OUT object to modify

Return Value

See **CiphObjStat** in **cipherr.h**

Update

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

The data passed in *buf* is passed through the hash algorithm

Synopsis

```
#include "fmciphobj.h"
int (*Update)(struct
HashObj * ctx,
const void * buf,
unsigned int length
);
```

Parameter	Description
ctx	IN/OUT object to modify
buf	IN message to hash

Parameter	Description
length	IN length of message

Return Value
See **CiphObjStat** in **cipherr.h**

Final

Final uses the object to finish a hash operation.

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

Synopsis

```
#include "fmciphobj.h"
int (*Final)(struct
HashObj * ctx,
unsigned char * hashVal,
unsigned int length,
unsigned int * plength
);
```

Parameter	Description
ctx	IN/OUT object to modify
hashVal	OUT where to place hash or NULL for length prediction
length	IN length of message
plength	OUT number of bytes (actually or potentially) returned in hashVal

Return Value
See **CiphObjStat** in **cipherr.h**

GetInfo

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

Synopsis

```
#include "fmciphobj.h"
int (*GetInfo)(struct HashObj * ctx, struct HashInfo * hinfo);
```

Parameter	Description
ctx	IN object to query

Parameter	Description
hinfo	OUT pointer to where to store the result (see the *HashInfo* description below)

HashInfo Structure

Allows application to determine characteristics of the digest algorithm.

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

Return Value

See **CiphObjStat** in **cipherr.h**

LoadParam

HashObjLoadParam directly modifies a Hash Object state.

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

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

See "UnloadParam" on the next page.

Synopsis

```
#include "fmciphobj.h"
int (*LoadParam)(struct
HashObj * ctx,
const unsigned char * parameters,
unsigned int paramlen
);
```

Parameter	Description
ctx	IN object to query
parameters	IN hash class specific information
paramlen	IN length (in bytes) of parameters

ctx: IN/OUT object to modify

Return Value

See **CiphObjStat** in **cipherr.h**

UnloadParam

HashObjUnloadParam queries a Hash Object state and returns certain information.

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

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

See "LoadParam" on the previous page.

Synopsis

```
#include "fmciphobj.h"
int (*UnloadParam)(struct
HashObj * ctx,
unsigned char * parameters,
unsigned int paramlen,
unsigned int * plen
);
```

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

Return Value

See **CiphObjStat** in **cipherr.h**

CHAPTER 12: Setting Privilege Level

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

Two possible settings are available:

PRIVILEGE_NORMAL=0

PRIVILEGE_OVERRIDE=1

SetPrivilegeLevel

This function is a SafeNet extension to PKCS#11. It can be used to set the privilege level of the caller to the specified value, if the caller has access to the function.

The function is available in the software **cryptoki** library to support FM emulation

The function cannot be called from outside the HSM (only 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 an FM, the privilege is automatically returned to normal when the current message is complete - when the **FM Dispatch** function or the currently intercepted Cryptoki function returns.

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

Synopsis

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

Parameter	Description
`level`	Desired privilege level

CHAPTER 13: SMFS Reference

SMFS is a Secure Memory File System (as exported to FMs).

It allows FMs to store keys into tamper-protected battery-backed Static RAM (SRAM)

It has the following general specifications:

- Arbitrary depth directory structure supported.
- File names are any character other than '\' or '/'.
- Path separator is '/' (the Windows '\' is not allowed)
- Files are of 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" below
- "Error Codes" below
- "File Attributes Structure (SmFsAttr)" on the next page
- "Function Descriptions" on the next page

Important Constants

- Max file name length is 15
- Max path length is 100
- Max number of open files is 32
- Max number of file search handles is 16

Error Codes

SMFS_ERR_ERROR	1	A general error has occurred
SMFS_ERR_NOT_INITED	2	The SMFS has not been initialized
SMFS_ERR_MEMORY	3	The SMFS has run out of memory
SMFS_ERR_NAME_TOO_LONG	4	The name given for a file is too long
SMFS_ERR_RESOURCES	5	The SMFS has run out of resources

SMFS_ERR_PARAMETER	6	An invalid parameter was passed to SMFS
SMFS_ERR_ACCESS	7	User does not have request access to file
SMFS_ERR_NOT_FOUND	8	Requested file was not found
SMFS_ERR_BUSY	9	Operation is being attempted on an open file
SMFS_ERR_EXIST	10	A file being created already exists
SMFS_ERR_FILE_TYPE	11	Operation being performed on wrong file type

File Attributes Structure (SmFsAttr)

This structure holds the file or directory attributes

Synopsis

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

Members

Size: Current file size in bytes or directory size in entries

isDir: Flag specifying if file is a directory

Function Descriptions

This section contains descriptions of the following functions:

> "SmFsCreateDir" on the next page
> "SmFsCloseFile" on the next page
> "SmFsCalcFree" on the next page
> "SmFsCreateFile" on page 89
> "SmFsDeleteFile" on page 89
> "SmFsFindFile" on page 89
> "SmFsFindFileClose" on page 90
> "SmFsFindFileInit" on page 90
> "SmFsGetFileAttr" on page 91
> "SmFsGetOpenFileAttr" on page 91
> "SmFsOpenFile" on page 91

> "SmFsReadFile" on page 92
> "SmFsRenameFile" on page 92
> "SmFsWriteFile " on page 93

SmFsCreateDir

Allocates SRAM memory and a directory entry for a directory.

Synopsis

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

Parameter	Description
`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 for success or an error condition.

SmFsCloseFile

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

Synopsis

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

Parameter	Description
`fh`	File handle of file to close.

Return Value

Returns 0 or an error condition.

SmFsCalcFree

Synopsis

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

Return Value

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

SmFsCreateFile

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

Synopsis

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

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

Return Value

Returns 0 for success or an error condition.

SmFsDeleteFile

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

Synopsis

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

Parameter	Description
name	Pointer to the absolute path of the file to delete

Return Value

Returns 0 or an error condition.

SmFsFindFile

Fetch name of next directory entry from file search context

Synopsis

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

Parameter	Description
sh	Search handle to continue

Parameter	Description
name	Pointer to location to hold found file name matching pattern
pattern	Length of name buffer

Return Value
Returns 0 or an error condition.

SmFsFindFileClose
Close a file search context.

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

Parameter	Description
sh	Search handle to close

Return Value
Returns 0 or an error condition.

SmFsFindFileInit
Creates a file iteration context.

Wild cards are:

- **?** - match any character
- ***** - match many characters

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

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

Return Value
Returns 0 or an error condition.

SmFsGetFileAttr
Get attributes of an open file. Returns an attributes structure for the unopen file '*name*'.

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

Parameter	Description
`name`	Pointer to absolute path
`a`	Pointer to the returned attributes structure

Return Value
Returns 0 or an error condition.

SmFsGetOpenFileAttr
Returns an attribute structure for the open file '*name*'.

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

Parameter	Description
`fh`	File handle
`a`	Pointer to the returned attributes structure

Return Value
Returns 0 or an error condition.

SmFsOpenFile
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.

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

Parameter	Description
fh	Pointer to the file handle
name	Pointer to the absolute path

Return Value
Returns 0 or an error condition.

SmFsReadFile
Reads data from file.

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

Parameter	Description
fh	Open file handle
offset	Zero-based starting point
buf	Pointer to the returned result
bc	The number of bytes to read from file

Return Value
Returns 0 or an error condition.

SmFsRenameFile
Renames a file.

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

Parameter	Description
oldName	Pointer to the absolute path of file to rename
newName	Pointer of new file name only (no path)

Return Value
Returns 0 or an error condition.

SmFsWriteFile
Write data to file.

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

Parameter	Description
`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.

CHAPTER 14: FMDEBUG Reference

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

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

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

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

Function Descriptions

This section contains the following function descriptions:

- "debug (macro)" below
- "printf/vprintf" on the next page
- "DBG_INIT" on the next page
- "DBG" on the next page
- "DBG_PRINT" on the next page
- "DBG_STR" on page 96
- "DUMP" on page 96
- "DBG_FINAL" on page 97

debug (macro)

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

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

Synopsis

```
debug( statements )
```

Example

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

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

printf/vprintf

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

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

1. Each **printf()**/**vprintf()** call prefaces its output with a log header that includes the FM's ID.
2. Each call to **printf()**/**vprintf()** has a new line appended to its output.

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

DBG_INIT

Not required. Retained for backwards compatibility with PSG.

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

Synopsis

```
int dbg_init()
```

DBG

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

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

On the ProtectServer PCIe HSMs, **printf** is preferred over use of this API.

Synopsis

```
int dbg(buf, len)
```

Parameter	Description
`buf`	Array of printable characters to output to the serial port
`len`	Length of buffer to output

DBG_PRINT

This function formats and dumps the given string to serial port 1 of the PSG.

Its use mirrors that of the C function **printf**.

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

On ProtectServer PCIe HSMs, **printf** is preferred over use of this API.

Synopsis

```
include <fmdebug.h>
int dbg_print(char *format, ...);
```

Parameter	Description
`format`	Format of the string to print. This argument is followed by the values to place inside the format string.

Return Value

Returns 0 or -1 for failure.

DBG_STR

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

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

On ProtectServer PCIe HSMs, **printf** is preferred over use of this API.

Synopsis

```
include <fmdebug.h>
int dbg_str()
```

Parameter	Description
`str`	String to output to serial port

DUMP

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

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

Synopsis

```
include <fmdebug.h>
void dump(char *desc, unsigned char *data, short len);
```

Parameter	Description
`desc`	Pointer to string that holds the description of the dumped buffer. This string is dumped immediately before the dumped buffer.
`data`	Pointer to buffer to be dumped
`len`	The length of the buffer to be dumped (in bytes)

DBG_FINAL

Not required. Retained for backwards compatibility with PSG.

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

Synopsis

```
include <fmdebug.h>
int dbg_final()
```

CHAPTER 15: 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 labelled by the MD_ prefix form the Message Dispatch (MD) API. The function prototypes are defined in the header file **md.h**. The libraries **etpso** (for local HSMs) and **etnetclient** (for remote HSMs) implement the PCIe bus and NetServer driver respectively. The driver is accessible via the **MD API**.

Functions included in this Reference are:

> "MD_Initialize" below
> "MD_Finalize" on the next page
> "MD_GetHsmCount" on the next page
> "MD_GetHsmState" on the next page
> "MD_ResetHsm" on page 101
> "MD_SendReceive" on page 102
> "MD_GetParameter" on page 105
> "FM Host Legacy Functions API" on page 106

Function Descriptions:

This section contains the following function descriptions:

MD_Initialize

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

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

Synopsis

```
#include <md.h>
MD_RV MD_Initialize(void)
```

Return Value

The function returns either MDR_OK or MDR_UNSUCCESSFUL.

MD_Finalize

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

Synopsis

```
#include <md.h>
void MD_Finalize(void)
```

Input Requirements

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

Return Value

None

MD_GetHsmCount

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

Synopsis

```
#include <md.h>
MD_RV MD_GetHsmCount(uint32* pHsmCount)
```

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

Input Requirements

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

Return Value

The HSM Count is returned in pHsmCount.

The function returns the following codes:

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

MD_GetHsmState

This function retrieves the current state of the specified HSM.

Synopsis

```
#include <md.h>
MD_RV MD_GetHsmState(uint32 hsmIndex,
                     HsmState_t* pState,
                     uint32* pErrorCode);
```

Parameter	Description
`hsmIndex`	Zero-based index of the HSM to query. For remote HSMs, HSMs are numbered according to the order that the HSMs IP addresses were entered in the ET_HSM_NETCLIENT_SERVERLIST registry key. For more information about this configuration item, see Specifying the Network Server(s) in the "Configuration Items" section of the *ProtectServer HSM and ProtectToolkit Installation Guide*. When **MD_Initialize()** is invoked, the message dispatch library assigns an index to each available HSM.
`pState`	Pointer to a variable to hold the HSM state. The pointer must not be NULL.
`pErrorCode`	Pointer to a variable which will hold the HSM error code when the function returns. The only error code that is returned is SCFS_SEE_LOG. The pointer may be NULL.

Input Requirements

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

Return Value

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

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

The function returns the following codes:

Function Code	Qualification
MDR_OK	N/A

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

MD_ResetHsm

This function is used to reset the specified HSM.

Synopsis

```
#include <md.h>
MD_RV MD_ResetHsm(uint32 hsmIndex);
```

Parameter	Description
`hsmIndex`	Zero-based index of the HSM to query. For remote HSMs, the HSM indices are numbered according to the order that the HSMs' IP addresses were entered in the ET_HSM_NETCLIENT_SERVERLIST registry key. For more information about this configuration item, see Specifying the Network Server(s) in the "Configuration Items" section of the *ProtectServer HSM and ProtectToolkit Installation Guide*. When **MD_Initialize ()** is invoked the message dispatch library assigns an index to each available HSM.

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. For more information about this configuration item, see Network Mode Server Configuration Items in the "Configuration Items" section of the *ProtectServer HSM and ProtectToolkit Installation Guide*. If this limitation has been set, then this function may only be called when the HSM stat is not S_NORMAL_OPERATION. See "MD_GetHsmState" on page 99 for further details.

Return Value

The HSM is reset.

The function returns the following codes:

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

MD_SendReceive

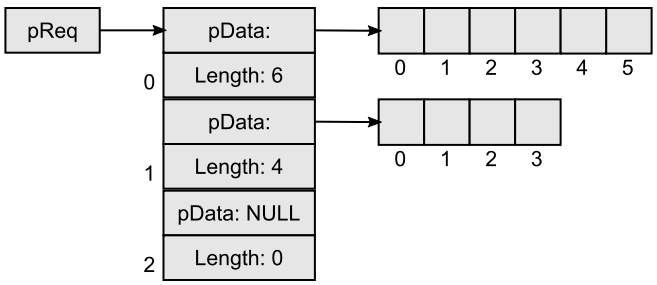
This function is used to send a request and receive the response.

Synopsis

```
#include <md.h>
MD_RV MD_SendReceive(uint32 hsmIndex,
                     uint32 originatorId,
                     uint16 fmNumber,
                     MD_Buffer_t* pReq,
                     uint32 timeout,
                     MD_Buffer_t* pResp,
                     uint32* pReceivedLen,
                     uint32* pFmStatus);
```

Parameter	Description
`hsmIndex`	Zero-based index of the HSM to query. For remote HSMs, HSMs are numbered according to the order that the HSMs' IP addresses were entered in the ET_HSM_NETCLIENT_SERVERLIST registry key. For more information about this configuration item, see Specifying the Network Server(s) in the "Configuration Items" section of the *ProtectServer HSM and ProtectToolkit Installation Guide*. When **MD_Initialize()** is invoked, the message dispatch library assigns an index to each available HSM.
`fmNumber`	Identifies whether the request is intended for an FM or not. This value must be set to the FMID of the FM that will handle the message (**#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.

Parameter	Description

pReq	Array of request buffers to send to the FM module. For user-defined functions, the structure and content of the array of buffers is user-defined. Refer to "javahsmreset" on page 37 and "javahsmstate" on page 37 for an example of how to construct the response and request buffers for a user-defined function in Java.
	Each buffer in the array is an **MD_Buffer_t** struct, which contains a pointer to the data and the number of bytes of data, as detailed below.
	`typedef struct`
	`{`
	`uint8*pData;`
	`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 5: 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 on how these buffers must be constructed. The memory for the **pResp** buffers must be allocated in the context of the application which calls the function. The **pData** field and length fields must be assigned appropriately to conform to the anticipated response packet. The buffers are filled in order until either the entire response is copied or the buffers overflow (this condition determined by **pReceivedLen**, described below). The value of this parameter can be NULL if the FM function will not return a response.
pReceivedLen	Address of variable to hold the total number of bytes placed in the response buffers. The memory for this variable must be allocated in the context of the application which calls the function. The value of this parameter can be NULL if the FM function will not return a response.

Parameter	Description
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.

Input Requirements

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

Return Value

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	N/A
MDR_ UNSUCCESSFUL	N/A
MDR_INVALID_ PARAMETER	If the pointer supplied for pReq is NULL, if the request requires a response and the pointer supplied for pResp is NULL or if pReserved is not zero.
MDR_NOT_ INITIALIZE	If the message dispatch library was not previously initialized successfully.
MDR_INVALID_ HSM_INDEX	If HSM index was not in the range of accessible HSMs.
MDR_ INSUFFICIENT_ RESOURCE	If there is insufficient memory on either the host or adapter
MDR_OPERATION_ CANCELLED	The operation was cancelled in the HSM. This code will not be returned.
MDR_INTERNAL_ ERROR	The HSM has detected an internal error. This code will be returned if there is a fault in the firmware or device driver.
MDR_ADAPTER_ RESET	The HSM was reset during the operation. This could be possibly due to the MD_ ResetHsm command being issued during the operation.
MDR_FM_NOT_ AVAILABLE	An invalid FM number was used.

MD_GetParameter

This function obtains the value of a system parameter.

Synopsis

```
#include <md.h>
MD_RV MD_GetParameter(MD_Parameter_t parameter,
                      void*pValue,
                      unsigned int valueLen);
```

Parameter	Description
`parameter`	The following parameter, defined in **md.h**, may be queried: MDP_MAX_BUFFER_LENGTH Value: 1 The recommended maximum buffer size (in bytes) for messages that can be sent using the MD library. While messages larger than this buffer size may be accepted by the library, exceeding it is not recommended. Different HSM access providers have different values for this parameter. When this parameter returns 0 via pValue, 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, which has the following buffer requirements: MDP_MAX_BUFFER_LENGTH, unsigned integer Size: 4 bytes 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.
`valueLen`	The length of the buffer, pValue, in bytes. If the buffer length is not correct, MDR_INVALID_PARAMETER is returned.

Input Requirements

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

Return Value

The function returns the following codes:

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

FM Host Legacy Functions API

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

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

Legacy Function	Superceded by
FM_Initialize	MD_Initialize
FM_Finalize	MD_Finalize
FM_DispatchRequest	MD_SendReceive

CHAPTER 16: HSM Functions Reference

A number of libraries are required to use the functionality provided by the SafeNet FM SDK. This chapter describes the function sets provided by these libraries.

As well as 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.

> "HIFACE Reply Management Functions" on the next page
> "Functionality module dispatch switcher function" on page 114
> "Serial Communication Functions" on page 114
> "High Resolution Timer Functions" on page 120
> "Cprov function patching helper function" on page 121
> "Current Application ID functions" on page 122
> "PKCS#11 State Management Functions" on page 123
> "FM Header Definition Macro" on page 130

Summary

Subset of ISO C99 standard library

The FM SDK supports a subset of the ISO C 99 standard library, as defined by ISO/IEC 9899:1999. In general, floating point math, multibyte character, localization and I/O APIs are not supported; **printf** and **vprintf** are exceptions and are 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
isalnum, isalpha, isblank, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, tolower, toupper

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

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

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

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

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

Extensions to the Standard C API

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

> **NOTE** These may not be available in emulation mode. See "Supported C APIs" on page 22.

ctype.h
isascii, toascii

string.h
strdup, strsep

Unsupported Standard C APIs

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

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

HIFACE Reply Management Functions

This section contains the following reply buffer management functions of the service module, provided by **libfmcsa8k.a**:

- "SVC_GetReplyBuffer" on the next page
- "SVC_ConvertReqToReply" on the next page
- "SVC_SendReply" on the next page
- "SVC_ResizeReplyBuffer" on page 110
- "SVC_DiscardReplyBuffer" on page 110
- "SVC_GetUserReplyBufLen" on page 111
- "SVC_GetPid" on page 111
- "SVC_GetOid" on page 111
- "SVC_GetRequest" on page 112
- "SVC_GetRequestLength" on page 112

> "SVC_GetReply" on page 112
> "SVC_GetReplyLength" on page 113

SVC_GetReplyBuffer

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

Synopsis

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

Parameter	Description
`token`	The token identifying the request.
`replyLength`	The length of the reply buffer requested by the caller.

Output Requirements

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

SVC_ConvertReqToReply

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

Synopsis

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

Parameter	Description
`token`	The token identifying the request.

Output Requirements

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

SVC_SendReply

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

Synopsis

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

Parameter	Description
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 values of this parameter does not affect the reply delivery in any way.

Output Requirements

None.

SVC_ResizeReplyBuffer

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

Synopsis

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

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

Output Requirements

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

SVC_DiscardReplyBuffer

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

Synopsis

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

Parameter	Description
token	The token identifying the request.

Output Requirements
None.

SVC_GetUserReplyBufLen

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

Synopsis
```
uint32 SVC_GetUserReplyBufLen( HI_MsgHandle token );
```

Parameter	Description
`token`	The token identifying the request.

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

SVC_GetPid

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

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

Parameter	Description
`token`	The token identifying the request.

Output Requirements
The Process identifier recorded in the request is returned.

SVC_GetOid

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

Synopsis
```
uint32 SVC_GetOid( HI_MsgHandle token );
```

Parameter	Description
`token`	The token identifying the request.

Output Requirements
The originator identifier recorded in the request is returned.

SVC_GetRequest
This function retrieves the address of request data.

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

Parameter	Description
token	The token identifying the request.

Output Requirements
The request buffer address is returned.

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

Synopsis
```
uint32 SVC_GetRequestLength( HI_MsgHandle token );
```

Input Parameters

Parameter	Description
token	The token identifying the request.

Output Requirements
The number of bytes in the request buffer is returned.

SVC_GetReply
This function retrieves the address of current reply buffer.

Synopsis
```
void *SVC_GetReply( HI_MsgHandle token );
```

Parameter	Description
token	The token identifying the request.

Output Requirements

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

SVC_GetReplyLength

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

Synopsis

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

Parameter	Description
`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.

Functionality module dispatch switcher function

This section contains the firmware message dispatch management functions.

FMSW_RegisterDispatch

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

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

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

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

The pair (reqBuffer, reqLength) defines the concatenated data that has been received on the request. See "MD_SendReceive" on page 102 function for the details of request dispatching.

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

Synopsis

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

Input Parameters

Parameter	Description
`fmNumber`	The FMID of the FM.
`dispatch`	Pointer on API handler function

Return Code

`FMSW_OK`: The function was registered successfully.

`FMSW_BAD_POINTER`: The function pointer is invalid

`FMSW_INSUFFICIENT_RESOURCES`: Not enough memory to complete operation

`FMSW_BAD_FM_NUMBER`: The FM number is incorrect.

`FMSW_ALREADY_REGISTERED`: A dispatch function was already registered.

Serial Communication Functions

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

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

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

> "SERIAL_GetNumPorts" below
> "SERIAL_Open" below
> "SERIAL_Close" on the next page
> "SERIAL_InitPort" on the next page
> "SERIAL_SendData" on the next page
> "SERIAL_WaitReply" on page 117
> "SERIAL_ReceiveData" on page 117
> "SERIAL_FlushRX" on page 118
> "SERIAL_GetControlLines" on page 118
> "SERIAL_SetControlLines" on page 119
> "SERIAL_SetMode" on page 119

SERIAL_GetNumPorts

This function returns the number of serial ports available.

Synopsis

```
int SERIAL_GetNumPorts(void);
```

Return Code

The number of serial ports available.

SERIAL_Open

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

Synopsis

```
int SERIAL_Open( int port );
```

Parameter	Description
`port`	Serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.

Return Code

0: Port opened successfully

others: An error prevented the serial port from opening

Comments

This function in no way guarantees safe sharing of the ports.

Any application can call **SERIAL_Close()** to get the access to the port, or can use SERIAL functions without opening the port first.

SERIAL_Close

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

Synopsis

```
void SERIAL_Close(int port);
```

Parameter	Description
`port`	Serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.

Return Code

N/A

Comments

See "SERIAL_Open" on the previous page

SERIAL_InitPort

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

Synopsis

```
int SERIAL_InitPort(int port);
```

Parameter	Description
`port`	Serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.

Return Code

`0`: The serial port was initialized successfully.

`-1`: There was an error initializing the port.

SERIAL_SendData

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

Synopsis

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

Parameter	Description
`port`	Serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.

Parameter	Description
buf	Pointer to an array of bytes to be sent
bufLen	Length of the buffer, in bytes
timeout	Milliseconds to wait for a character to be sent. A timeout of -1 will use the default timeout. **NOTE** The timeout value refers to the total time taken to send the data. For example, a 2 millisecond timeout for sending 10 characters in 9600 baud setting will always fail - the timeout must be at least 10 milliseconds.

Return Code

0: The characters were sent successfully.

-1: There was an error.

SERIAL_WaitReply

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

Synopsis

```
int SERIAL_WaitReply( int port );
```

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

Return Code

0: There is a character at the serial port.

-1: Timeout occurred, and no data appeared.

SERIAL_ReceiveData

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

Synopsis

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

Parameter	Description
port	Serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.

Parameter	Description
buf	Pointer to an array of bytes, which will hold the received data.
len	Pointer to an integer which will hold the actual number of characters received.
bufLen	Both the maximum amount of data, in bytes, of the buffer,and the number of bytes requested from the serial port.
timeout	Milliseconds to wait for a character to appear. A timeout of -1 will use the default timeout

Return Code

0: Requested number of bytes has been received.

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

SERIAL_FlushRX

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

Synopsis

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

Parameter	Description
port	Serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.

SERIAL_GetControlLines

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

Synopsis

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

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

Return Code

0: The function succeeded

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

Comments

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

SERIAL_SetControlLines

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

Synopsis

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

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

Return Code

`0`: The function succeeded

`-1`: The function failed

Comments

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

SERIAL_SetMode

Used to set the serial port communication parameters.

Synopsis

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

Parameter	Description
port	Serial port number (0 based). Specify port 0 to redirect the output to the hsmtrace log.
baud	Baud rate.
numBits	Number of bits in a character. Should be 7 or 8
parity	One of the following: > SERIAL_PARITY_NONE > SERIAL_PARITY_ODD > SERIAL_PARITY_EVEN > SERIAL_PARITY_ONE > SERIAL_PARITY_ZERO
numStop	Number of stop bits in a character. Should be 1 or 2
hs	Handshake type. Should be one of the following: > SERIAL_HS_NONE > SERIAL_HS_RTSCTS > SERIAL_HS_XON_XOFF

> **NOTE** Serial flow control is not implemented in the current HSM firmware. This value should be set to SERIAL_HS_NONE.

Return Code

0: Mode changed successfully

-1: There was an error

High Resolution Timer Functions

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

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

> "THR_BeginTiming" below

> "THR_UpdateTiming" on the next page

THR_BeginTiming

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

Synopsis

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

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

Return Code

N/A

THR_UpdateTiming

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

Synopsis

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

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

Return Code

N/A

Cprov function patching helper function

This section contains information about Cprov function patching operations.

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

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

OS_GetCprovFuncTable

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

Synopsis

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

Return Code

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

Current Application ID functions

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

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

> "FM_GetCurrentPid" below

> "FM_GetCurrentOid" below

> "FM_SetCurrentPid" on the next page

> "FM_SetCurrentOid" on the next page

FM_GetCurrentPid

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

Synopsis

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

Return Code

The PID of the application which originated the request.

FM_GetCurrentOid

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

Synopsis

```
unsigned long FM_GetCurrentOid(void);
```

Return Code
The OID of the application which originated the request.

FM_SetCurrentPid

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

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

Parameter	Description
pid	The new PID to be recorded in the request.

Return Code
N/A

FM_SetCurrentOid

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

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

Parameter	Description
oid	The new OID to be recorded in the request.

Return Code
N/A

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 cleanup of the associated buffer when the structure in question is destroyed.

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

> "FM_SetAppUserData" on the next page
> "FM_GetAppUserData" on the next page
> "FM_SetSlotUserData" on page 125
> "FM_GetSlotUserData" on page 126

> "FM_SetTokenUserData" on page 126
> "FM_GetTokenUserData" on page 127
> "FM_SetTokenAppUserData" on page 128
> "FM_GetTokenAppUserData" on page 128
> "FM_SetSessionUserData" on page 129
> "FM_GetSessionUserData" on page 130

FM_SetAppUserData

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

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

Synopsis

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

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

Return Code

CKR_OK: The operation was successful.

CKR_ARGUMENTS_BAD: freeUserData was NULL, when userData was not NULL, or fmNo was not FM_NUMBER_CUSTOM_FM.

CKR_CRYPTOKI_NOT_INITIALIZED: Cryptoki is not yet initialized.

FM_GetAppUserData

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

Synopsis

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

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

Return Code

CKR_OK: The operation was successful. The associated user data is placed in the variable specified by ppUserData.

CKR_ARGUMENTS_BAD: ppUserData was NULL, or fmNo was not FM_NUMBER_CUSTOM_FM.

CKR_CRYPTOKI_NOT_INITIALIZED: Cryptoki is not yet initialized.

FM_SetSlotUserData

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

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

Synopsis

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

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

Return Code

CKR_OK: The operation was successful.

CKR_ARGUMENTS_BAD: freeUserData was NULL, when userData was not NULL, or fmNo was not FM_NUMBER_CUSTOM_FM.

CKR_CRYPTOKI_NOT_INITIALIZED: Cryptoki is not yet initialized.

FM_GetSlotUserData

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

Synopsis

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

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

Return Code

CKR_OK: The operation was successful. The associated user data is placed in the variable specified by ppUserData.

CKR_ARGUMENTS_BAD: ppUserData was NULL or fmNo was not FM_NUMBER_CUSTOM_FM.

CKR_SLOT_ID_INVALID: The specified slot ID is invalid.

CKR_CRYPTOKI_NOT_INITIALIZED: Cryptoki is not yet initialized.

FM_SetTokenUserData

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

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

Synopsis

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

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

Return Code

CKR_OK: The operation was successful.

CKR_ARGUMENTS_BAD: freeUserData was NULL or fmNo was not FM_NUMBER_CUSTOM_FM.

CKR_SLOT_ID_INVALID: The specified slot ID is invalid.

CKR_TOKEN_NOT_PRESENT: The specified slot does not contain a token.

CKR_CRYPTOKI_NOT_INITIALIZED: Cryptoki is not yet initialized.

FM_GetTokenUserData

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

Synopsis

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

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

Return Code

CKR_OK: The operation was successful.

CKR_ARGUMENTS_BAD: ppUserData was NULL or fmNo was not FM_NUMBER_CUSTOM_FM.

CKR_SLOT_ID_INVALID: The specified slot ID is invalid.

CKR_CRYPTOKI_NOT_INITIALIZED: Cryptoki is not yet initialized.

FM_SetTokenAppUserData

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

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

Synopsis

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

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

Return Code

CKR_OK: The operation was successful.

CKR_ARGUMENTS_BAD: freeUserData was NULL or fmNo was not FM_NUMBER_CUSTOM_FM.

CKR_SLOT_ID_INVALID: The specified slot ID is invalid.

CKR_TOKEN_NOT_PRESENT: The specified slot does not contain a token.

CKR_CRYPTOKI_NOT_INITIALIZED: Cryptoki is not yet initialized.

FM_GetTokenAppUserData

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

Synopsis

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

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

Return Code

CKR_OK: The operation was successful.

CKR_ARGUMENTS_BAD: ppUserData was NULL or fmNo was not FM_NUMBER_CUSTOM_FM.

CKR_SLOT_ID_INVALID: The specified slot ID is invalid.

CKR_CRYPTOKI_NOT_INITIALIZED: Cryptoki is not yet initialized.

FM_SetSessionUserData

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

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

Synopsis

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

Parameter	Description
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.

Parameter	Description
userData	Address of the memory block that will be associated with the session handle. if it is NULL, the current associated buffer is freed.
freeUserData	Address of a function that will be called to free the userData, if the library decides that it should be freed. It must be non-NULL if userData is not NULL.

Return Code

CKR_OK: The operation was successful.

CKR_ARGUMENTS_BAD: freeUserData was NULL or fmNo was not FM_NUMBER_CUSTOM_FM.

CKR_SESSION_HANDLE_INVALID: The specified session handle is invalid.

CKR_CRYPTOKI_NOT_INITIALIZED: Cryptoki is not yet initialized.

FM_GetSessionUserData

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

Synopsis

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

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

Return Code

CKR_OK: The operation was successful. The associated user data is placed in the variable specified by ppUserData.

CKR_ARGUMENTS_BAD: ppUserData was NULL or fmNo was not FM_NUMBER_CUSTOM_FM.

CKR_SESSION_HANDLE_INVALID: hSession is not a valid session handle.

FM Header Definition Macro

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

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

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

Usage

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

MY_FM_NUMBER: Must be the manifest constant FMID of the FM, in this software version.

FM_VERSION: A 16 bit integer, of the form 0xmmMM, where mm is the minor number, and MM is the major number (It is displayed as VMM.mm in ctconf). Example: V1.0f . is encoded as 0x0f01.

SERIAL_NO: An integer representing the serial number of the FM.

MANUFACTURER_ID: A string of at most 32 characters, which contains the manufacturer name. This does not need to be NULL terminated.

PRODUCT_ID: A string consisting of a maximum of 16 characters, which contains the FM name. This does not need to be NULL terminated.

CHAPTER 17: USB API Reference

You can use the USB API to write applications that can interact with the HSM via the card USB port:

This functionality can include:

- wrapping of PKCS objects and storing them on a USB flash memory drive
- Performing backup of SMFS stored key (non-PKCS keys)

The USB API works with your custom FM to enable the desired functionality.

> **NOTE** The ProtectServer USB API has been tested and validated with a representative sample of common USB memory drives. Thales cannot guarantee compatibility with all available brands/models.

This chapter contains the following sections:

- "USB File System" below
- "Data Structures" below
- "Example Usage" on the next page
- "Function Descriptions" on the next page

USB File System

The file system that the API creates on the USB memory drive is structured as follows:

Block 0: Header | Block 1 | Block 2 | Block N

```
<DataLen><LabelLen><FSlabel><DataLen><Data><DataLen><Data><...><DataLen><Data>
```

The memory block size is determined automatically and has the most common value of 512 octets.

- `<DataLen>` and `<LabelLen>`: These fields have a uint32 data type and they are stored in the USB memory as Big Endian values.
- `<FSLabel>`: a 0-terminated string with a maximum length of 32 bytes.

Data Structures

The following two data structures are used in the FM USB API:

```
typedef struct _capacity {
        uint32_t max_lba_msb;    //used for 'Read Capacity 16'. Contains MSB of max block
number.
        uint32_t max_lba;        //max number of logical block on the device for 'Read Capacity
10' or LSB of max number of logical block for 'Read Capacity 16'.
        uint32_t block_size;
        uint32_t device_size;    //Contains device size in Gb. 32 bits are capable to hold the
capacity of 4 Gb.
} capacity_t;



typedef struct _dev_properties {
        uint16_t vendorId;
        uint16_t productId;
        uint8_t endpoint_in;
        uint8_t endpoint_out;
} dev_properties_t;
```

Example Usage

Currently, one application of the USB API is supported. The following scenario assumes that cryptographic object A exists on the HSM token.

To wrap object A from the HSM token to a USB memory drive

1. Create wrapping key B using BPE.
2. Wrap object A with wrapping key B.
3. Copy wrapped key to USB memory drive using the FM USB API calls.
4. [Optional] Destroy wrapping key B and object A on the HSM token.

To restore the source object A to the token

1. Read wrapped object A from the USB memory drive.
2. Create a new wrapping key B with BPE (or use existing key if it was never deleted).
3. Unwrap wrapped object A on the HSM token.

Function Descriptions

Functions included in this reference are:

> "USBFS_Close" on the next page
> "USBFS_Destroy" on the next page
> "USBFS_Finalize" on page 135
> "USBFS_GetInfo" on page 136
> "USBFS_Init" on page 137
> "USBFS_New" on page 137

> "USBFS_Open" on page 138
> "USBFS_ReadData" on page 139
> "USBFS_WriteData" on page 140

This chapter also contains a list of vendor-defined error codes that can be produced by the USB API:

> "USB API Vendor-Defined Error Codes" on page 140

USBFS_Close

Writes the header from the HSM memory back to the USB file system.

Synopsis

```
int USBFS_Close(
    void *handlev,
    dev_properties_t *dp,
    capacity_t *cap,
    uint8_t *header
);
```

Input Parameters

Parameter	Description
`cap`	Capacity
`dp`	Structure
`handlev`	Device handle
`header`	Pointer to file system block 0

Input Requirements

Refer to the FM sample **usbdemo** for the call sequence requirements.

Return Value

The function returns CKR_USB_OK if successful, or one of the error codes in "USB API Vendor-Defined Error Codes" on page 140.

USBFS_Destroy

Destroys the existing USB file system, including label and size.

Synopsis

```
int USBFS_Destroy(
    void *handle,
    dev_properties_t *dp,
```

```
    capacity_t *cap,
    uint8_t **header
);
```

Input Parameters

Parameter	Description
cap	Capacity
dp	Structure
handlev	Device handle
header	Pointer to file system block 0

Input Requirements

Refer to the FM sample **usbdemo** for the call sequence requirements.

Return Value

The function returns CKR_USB_OK if successful, or one of the error codes in "USB API Vendor-Defined Error Codes" on page 140.

USBFS_Finalize

Finalizes the USB API sequence.

This is the last function required in any sequence of USB-related calls. It frees context, device handler, attaches the kernel driver if required, and frees any memory that was allocated for the FS header.

Synopsis

```
int USBFS_Finalize(
    void *ctx,
    void *handlev,
    int *kernelDriverAttachedFlag,
    uint8_t **header
);
```

Input Parameters

Parameter	Description
ctx	Current context
handlev	Device handle
header	Header string

Parameter	Description
kernelDriverAttachedFlag	If the kernel driver was attached before running the operation, this value will be TRUE.

Input Requirements
Refer to the FM sample **usbdemo** for the call sequence requirements.

Return Value
The function returns CKR_USB_OK if successful, or one of the error codes in "USB API Vendor-Defined Error Codes" on page 140.

USBFS_GetInfo
Returns the USB file system label and the length of the stored data in bytes.

Synopsis
```
int USBFS_GetInfo(
    void *handle,
    dev_properties_t *dp,
    capacity_t *cap,
    uint32_t *dataLen,
    uint8_t *label,
    uint8_t *header
);
```

Input Parameters

Parameter	Description
cap	Capacity
dataLen	Length of the data string to follow
dp	Structure
handle	Device handle
header	Pointer to file system block 0
label	File system label

Input Requirements
Refer to the FM sample **usbdemo** for the call sequence requirements.

Return Value

The function returns CKR_USB_OK if successful, or one of the error codes in "USB API Vendor-Defined Error Codes" on page 140.

USBFS_Init

Initializes the USB sub-system and returns the USB handle.

This is the first function required in any sequence of USB-related calls. It returns context, USB handler, properties and capacity structures, and the "Kernel Driver attached" flag (which indicates whether the kernel driver should be attached at the end of the execution).

Synopsis

```
int USBFS_Init(
    void *ctxv,
    void **handlev,
    dev_properties_t *dp,
    capacity_t *cap,
    int *kernelDriverAttachedFlag
);
```

Input Parameters

Parameter	Description
`cap`	Capacity
`ctxv`	Current context
`dp`	Structure
`handlev`	Device handle
`kernelDriverAttachedFlag`	If the kernel driver was attached before running the operation, this value will be TRUE.

Input Requirements

Refer to the FM sample **usbdemo** for the call sequence requirements.

Return Value

The function returns CKR_USB_OK if successful, or one of the error codes in "USB API Vendor-Defined Error Codes" on page 140.

USBFS_New

Creates a new, empty file system.

This function allocates space for the file system header, which contains only label length and label fields.

Synopsis

```
int USBFS_New(
    void *handle,
    char * label,
    dev_properties_t *dp,
    capacity_t *cap,
    uint8_t **header
);
```

Input Parameters

Parameter	Description
cap	Capacity
dp	Structure
handle	Device handle
header	Pointer to file system block 0
label	File system label

Input Requirements

The USBFS library has been initialized with **USBFS_Init()**, and there is no current existing file system.

Return Value

The function returns CKR_USB_OK if successful, or one of the error codes in "USB API Vendor-Defined Error Codes" on page 140. If a file system already exists, CKR_USB_INITIALIZED is returned. Use **USB_Destroy()** first to erase the current file system.

USBFS_Open

Reads the header into the HSM memory.

Synopsis

```
int USBFS_Open(
    void *handle,
    dev_properties_t *dp,
    capacity_t *cap,
    uint8_t **header
);
```

Input Parameters

Parameter	Description
cap	Capacity

Parameter	Description
dp	Structure
handle	Device handle
header	Pointer to file system block 0

Input Requirements
Refer to the FM sample **usbdemo** for the call sequence requirements.

Return Value
The function returns CKR_USB_OK if successful, or one of the error codes in "USB API Vendor-Defined Error Codes" on the next page.

USBFS_ReadData
Reads specified number of bytes from the USB file system to the HSM memory.

Synopsis
```
int USBFS_ReadData(
    void *handle,
    dev_properties_t *dp,
    capacity_t *cap,
    uint8_t *data,
    uint32_t *dataLen,
    uint8_t *header
);
```

Input Parameters

Parameter	Description
cap	Capacity
data	Data string
dataLen	Length of the data string to follow
dp	Structure
handle	Device handle
header	Pointer to file system block 0

Input Requirements
Refer to the FM sample **usbdemo** for the call sequence requirements.

Return Value

The function returns CKR_USB_OK if successful, or one of the error codes in "USB API Vendor-Defined Error Codes" below.

USBFS_WriteData

Writes the specified number of bytes (dataLen) to the USB file system, starting from block 1. This will overwrite the existing contents.

Synopsis

```
int USBFS_WriteData(
    void *handle,
    dev_properties_t *dp,
    capacity_t *cap,
    uint8_t *data,
    uint32_t dataLen,
    uint8_t *header
);
```

Input Parameters

Parameter	Description
`cap`	Capacity
`data`	Data string
`dataLen`	Length of the data string to follow
`dp`	Structure
`handle`	Device handle
`header`	Pointer to file system block 0

Input Requirements

Refer to the FM sample **usbdemo** for the call sequence requirements.

Return Value

The function returns CKR_USB_OK if successful, or one of the error codes in "USB API Vendor-Defined Error Codes" below.

USB API Vendor-Defined Error Codes

The table below lists the error codes that may be returned from USB API calls.

Name	Value	Description
CKR_USB_OK	0	No Error
CKR_USB_GENERAL_ERROR	0x80000100	Error, cause unknown.
CKR_USB_INITIALIZED	0x80000101	The USB device has already been initialized.
CKR_USB_NO_DEVICES	0x80000102	No USB device is present.
CKR_USB_TRANSFER	0x80000103	Error during USB read/write operations.
CKR_USB_FS_EXISTS	0x80000104	A USB file system already exists.
CKR_USB_MEMORY	0x80000105	Memory allocation error.
CKR_USB_FS_NOT_PRESENT	0x80000106	No USB file system has been created.
CKR_USB_FS_NOT_OPENED	0x80000107	The file system must be opened before continuing.

Glossary

A

Adapter

The printed circuit board responsible for cryptographic processing in a HSM

AES

Advanced Encryption Standard

API

Application Programming Interface

ASO

Administration Security Officer

Asymmetric Cipher

An encryption algorithm that uses different keys for encryption and decryption. These ciphers are usually also known as public-key ciphers as one of the keys is generally public and the other is private. RSA and ElGamal are two asymmetric algorithms

B

Block Cipher

A cipher that processes input in a fixed block size greater than 8 bits. A common block size is 64 bits

Bus

One of the sets of conductors (wires, PCB tracks or connections) in an IC

C

CA

Certification Authority

CAST

Encryption algorithm developed by Carlisle Adams and Stafford Tavares

Certificate

A binding of an identity (individual, group, etc.) to a public key which is generally signed by another identity. A certificate chain is a list of certificates that indicates a chain of trust, i.e. the second certificate has signed the first, the

third has signed the second and so on

CMOS

Complementary Metal-Oxide Semiconductor. A common data storage component

Cprov

ProtectToolkit C - SafeNet's PKCS #11 Cryptoki Provider

Cryptoki

Cryptographic Token Interface Standard. (aka PKCS#11)

CSA

Cryptographic Services Adapter

CSPs

Microsoft Cryptographic Service Providers

D

Decryption

The process of recovering the plaintext from the ciphertext

DES

Cryptographic algorithm named as the Data Encryption Standard

Digital Signature

A mechanism that allows a recipient or third party to verify the originator of a document and to ensure that the document has not be altered in transit

DLL

Dynamically Linked Library. A library which is linked to application programs when they are loaded or run rather than as the final phase of compilation

DSA

Digital Signature Algorithm

E

Encryption

The process of converting the plaintext data into the ciphertext so that the content of the data is no longer obvious. Some algorithms perform this function in such a way that there is no known mechanism, other than decryption with the appropriate key, to recover the plaintext. With other algorithms there are known flaws which reduce the difficulty in recovering the plaintext

F

FIPS

Federal Information Protection Standards

FM

Functionality Module. A segment of custom program code operating inside the CSA800 HSM to provide additional or changed functionality of the hardware

FMSW

Functionality Module Dispatch Switcher

H

HA

High Availability

HIFACE

Host Interface. It is used to communicate with the host system

HSM

Hardware Security Module

I

IDEA

International Data Encryption Algorithm

IIS

Microsoft Internet Information Services

IP

Internet Protocol

J

JCA

Java Cryptography Architecture

JCE

Java Cryptography Extension

K

Keyset

A keyset is the definition given to an allocated memory space on the HSM. It contains the key information for a specific user

KWRAP

Key Wrapping Key

M

MAC

Message authentication code. A mechanism that allows a recipient of a message to determine if a message has been tampered with. Broadly there are two types of MAC algorithms, one is based on symmetric encryption algorithms and the second is based on Message Digest algorithms. This second class of MAC algorithms are known as HMAC algorithms. A DES based MAC is defined in FIPS PUB 113, see http://www.itl.nist.gov/div897/pubs/fip113.htm. For information on HMAC algorithms see RFC-2104 at http://www.ietf.org/rfc/rfc2104.txt

Message Digest

A condensed representation of a data stream. A message digest will convert an arbitrary data stream into a fixed size output. This output will always be the same for the same input stream however the input cannot be reconstructed from the digest

MSCAPI

Microsoft Cryptographic API

MSDN

Microsoft Developer Network

P

Padding

A mechanism for extending the input data so that it is of the required size for a block cipher. The PKCS documents contain details on the most common padding mechanisms of PKCS#1 and PKCS#5

PCI

Peripheral Component Interconnect

PEM

Privacy Enhanced Mail

PIN

Personal Identification Number

PKCS

Public Key Cryptographic Standard. A set of standards developed by RSA Laboratories for Public Key Cryptographic processing

PKCS #11

Cryptographic Token Interface Standard developed by RSA Laboratories

PKI

Public Key Infrastructure

ProtectServer

SafeNet HSM

ProtectToolkit C

SafeNet's implementation of PKCS#11. Protecttoolkit C represents a suite of products including various PKCS#11 runtimes including software only, hardware adapter, and host security module based variants. A Remote client and server are also available

ProtectToolkit J

SafeNet's implementation of JCE. Runs on top of ProtectToolkit C

R

RC2/RC4

Ciphers designed by RSA Data Security, Inc.

RFC

Request for Comments, proposed specifications for various protocols and algorithms archived by the Internet Engineering Task Force (IETF), see http://www.ietf.org

RNG

Random Number Generator

RSA

Cryptographic algorithm by Ron Rivest, Adi Shamir and Leonard Adelman

RTC

Real-Time Clock

S

SDK

Software Development Kits Other documentation may refer to the SafeNet Cprov and Protect Toolkit J SDKs. These SDKs have been renamed ProtectToolkit C and ProtectToolkit J respectively. The names Cprov and ProtectToolkit C refer to the same device in the context of this or previous manuals. The names Protect Toolkit J and ProtectToolkit J refer to the same device in the context of this or previous manuals.

Slot

PKCS#11 slot which is capable of holding a token

SlotPKCS#11

Slot which is capable of holding a token

SO

Security Officer

Symmetric Cipher

An encryption algorithm that uses the same key for encryption and decryption. DES, RC4 and IDEA are all symmetric algorithms

T

TC

Trusted Channel

TCP/IP

Transmission Control Protocol / Internet Protocol

Token

PKCS#11 token that provides cryptographic services and access controlled secure key storage

TokenPKCS#11

Token that provides cryptographic services and access controlled secure key storage

U

URI

Universal Resource Identifier

V

VA

Validation Authority

X

X.509

Digital Certificate Standard

X.509 Certificate

Section 3.3.3 of X.509v3 defines a certificate as: "user certificate; public key certificate; certificate: The public keys of a user, together with some other information, rendered unforgeable by encipherment with the private key of the certification authority which issued it"