IBM 4767 PCIe Cryptographic Coprocessor
Custom Software Interface Reference

Note: Before using this information and the products it supports, be sure to read the general information under “Notices” on page 145.

Third Edition (June, 2018)
This and other publications related to the IBM 4767 PCIe Cryptographic Coprocessor can be obtained in PDF format from the product website. Click on the HSM 4767 link at *www.ibm.com/security/cryptocards*, and then click on the Library link.

Reader’s comments can be communicated to IBM by contacting the Crypto team at crypto@us.ibm.com.

Contents

Figures

About this document

The IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference describes the 4767 application programming interface (API) function calls that applications running on the 4767 use to obtain cryptographic and communication services from the operating system. It also describes the function calls that applications running on the host use to interact with applications running on the cryptographic coprocessor.
This manual is intended for developers who are creating applications to use with the coprocessor. This manual should be used in conjunction with the manuals listed under “Related publications”.

Prerequisite knowledge

The reader of this document should understand how to perform basic tasks (including editing, system configuration, file system navigation, and creating application programs) on the host machine. Familiarity with the Linux operating system that runs within the coprocessor hardware and application development process (as described in the *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Developer's Toolkit Guide*) may also be helpful.

Organization of this document

“Overview” discusses the separation of the 4767 API into host- and coprocessor-side components and describes how an application on the host interacts with an application on the cryptographic coprocessor. It includes the source for a sample host application and a sample coprocessor application that illustrate this interaction.

“Host-side API” describes the host-side portion of the 4767 API in detail.

“Coprocessor-side API” describes the coprocessor-side portion of the 4767 API in detail.

“Error code formatting” details how return codes are formatted.

“DES weak, semi-weak, and possibly weak keys” lists keys that are not suitable for use as DES keys. The random number generator can be instructed not to return any of these numbers.

“IBM root public key” lists (in hex) the IBM root public key.

“Notices” includes product and publication notices.

A list of abbreviations, a glossary, and an index complete the manual.

Typographic conventions

This publication uses the following typographic conventions:

Commands that you enter verbatim onto the command line are presented in `monospace` type.

Variable information and parameters, such as file names, type names, and function names (depending on context), are presented in *italic type*.

Constants are presented in **bold** type.

The names of items that are displayed in graphical user interface (GUI) applications--such as pull-down menus, check boxes, radio buttons, and fields--are presented in **bold** type.

Items displayed within pull-down menus are presented in ***bold italic*** type.

System responses in a shell-based environment are presented in `monospace` type.

Web addresses and directory paths are presented in *italic* type.

For readability in information that applies to both Linux® and Windows®, this publication uses Linux

naming conventions instead of showing both Linux and Windows conventions. For example, *cctk/<version>/samples* would be *cctk\<version>\samples* on Windows.

Syntax diagrams

The syntax diagrams in this section follow the typographic conventions listed in “Typographic conventions.” Optional items appear in brackets. Lists from which a selection must be made appear in braces with vertical bars separating the choices. See the following example.

```
COMMAND firstarg [secondarg] {a | b}
```

A value for *firstarg* must be specified. *secondarg* may be omitted. Either *a* or *b* must be specified.

Note: <CLU> is used generically throughout this document to indicate either *csulclu* on Linux® or *csufclu* on IBM AIX®, depending on the operating system for the machine in which the adapter is installed.

Related publications

Publications about IBM’s family of cryptographic coprocessors are available at: *http://www.ibm.com/security/cryptocards*.

Publications specific to the IBM 4767 PCIe Cryptographic Coprocessor and to CCA are available at: *http://www.ibm.com/security/cryptocards/pciecc2/library.shtml*.

The *CCA Basic Services Reference and Guide* has a section titled “Related Publications” that describes cryptographic standards, research, and practices relevant to the coprocessor. This document is available at: *http://www.ibm.com/security/cryptocards/pciecc2/library.shtml*.

Deterministic Random Bit Generation (DRBG) standards are described in NIST Special Publication 800-90A, “Recommendation for Random Number Generation using Deterministic Random Bit Generators.” This publication is available at : *http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf*.

Summary of changes

This edition of the *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference* contains product information that is current with the IBM 4767 PCIe Cryptographic Coprocessor announcements.

1 Overview

The IBM 4767 API allows applications running on the host to interact with applications running on the cryptographic coprocessor. The 4767 API includes a set of functions an application running on the host may invoke (the host-side API) and a set of functions an application running on the cryptographic coprocessor may invoke (the coprocessor-side API).

This chapter describes how an application on the host interacts with an application on the cryptographic coprocessor and illustrates the flow of data and messages among the various agents involved in the process. It also includes the source for a sample host application and a sample coprocessor application that illustrates the interaction.

1.1 System architecture

Figure 1 illustrates the principal agents and functional blocks in the system, with connections between components that communicate directly with one another.

Requests for service from the host application are sent via the host cryptographic coprocessor device driver. The host device driver

1. prepares buffers containing the request and any associated data,
2. allocates space to hold the expected reply, and
3. schedules a DMA operation to transfer the request and data to the coprocessor. *

.A device driver on the coprocessor (the Communication driver) receives requests and associated data from the host. The Communication driver transfers the requests and data to the target application on the coprocessor and schedules a DMA operation to transfer any reply to the host.*

* The host device driver and the Communication driver communicate directly across the PCIe bus in some cases.

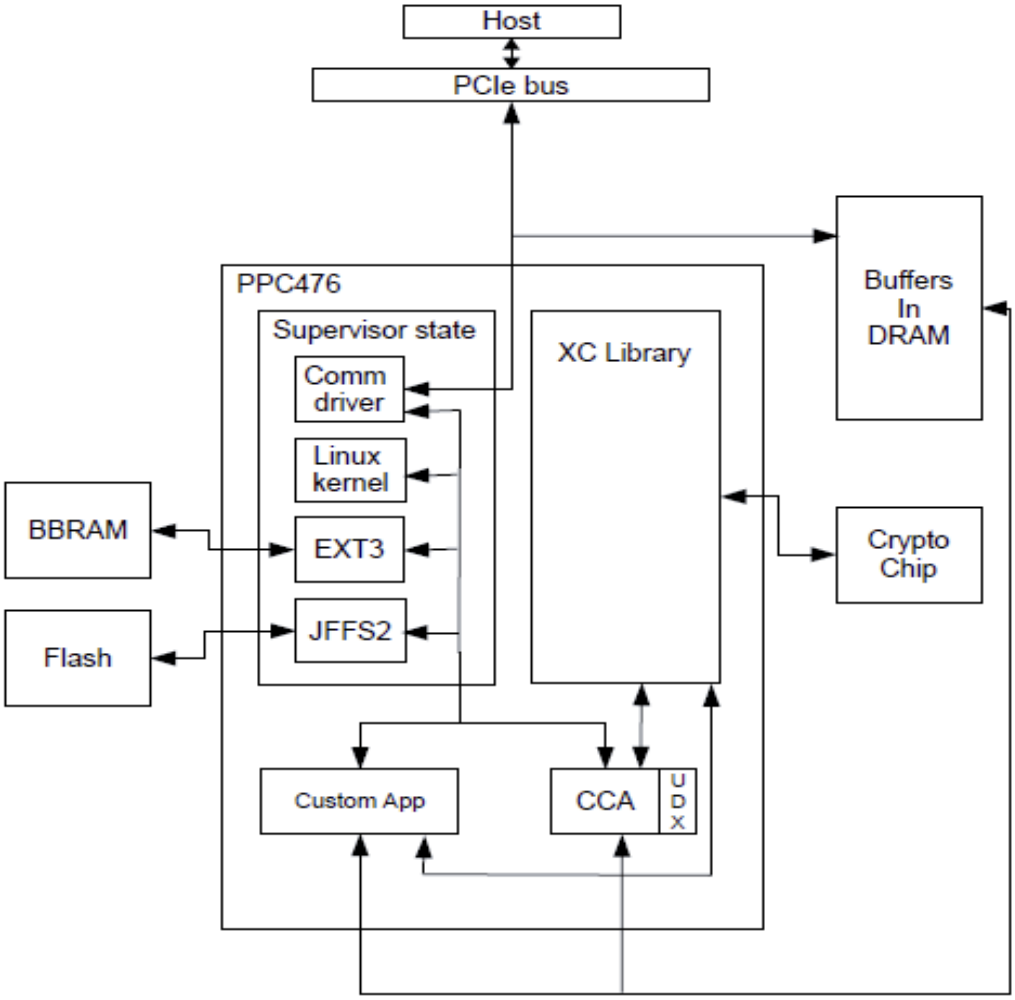

Figure 1 System architecture

1.2 Host and coprocessor interaction

Most API calls on the coprocessor are nonblocking. The IBM 4767 Toolkit samples provide a library that makes them appear to be blocking. The library is designed to make the translation of an existing, pre-4767 card application to the 4767 relatively straightforward. The description that follows assumes the use of that library.

The host application and coprocessor application exchange information as follows:

1. The coprocessor application calls *xcAttachWithCDUOption* to register with the Communication driver and passes the driver a structure of type *agentID_t* that uniquely identifies the coprocessor application.
2. The host application calls *xcAdapterCount* to determine how many cryptographic coprocessors are in the system.

3. The host application calls *xcOpenAdapter* to establish a communication channel between the host application and the coprocessor. *xcOpenAdapter* returns a handle which uniquely identifies a communication channel with the coprocessor.
4. The coprocessor application creates one or more “fibers” (user threads), at least one of which calls *xcGetRequest* to await the receipt of a request from the host.
5. The host application calls *xcRequest* to send a request to the coprocessor. The request includes both the handle returned by *xcOpenAdapter* and the application's *agentID_t*. The host application must pass one data buffer to the coprocessor and may pass two data buffers.
6. The *xcGetRequest* call returns to the coprocessor application.
7. The coprocessor application processes the request, possibly requesting services (for example, DES or RSA operations) from the libxccmn.so library.
8. The coprocessor application calls *xcPutReply* to return the result of the request to the host application. The coprocessor application must supply a return code and one data buffer and may supply two data buffers.

   Steps 5 through 9 are repeated each time the host generates a request. Several host applications may interact with the coprocessor application at the same time (i.e., several host applications can simultaneously have outstanding calls to *xcRequest*).
9. The host application calls *xcCloseAdapter* to close the communication channel between the host application and the coprocessor.

The host application initiates most transactions. The coprocessor application can initiate one by calling *xcPutEvent*.

1.3 Virtual packets

When the host application sends a request to the coprocessor application, the host application must supply one buffer of data to be transmitted to the coprocessor application and may supply two such buffers. For historical reasons, the first (mandatory) buffer is called the "request control block" and the second is called the "request data".

The Communication driver on the coprocessor assembles the information supplied by the host application into a "virtual packet", which is then presented to the coprocessor application. The virtual packet contains:

1. a virtual packet header (*xcVirtualPacket_t*),
2. the request control block supplied by the host,
3. the request data (if any) supplied by the host, and
4. padding.

The last field in the virtual packet header (*dataStart*) is also the first byte of the request control block. The request data (if any) immediately follows the end of the request control block (with no padding in between).

Together, the request control block and request data constitute a "host request block."

The total length of the host request block must be verified by the MAX_TRANSFER_CHECK macro provided with the cctk.

When the coprocessor application replies to a host request, the coprocessor application must supply one buffer of data to be transmitted to the host application (the "reply control block") and may supply two such buffers (the reply control block and the "reply data"). In contrast to the host behavior, these buffers are not

assembled into a single image. Instead, they are written directly to the buffers the host supplies.

The coprocessor application must supply exactly as many reply buffers as the host application expects.

1.4 Byte order

The host and coprocessor drivers involved in host-coprocessor communication do not attempt to enforce a consistent byte order. The coprocessor application should choose the byte order it expects to receive (typically big-endian, since the coprocessor is a big-endian machine) and the host application should ensure any multibyte numeric data is transmitted in that byte order.

The host device driver will automatically, when copying the virtual packet, byte swap the following fields: userDef, agentID, and (on reply) Status.

1.5 Request priority

The coprocessor maintains two separate queues for requests. A host application can specify the queue to which a particular request is directed, and the coprocessor application can specify a range of queues it wishes to examine to determine whether or not a request is pending. The coprocessor application can also control which queue is examined first (which in turn determines the order in which the queues are examined).

A coprocessor application calls *xcGetRequest* to retrieve a request from the host. The application passes a *getReq_t* structure whose *startMRB* field specifies which queue to examine first and whose *endMRB* field specifies which queue to examine last. If *startMRB* is less than *endMRB* the queues are searched in ascending numerical order. If *startMRB* is greater than *endMRB* the queues are searched in descending numerical order. If the two fields are equal only a single queue is searched.

1.6 Software attacks and defensive coding

Coprocessor applications run in a secure environment and often manipulate or manage sensitive data. To reduce the likelihood that this data will be compromised, a coprocessor application must assume any host application to which it provides service may have been written by an adversary in an attempt to mount an attack on the coprocessor application. For example, the coprocessor application should thoroughly validate any arguments provided by the host application.

The coprocessor attempts to limit the amount of damage an errant coprocessor application can cause. If an application terminates (via *exit( )* or by returning from *main( )*) or if one of the tasks in the application generates an exception (for example, divide by zero or addressing exception) and the application did not supply a fault handler for the task, the coprocessor may halt the system. If there is an unhandled exception, it will force the card to reboot itself automatically, and log messages will be sent to the syslog daemon and transmitted to the host. The host configuration determines what happens to syslog messages.

1.7 Sample applications

The following applications (found in the Toolkit) illustrate the concepts described in this chapter. The applications include the following header and source code files:

- *OEM_hdr.h* defines the protocol used between the host and coprocessor applications.
- *OEM_card.c* is the coprocessor application source code.
- *OEM_host.c* is the host application source code.

This simple example illustrates the communication mechanism between the host and the coprocessor; it does not utilize the cryptographic capabilities of the coprocessor.

Various structures can be passed between host and coprocessor applications. It is important to compile both applications with the same structure-packing conventions. This can be controlled with a compiler command line flag, or by a pragma in common header files.

1.7.1 How to compile and link the sample programs

OEM_host.c should be compiled and linked just like any other application on the host (link with *libcsulcca.so* on Linux and *csuncca.lib* on Windows).

Refer to the *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Developer's Toolkit Guide* for information on how to compile and link *OEM_card.c* (the coprocessor application) and how to load the executable into the coprocessor.

The host application and the coprocessor application must agree on the packing conventions for structures used in the interface between them (defined in *OEM_hdr.h* and the various IBM 4767 Toolkit header files). You may need to add pragmas to these files to ensure that is the case.

2 Host-side API

The host-side portion of the 4767 API (host-side API) allows an application running on the host to exchange information with an application running on a cryptographic coprocessor.

Host-side API calls can be used to determine the number of cryptographic coprocessors installed on the host, establish a communication channel to a specific coprocessor, exchange information via the channel with a specific application running on the coprocessor, and close the channel.

This chapter describes each of the functions supplied by the host-side API. Each description includes the function prototype (in C), the inputs to the function, the outputs returned by the function, and the most common return codes generated by the function.

2.1 General information

This section contains information about the host-side API functions.

2.1.1 Host-side API functions

The host-side API includes the following functions:

xcAdapterCount	Determine the number of cryptographic coprocessors installed on the host.
xcOpenAdapter	Establish a communication channel to a specific coprocessor.
xcRequest	Send a request across an open communication channel to a specific application and receive the reply.
xcCloseAdapter	Close a communication channel that was previously opened via a call to xcOpenAdapter.
xcGetAdapterData	Retrieve identification data from a coprocessor.
xcGetHardwareInfo	Retrieve a coprocessor's hardware version.
xcGetTamperBits	Retrieve a coprocessor's tamper status.
xcResetAdapter	Reset a coprocessor.

All host-side API calls are synchronous (that is, the calls do not return until the corresponding function is complete).

2.1.2 Header files

The prototypes for these functions are contained in *xc_host.h*. Other header files used to create host applications are *y_hostRB.h*, *xc_types.h* and *xc_err.h*. The code that implements the host-side API functions is in *libcsulcca.so* on Linux and is in *csuncca.dll* on Windows. The library is included in the Toolkit; refer to the *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Developer's Toolkit Guide* for details. The prototypes in *xc_host.h* include keywords, preprocessor directives, or both that ensure the functions are called using the appropriate linkage convention regardless of the default linkage convention in effect during compilation. For clarity, the prototypes that appear in this chapter do not include this syntax.

2.1.3 Sample code

Examples of the use of many of the host-side API functions can be found in the following files shipped with

the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Developer's Toolkit:

- *samples/toolkit/rte/host/hre.c*
- *samples/toolkit/skeleton/host/skelhost.c*

2.1.4 Error codes

“Error code formatting” on page 141 describes the format of a return code. Note that although the host-side API calls return a 32-bit return code, in some cases the low-order bits of the value contain additional information rather than a constant value:

- If the cryptographic coprocessor's power-on self test (POST) fails, a host-side API call may return **POST_ERR** in the high-order 16 bits of the return code and a value that identifies the specific POST checkpoint that failed in the low-order 16 bits. POST checkpoint identifiers are subject to change and are not made publicly available.
- If the cryptographic coprocessor microcode detects an attempt to tamper with the physical security of the adapter, a host-side API call may return **HDDSecurityTamper** in the high-order 24 bits of the return code and the state of the hardware tamper bits (defined in *xc_types.h*) in the low-order 8 bits.
- If a host-side API call invokes the host operating system for service and the invocation fails, the host-side API call may return **HOST_OS_ERR** in the high-order 16 bits of the return code and the error code returned by the system call (or a portion of it) in the low-order 16 bits.

2.1.5 xcAdapterCount - count installed coprocessors

xcAdapterCount determines the number of cryptographic coprocessors installed on the host computer and returns the value to the caller. The state of the adapter is not considered when determining the count. The information comes from the PCIe interface. No communication with the adapter occurs.

Function prototype

```
unsigned int xcAdapterCount( xcAdapterNumber_t *pAdapterCount );
```

Input

On entry to this routine:

pAdapterCount contains the address of a buffer large enough to hold an item of type *xcAdapterNumber_t*.

Output

On successful exit from this routine:

**pAdapterCount* contains the number of coprocessors installed on the host.

Notes

An *xcOpenAdapter* call is not required prior to this request.

Coprocessors counted during boot

The number of coprocessors installed on the host is determined by the host device driver for the cryptographic coprocessor when the host is booted and is not updated to reflect any physical changes to the system (for example, removal of a coprocessor while the host is suspended or in hibernation) until a subsequent reboot or until stopping and then starting the host device driver.

xcAdapterNumber_t is arithmetic

An item of type *xcAdapterNumber_t* can be used in an arithmetic context (for example, as an array index or for-loop terminal value).

Return codes

Common return codes generated by this routine are:

HDDGood (that is, 0)	The operation was successful.
HDDError	The operation was unsuccessful.
HOST_OS_ERROR	An error occurred on a call to the host operating system (the low-order 16 bits contain the return code from the call that failed).

Refer to *xc_err.h* for a comprehensive list of return codes.

2.1.6 xcOpenAdapter - open channel to coprocessor

xcOpenAdapter establishes a communication channel between a host application and a specific coprocessor. The host application may interact with any application running on the coprocessor through the channel and may only interact with applications on coprocessors with which communication channels have been established.

Function prototype

```
Unsigned int xcOpenAdapter( xcAdapterNumber_t  AdapterNumber,
                            xcAdapterHandle_t *pAdapterHandle );
```

Input

On entry to this routine:

AdapterNumber identifies one of the cryptographic coprocessors installed on the host. *AdapterNumber* must contain an integer greater than or equal to zero and less than the value returned in the **pAdapterCount* output from a call to *xcAdapterCount* .

pAdapterHandle contains the address of a buffer large enough to hold an item of type *xcAdapterHandle_t*.

Output

On successful exit from this routine:

**pAdapterHandle* contains a handle that can be used in subsequent host-side API calls to identify the cryptographic coprocessor to which the call refers.

Notes

Assignment of numbers to coprocessors

The number assigned to a particular cryptographic coprocessor depends on the order in which information about devices in the system is presented to the device driver by the host operating system. At the present time there is no way to tell *a priori* which coprocessor will be assigned a given number. Adapter numbers are zero-based, so it is important to note that the first adapter in the system is adapter 0 instead of adapter 1.

Multiple communication channels

A host application may establish communication channels to more than one coprocessor by calling *xcOpenAdapter* multiple times with different *AdapterNumber* arguments. A host application may also establish more than one communication channel to a single coprocessor by calling *xcOpenAdapter* multiple times with the same *AdapterNumber* argument. In either case, each call to *xcOpenAdapter* returns a new handle in **pAdapterHandle*.

Return codes

Common return codes generated by this routine are:

HDDGood (that is, 0)	The operation was successful.
HDDInvalidParm	One or more inputs were not valid.
HDDTooManyOpens	The device driver or host operating system cannot create a new communication channel due to lack of resources.

HDDDeviceBusy	The device driver cannot open a communication channel to interact with an application on the adapter because another process on the host already has a channel open in order to interact with the adapter's system.
HDDError	The operation was unsuccessful.
HOST_OS_ERROR	An error occurred on a call to the host operating system (the low-order 16 bits contain the return code from the call that failed).

Refer to *xc_err.h* for a comprehensive list of return codes.

2.1.7 xcRequest - send request to coprocessor application

xcRequest sends a request across a communication channel to a specific application running on the target coprocessor and waits for and receives the application's reply.

Function prototype

```
Unsigned int xcRequest( xcAdapterHandle_t AdapterHandle,
                        xcRB_t            *pRequestBlock );
```

Input

On entry to this routine:

AdapterHandle identifies a communication channel to one of the cryptographic coprocessors installed on the host. *AdapterHandle* is the value returned from a call to *xcOpenAdapter*.

pRequestBlock contains the address of a request block whose fields are initialized as follows:

- *AgentID* identifies the coprocessor application to which the request should be delivered.
- *UserDefined* is passed to the coprocessor but is not otherwise examined by the host or coprocessor drivers. *UserDefined* typically specifies which of several services offered by the coprocessor application the host application wishes to invoke.
- *RequestControlBlkLength* is the length in bytes of the request control block. A request control block must be supplied (that is, *RequestControlBlkLength* must be greater than 64 bytes and must be a multiple of 8).

  RequestControlBlkLength must be small enough to ensure the length of the virtual packet created on the coprocessor is 64K bytes or less. See “Virtual packets” on page 3 for details. Call the CALC_MAX_TRANSFER_DATA macro, defined in cmncryt2.h, to calculate the maximum transfer size. This is not a constant but instead it depends on the contents of the request block. Both *RequestControlBlkLength* and the sum of *RequestControlBlkLength* + *RequestDataLength* are affected by this limit.
- *RequestControlBlkAddr* is the address of a buffer containing the request control block.
- *RequestDataLength* is the length in bytes of the request data.

  RequestDataLength may be 0. If it is nonzero, it must be small enough to ensure the length of the virtual packet created on the coprocessor is 64K bytes or less. See “Virtual packets” on page 3 for details. In particular:

  See the discussion for *RequestControlBlkLength* above about the maximum transfer size.
- *RequestDataAddress* is the address of a buffer containing the request data.

  If *RequestDataLength* is 0, *RequestDataAddress* should also be 0.
- *ReplyControlBlkLength* is the length in bytes of a buffer into which the coprocessor application writes the reply control block.

  ReplyControlBlkLength must be a multiple of 8, at least 64, and not more than 64K.
- *ReplyControlBlkAddr* is the address of the buffer into which the coprocessor application writes the reply control block. The buffer must be large enough to accommodate the coprocessor application's

reply.

- *ReplyDataLength* is the length in bytes of a buffer into which the coprocessor application writes the rbeply data. If no reply data is expected, *ReplyDataLength* should be 0. If it is nonzero, it must be a multiple of 8, at least 64, and not more than 64K.

  If *ReplyControlBlkLength* is 0, *ReplyDataLength* must also be 0.

- *ReplyDataAddr* is the address of the buffer into which the coprocessor application writes the reply data. The buffer must be large enough to accommodate the coprocessor application's reply.

  If *ReplyDataLength* is 0, *ReplyDataAddr* should also be 0.

- *PriorityWindow* specifies the request queue on which the request is placed. See “Request priority” on page 4 for details.

The host device driver ensures that *AgentID* and *UserDefined* are converted to the coprocessor's native endianness before the request is sent. That is, code on the coprocessor does not need to worry about the endianness of these values. The endiannness of any multibyte fields in the request control block and request data must be handled by the application itself (either on the host or on the coprocessor).

Output

On successful exit from this routine:

The following fields of **pRequestBlock* are changed as noted:

- *RequestID* is a unique identifier for the host request. This field is filled in by the device driver.
- *UserDefined* is set to the value specified by the coprocessor application in the *UserDef* field of the reply block when the coprocessor application calls *xcPutReply*. See “xcPutReply - send a reply to the host” on page 34 for details.
- *Status* is one of the following:
  - If the request was successfully delivered to the coprocessor application, *Status* is the status field from the coprocessor application's reply (that is, *putRep_t.status*) as described in more detail in “xcPutReply - send a reply to the host” on page 34.
  - If the request was not successfully delivered to the coprocessor application, or if the coprocessor application attempts to return more data than will fit in the buffer the host has allocated to hold it, Status is an error code from the Communication driver on the coprocessor (**XCCM_***).

The buffers referenced by *pRequestBlock->ReplyControlBlkAddr* and *pRequestBlock->ReplyDataAddress* could be updated. Note, however, that the lengths of the buffers (in *pRequestBlock->ReplyControlBlkLength* and *pRequestBlock->ReplyDataLength*) are not updated to reflect the number of bytes of data actually transferred from the coprocessor application.

The host device driver ensures that UserDefined and Status are converted to the host's native endianness before the reply is delivered to the host application. The endianness of any multibyte fields in the reply control block and the reply data must be handled by the application itself (either on the host or on the coprocessor).

Return codes

Common return codes generated by this routine are:

HDDGood (that is, 0)	The operation was successful.
HDDError	The operation was unsuccessful.
HOST_OS_ERROR	An error occurred on a call to the host operating system.

Common return codes placed in the *Status* field by the host device driver or the coprocessor application are:

HDDInvalidLength	The length of a buffer associated with the request is not a multiple of 8 or is out of range.
HDDInvalidParm	One or more inputs were not valid.
HDDDeviceBusy	Due to the lack of resources, a new request cannot be initiated until a pending request has completed. Try again later.
HDDRequstAborted	The request was aborted (for example, because an application on the coprocessor faulted).
HDDError	The operation was unsuccessful.

Common status codes returned from the Communication Driver are:

XCCM_UNDELIVERABLE	The identifier in *pRequestBlock->AgentID* does not match the identifier of any registered agent on the coprocessor.
XCCM_PROCESS_DIED	The coprocessor application process died or is dying.

Refer to *xc_err.h* for a comprehensive list of return codes.

2.1.8 xcCloseAdapter - close channel to coprocessor

xcCloseAdapter closes a communication channel that was previously opened through a call to *xcOpenAdapter*.

Function prototype

```
unsigned int xcCloseAdapter( xcAdapterHandle_t AdapterHandle );
```

Input

On entry to this routine:

AdapterHandle identifies a communication channel to one of the cryptographic coprocessors installed on the host. *AdapterHandle* must contain the handle returned in the **pAdapterHandle* output from a call to *xcOpenAdapter*.

Output

On successful exit from this routine:

The communication channel identified by *AdapterHandle* has been closed. The handle should not be subsequently passed as an argument to any host-side API function.

Return codes

Common return codes generated by this routine are:

HDDGood (that is, 0)	The operation was successful.
HDDError	The operation was unsuccessful.
HOST_OS_ERROR	An error occurred on a call to the host operating system (the low-order 16 bits contain the return code from the call that failed).

Refer to *xc_err.h* for a comprehensive list of return codes.

2.1.9 xcGetAdapterData - retrieve identification data from a coprocessor

xcGetAdapterData retrieves a coprocessor's "Vital Product Data".

Function prototype

```
unsigned int xcGetAdapterData( xcAdapterHandle_t AdapterHandle,
                               xcVpd_t           *pVpd );
```

Input

On entry to this routine:

AdapterHandle identifies a communication channel to one of the cryptographic coprocessors installed on the host. *AdapterHandle* is the value returned from a call to *xcOpenAdapter*.

pVpd contains the address of a buffer.

Output

On successful exit from this routine:

**pVpd* contains the coprocessor's identification data. The fields of **pVpd* are set as follows:

- *ds_tag* is 0x82.
- *ds_length* is the length in bytes of *pVpd->ds*. *This field is in little-endian byte order as required by PCI specification.*
- *ds* contains the (unquoted) ASCII text "IBM 4767-001 PCI-e Cryptographic Coprocessor".
- *vpdr_tag* is 0x90.
- *vpdr_length* is the length in bytes of the remainder of the coprocessor's identification data. *This field is in little-endian byte order as required by PCI specification.*
- *ec_tag* contains the (unquoted) ASCII text "EC".
- *ec_length* is the length in bytes of *pVpd->ec*.
- *ec* contains ASCII text that identifies the EC (engineering change) level of the coprocessor.
- *pn_tag* contains the (unquoted) ASCII text "PN".
- *pn_length* is the length in bytes of *pVpd->pn*.
- *pn* contains ASCII text that specifies the part number of the coprocessor.
- *fn_tag* contains the (unquoted) ASCII text "FN".
- *fn_length* is the length in bytes of *pVpd->fn*.
- *fn* contains ASCII text that specifies the FRU (field replaceable unit) number of the coprocessor.
- *ve_tag* contains the (unquoted) ASCII text "VE".
- *ve_length* is the length in bytes of *pVpd->ve*.
- *ve* contains ASCII text that identifies the secure part number of the coprocessor.

- *mf_tag* contains the (unquoted) ASCII text “MN”.
- *mf_length* is the length in bytes of *pVpd->mf*.
- *mf* contains ASCII text that identifies the location that manufactured the coprocessor.
- *sn_tag* contains the (unquoted) ASCII text “SN”.
- *sn_length* is the length in bytes of *pVpd->sn_hdr* and *pVpd->sn* together.
- *sn_hdr* contains ASCII text that specifies the coprocessor's serial number header.
- *sn* contains ASCII text that specifies the coprocessor's serial number.
- *rv_tag* contains the (unquoted) ASCII text “RV”.
- *rv_length* is the length in bytes of *pVpd->reserved*.
- *checksum* contains a checksum that covers everything in the structure prior to the checksum.
- *reserved* may be changed but its contents have no meaning.
- *end_tag* is 0x78.

Most of the fields in **pVpd* may contain the same values across multiple (or all) coprocessors. Only the *sn* field is guaranteed to be unique.

Return codes

Common return codes generated by this routine are:

HDDGood (that is, 0)	The operation was successful.
HDDError	The operation was unsuccessful.
HDDInvalidParm	The length was too short.
HOST_OS_ERROR	An error occurred on a call to the host operating system (the low-order 16 bits contain the return code from the call that failed).

Refer to *xc_err.h* for a comprehensive list of return codes.

2.1.10 xcGetHardwareInfo - retrieve a coprocessor's hardware version

xcGetHardwareInfo retrieves a coprocessor's hardware version.

Function prototype

```
unsigned int xcGetHardwareInfo( xcAdapterHandle_t AdapterHandle,
                                xcHWInfo_t       *pHWInfo );
```

Input

On entry to this routine:

AdapterHandle identifies a communication channel to one of the cryptographic coprocessors installed on the host. *AdapterHandle* is the value returned from a call to *xcOpenAdapter*.

pHWInfo contains the address of a buffer large enough to hold an item of type *xcHWInfo_t*.

Output

On successful exit from this routine:

**pHWInfo* contains the coprocessor's hardware version. The fields of **pHWInfo* are set as follows:

- *version* is the version of this structure. The following description is for version 1.
- *serial_num* is the null-terminated ASCII string of the serial number of the coprocessor.
- *part_num* is the null-terminated ASCII string of the part number of the coprocessor.
- *ec_level* is the null-terminated ASCII string of the engineering change level of the coprocessor.
- *fru_num* is the null-terminated ASCII string of the FRU number of the coprocessor.
- *mf_loc* is the null-terminated ASCII string of the manufacturing location of the coprocessor.
- *post0_ver* is the version of the POST 0 firmware.
- *post1_ver* is the version of the POST 1 firmware.
- *post2_ver* is the version of the POST 2 firmware.
- *mb0_ver* is the version of the Miniboot 0 firmware.
- *mb1_ver* is the version of the Miniboot 1 firmware.
- *fpga_rev* is the revision level of the firmware Field Programmable Gate Array firmware.
- *asic_rev* is the revision level of the Application Specific Integrated Circuit.
- *card_rev* is the revision level of the coprocessor hardware.

Return codes

Common return codes generated by this routine are:

HDDGood (that is, 0)	The operation was successful.
HDDError	The operation was unsuccessful.

HOST_OS_ERROR	An error occurred on a call to the host operating system (the low-order 16 bits contain the return code from the call that failed).

Refer to *xc_err.h* for a comprehensive list of return codes.

2.1.11 xcResetAdapter - reset a coprocessor

xcResetAdapter performs a hardware reset of a coprocessor. This reboots the embedded OS and restarts any applications running on the coprocessor.

Function prototype

```
unsigned int xcResetAdapter( xcAdapterHandle_t AdapterHandle );
```

Input

On entry to this routine:

AdapterHandle identifies a communication channel to one of the cryptographic coprocessors installed on the host. *AdapterHandle* is the value returned from a call to *xcOpenAdapter*.

Output

On successful exit from this routine the coprocessor has been reset.

Return codes

Common return codes generated by this routine are:

HDDGood (that is, 0)	The operation was successful.
HDDError	The operation was unsuccessful.
HOST_OS_ERROR	An error occurred on a call to the host operating system (the low-order 16 bits contain the return code from the call that failed).

Refer to *xc_err.h* for a comprehensive list of return codes.

2.1.12 xcGetTamperBits – Read the tamper bits of a coprocessor

xcGetTamperBits reads the tamper status of the coprocessor.

Function prototype

```
unsigned int xcGetTamperBits( xcAdapterHandle_t AdapterHandle,
                              xcHdwTmpr_t      *pHdwTamperBits );
```

Input

On entry to this routine:

AdapterHandle identifies a communication channel to one of the cryptographic coprocessors installed on the host. *AdapterHandle* is the value returned from a call to *xcOpenAdapter*.

pHdwTamperBits points to a buffer large enough to hold an item of type xcHdwTmpr_t.

Output

On successful exit from this routine:

**pHdwTamperBits* contains the tamper status of the coprocessor. The fields of pHdwTamperBits are set as follows:

- *length* contains the length of this structure.
- *Tmprbits* is an 8-bit field. The bits of this field are set as follows: most significant to least significant, if a bit is on the tamper has been detected: X-Ray Tamper, Intrusion Latch, Dead Battery, Temperature, Voltage, Card in Reset, Mesh Violation, and Low Battery Warning.

NOTE: any bits set other than the next most significant (Intrusion Latch), the 3rd least significant (Card in Reset), and the least significant (Low Battery), indicate that the card is permanently unusable.

Return codes

Common return codes generated by this routine are:

HDDGood (that is, 0)	The operation was successful.
HDDError	The operation was unsuccessful.
HOST_OS_ERROR	An error occurred on a call to the host operating system (the low-order 16 bits contain the return code from the call that failed).

Refer to *xc_err.h* for a comprehensive list of return codes.

3 Coprocessor-side API

The coprocessor-side portion of the IBM 4767 API allows an application running on a cryptographic coprocessor to request services from the various device drivers running on the coprocessor and to exchange information with an application running on the host on which the cryptographic coprocessor is installed.

coprocessor-side API calls can be used to perform various cryptographic operations (including DES and public key encryption and decryption, hashing, general large integer modular functions, and random number generation) and to receive requests from and return results to applications running on the host. A coprocessor application can also make calls directly to the Linux operating system that controls the coprocessor. For specific questions concerning the coprocessor's embedded OS, e-mail the Crypto team at crypto@us.ibm.com.

This chapter describes each of the functions supplied by the coprocessor-side API. Each description includes the function prototype (in C), the inputs to the function, the outputs returned by the function, and the most common return codes generated by the function.

3.1 General information

3.1.1 Coprocessor-side API functions

The coprocessor-side API includes functions in the following categories:

- Host communication
- Hash Operation functions
- Symmetric key operations
- Public key algorithms
- Large integer modular arithmetic
- Random number generator
- Coprocessor configuration
- Outbound authentication
- Miscellaneous functions

3.1.2 Common structure

Except for the host communication functions and the miscellaneous function, the request blocks for each function contain fields to handle the asynchronous operation. Each request block has a field named “cmn” which must contain the following pointers on input to the function:

- *pxcRequestHandle,*which is a pointer to a handle for this request. The *pThread* field in this handle must point to the Thread Handle received from the call to xcInitMappings.
- *pxcOpHandle* is a void **. Allocate a void * , set it to NULL, and pass the address of the void *.
- *pxcPrivateHandle,* which is a pointer to card application resources. The data in this handle is opaque to the xc* library routine.

- *pTimeBaseStart* is a pointer to a variable in user space to hold the starting timebase value of this function. This may be used for timing tests or debugging. Set to NULL if timing is not required.
- *pTimeBaseEnd* is a pointer to a variable in user space to hold the ending timebase value of the function. This may be used for timing tests or for debugging. Set to NULL if timing is not required.

On output from each function, the contents of the *cmn* parameter will have been updated as follows:

- *pxcOpHandle* will be a pointer to resources required by the xc* library module for this routine. The contents of this handle are opaque to the application, but the handle may be used (for example by the xcAsyncHandleCheck function) to distinguish between outstanding requests.
- *pTimeBaseStart* contains the starting timebase value of this function, if it was not NULL on input.
- pTimeBaseEnd contains the ending timebase value of the function, if it was not NULL on input.

3.1.3 Host communication functions

These functions allow a coprocessor application to interact with a host application and to obtain permission to request services from the coprocessor device drivers:

xcAttachWithCDUOption	Register a coprocessor application so that a host application can direct requests to it and so it can request cryptographic and other sensitive services from the coprocessor device drivers.
xcGetRequest	Get a request from the host application.
xcPutReply	Issue the reply to a host application's request.
xcQueryMRBstatus	Count active and available host requests.

3.1.4 Hash Operation Functions

These functions allow a coprocessor application to ask the Symmetric Key Cipher Hash (SKCH) driver to compute a condensed representation of a block of data using various standard hash algorithms:

xcSHA1	Compute the hash of a block of data using the Secure Hash Algorithm (SHA-1) as defined in FIPS Publication 180-1.
xcSHA2	Compute the hash of a block of data using the Secure Hash Algorithm (SHA-2) as defined in FIPS Publication 180-2.
xcSHA3	Compute the hash of a block of data using the Secure Hash Algorithm (SHA-3) as defined in FIPS Publication 180-4.

3.1.5 Symmetric Key Operation Functions

These functions allow a coprocessor application to ask the SKCH Driver to perform various operations with symmetric (secret) keys:

cmacGenerateNIST	Generate a Message Authentication Code according to

	NIST standard NIST SP 800-38B.
cmacVerifyNIST	Verify a Message Authentication Code according to the NIST standard NIST SP 800-38B.
cmacKeyBlockGenerateTR31	Wrap a cryptographic key using the NIST X9/TR-31 Interoperable Secure Key Exchange Key Block Specification for Symmetric Algorithms
cmacKeyBlockVerifyAndUnwrapTR31	Unwrap a cryptographic key using the NIST X9/TR-31 Interoperable Secure Key Exchange Key Block Specification for Symmetric Algorithms
xcAES	Encipher or decipher an arbitrary amount of data using the AES algorithm.
xcAESKeyWrapX9102 xcAESKeyUnwrapX9102 xcAESKeyWrapX9102Hash xcAESKeyUnwrapX9102Hash xcAESKeyWrapNIST and xcAESKeyUnwrapNIST	Wrap or unwrap a cryptographic key using the ANSI X9.102 or the NIST algorithm[1]. The ANSI X9.102 process can either incorporate up to 255 bytes of "associated data" or the hash of an arbitrary amount of associated data.
xcTDES	Encipher or decipher an arbitrary amount of data or generate a message authentication code using the triple-DES algorithm.
xcDES	Encipher or decipher an arbitrary amount of data or generate a message authentication code (MAC) using the DES algorithm.
xcDES8bytes	Encipher or decipher eight bytes of data using the DES algorithm.
xcDES3Key	Triple-encipher (wrap) or triple-decipher (unwrap) a cryptographic key using the DES algorithm.
xcVfpe	Perform VISA Format Preserving Encryption on a translated string of bytes or nibbles.

3.1.6 Public Key Algorithm Operation Functions

These functions allow a coprocessor application to request services from the Public Key Algorithm (PKA) Driver, which uses the coprocessor's large-integer modular math hardware to support public key cryptographic operations:

xcRSAKeyGenerate	Generate an RSA keypair.
xcRSA	Encipher or decipher a block of data using the RSA algorithm or wrap or unwrap an X9.31 encapsulated hash.
xcECCKeyGenerate	Generate an ECC keypair.

1 See *http://csrc.nist.gov/groups/ST/toolkit/documents/kms/key-wrap.pdf*

xcECC	Sign or verify the signature for an arbitrary amount of data using the ECC algorithm.

3.1.7 Large Integer Modular Math Operation Functions

These functions allow a coprocessor application to ask the PKA Driver to perform specific operations on large integers:

xcModMath	Perform a modular multiplication (C = A x B mod N), modular exponentiation (C = A^B mod N), or modular reduction (C = A mod N).

3.1.8 Random Number Generator Operation Functions

These functions allow a coprocessor application to request services from the Random Number Generator (RNG) Driver, which uses a hardware noise source and a pseudo-random number generator to deliver random bits that meet the applicable FIPS standards:

xcDRBGgenerate	Generate a random number.
xcDRBGinstantiate	Instantiate a DRBG random number generator
xcDRBGreseed	Reseed a single instance of the DRBG random number generator
xcDRBGuninstantiate	Uninstantiate a single instance of the DRBG random number generator, freeing its resources

3.1.9 Coprocessor Configuration Functions

These functions configure certain processor features or return information about the coprocessor:

xcGetConfig	Get information about the coprocessor.
xcClearILatch	Clear the coprocessor intrusion latch.
xcClearLowBatt	Clear the coprocessor low battery warning latch.

3.1.10 Outbound Authentication Functions

These functions allow a coprocessor application to authenticate itself to an application on the host:

xcOAGenerate **xcOAPrivOp**	Call the Outbound Authentication mechanism to perform one of the following tasks:

xcOAGetDir **xcOAGetCdert** **xcOADelete** **xcOAStatus**	• Count or list all certificates. • Retrieve a certificate • Generate a key pair and certificate list that contains the public half of the key pair. • Delete a certificate and the corresponding key pair • Perform an operation using one of the keys from a key pair in the OA certificate chain • Verify the signature in one certificate using the public key from another certificate • Obtain information about the status of and the software installed on the coprocessor

3.1.11 Header files

The prototypes for most coprocessor-side API functions are contained in *xc_api.h*. The prototypes for the Outbound Authentication functions are in *xc_oa.h*. Many other header files are used to create coprocessor applications, including *xc_types.h* and *xc_err.h*. These files are included in the IBM 4767 PCIe Application Program Development Toolkit. Refer to the *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Developer's Toolkit Guide* for details.

3.1.12 Sample code

Examples of the use of many of the coprocessor-side API functions can be found in the following subdirectories shipped with the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Developer's Toolkit:

- Host Communication Functions

```
samples/toolkit/oem/host
samples/toolkit/rte/host
samples/toolkit/skeleton/host
```

- Hash Functions

```
samples/toolkit/skeleton/hshserv
```

- Symmetric Key Functions

```
samples/toolkit/skeleton/desserv
```

- Public Key Algorithm Functions

```
samples/toolkit/skeleton/pkaserv
```

- Large Integer Modular Math Functions

```
samples/toolkit/skeleton/limserv
```

- Random Number Generator Functions
  `samples/toolkit/skeleton/rngserv`

- Coprocessor Configuration Functions
  `samples/toolkit/skeleton/adptserv`

- Outbound Authentication Functions
  `samples/toolkit/oa`

3.1.13 Device names and directories

/dev/ttyS0 Serial port

Mount point */bbram* is battery-backed RAM. Data in this directory is transparently encrypted on write and decrypted on read using a system-generated AES key.

Mount point /flash/USER is storage in Flash memory. Data in this directory is transparently encrypted on write and decrypted on read using a system-generated AES key.

3.1.14 Data format

Unless otherwise noted, all integers are in the native format for the processor. For the IBM 4767 coprocessor, the native format is big-endian.

3.1.15 Mapped kernel buffers and DMA-eligible buffers

Communication between an application on the host and an application on the coprocessor is generally performed via DMA – the host driver establishes request and reply buffers on the host, the coprocessor Communication driver establishes request and reply buffers on the coprocessor, and DMA hardware on the coprocessor transfers data between the appropriate buffers at the appropriate times.

The DMA hardware requires that a buffer that takes part in a DMA operation possess certain properties.

A coprocessor application cannot in general ensure that a buffer it creates has the necessary properties, so the Communication driver provides suitable buffers. In particular, when a coprocessor application asks the Communication driver to return the next request received from the host,

1. The Communication driver will map the kernel buffer containing the request into the application's address space. The Communication driver thereby avoids copying the request, which would otherwise be necessary.

   The request buffer is mapped read-write.

2. The Communication driver maps two kernel buffers into the application's address space to hold the application's response to the request (which may be in one or two pieces). By placing its response in these buffers the application avoids a copy operation when the Communication driver returns the response.

   The reply buffers are mapped read-write.

See “xcGetRequest – get a request from the host “ on page 30 for details.

Certain non-communication-related coprocessor operations (for example, hashing) are also performed using DMA. If a coprocessor application is designed so that input to such operations and/or the output they generate reside in a DMA-eligible buffer, throughput can be enhanced.

A piece of a mapped kernel buffer is DMA-eligible if the offset from the beginning of the mapped kernel buffer to the first byte of the piece is a multiple of 8.

A coprocessor application must not attempt to free a mapped kernel buffer.

3.2 Host communication functions

The functions described in this section allow a coprocessor application to interact with a host application and to obtain permission to request services from the coprocessor device drivers.

3.2.1 xcAttachWithCDUOption - register to receive requests

xcAttachWithCDUOption register a coprocessor application with the 4767 Communication driver so that the application can receive requests from the host. Registration is also required to request cryptographic and other sensitive services from the 4767 device drivers. Each embedded application must issue an attach with a distinct agent ID before calling any other function.

Function prototype

```
int xcAttachWithCDUOption( uint16_t agentID,
                           uint16_t CDUable );
```

Input

On entry to this routine:

agentID must uniquely identify the coprocessor application.

The following agent IDs are reserved by IBM and should not be used: 0x4341, 0x6866, 0x6867, and 0xFFF0 through 0xFFFF.

CDUable must be **NONCDUABLE**.

Output

None.

Return codes

On successful exit from this routine, the function returns a positive, nonzero file descriptor that can be passed as an argument to other functions. The coprocessor application can receive requests from the host and call other functions in the coprocessor-side API.

Common error codes generated by this routine are:

XCCM_ALREADY_ATTACHED	Another application has already attached using the agent ID passed on the call to *xcAttachWithCDUOption* or the current application has already successfully called *xcAttachWithCDUOption*.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.2.2 xcDetach - deregister a coprocessor application

xcDetach is a signoff request; it indicates that a coprocessor application no longer wishes to communicate with the host.

Function prototype

```
int xcDetach( int fd );
```

Input

On entry to this routine:

fd is a file descriptor returned by *xcAttachWithCDUOption*.

Output

None.

Notes

If there are any requests sent by the host to the coprocessor application that have not been delivered to the application when the applications calls *xcDetach*, those requests are canceled. The host application receives a reply whose *Status* field is **XCCM_UNDELIVERABLE**.

Return codes

Common return codes generated by this routine are:

DETACH_SUCCESS	The detach was successful.
DETACH_FAILED	The detach was unsuccessful.

Refer to xc_err.h for a comprehensive list of return codes.

3.2.3 xcGetRequest – get a request from the host

xcGetRequest checks the queue to determine whether the host has sent a request to the coprocessor application and returns the first available request.

Function prototype

```
int xcGetRequest(getReq_t *pGetReq );
```

Input

On entry to this routine:

pGetReq contains the address of a request block whose fields are initialized as follows:

- *pxcHandle* is the (xcthread_t) handle which was returned to the thread from the xcInitMappings call.
- *startMRB* is the index of the first request queue to examine to find a pending request (see “Request priority” on page 4). Typically this should be set to '0'.
- *endMRB* is the index of the last request ueue to examine to find a pending request (see “Request priority” on page 4). Typically this should be set to '1'.

  If *startMRB* is less than *endMRB*, *xcGetRequest* searches the request queues in increasing order of index (for example, 0, 1). If *startMRB* is greater than *endMRB*, *xcGetRequest* searches the request queues in decreasing order of index (for example, 1, 0). If the two fields are equal, *xcGetRequest* examines a single queue.
- *method* is 0 for asynchronous operation of this function, or 1 for synchronous operation.

Output

On successful exit from this routine:

The *pxcRequestHandle* field of **pGetReq* is set.

pxcRequestHandle is a pointer to an object whose fields are initialized as noted:

- *startMRB* is the same as the *pGetReq->startMRB.*
- *endMRB* is the same as the *pGetReq->endMRB.*
- *srcMRB* is the index of the request queue from which the virtual packet was obtained.
- *sizeHRB* is the size in bytes of the virtual packet.
- *pRequestUVirt* is a pointer to the data transferred from the device driver. This will be the Virtual Packet.
- *pReplyUVirt* is a pointer to the space where the Reply Control Block portion of the reply to the request received from the host should be placed for transfer (see xcPutReply - send a reply to the host” on page 34). The buffer is 64K bytes long and is mapped read-write. See “Mapped kernel buffers and DMA-eligible buffers” on page 26.
- *pReplyDataUVirt* is a pointer to the space where the Reply Data Block portion of the reply to the request received from the host should be placed for transfer (see xcPutReply - send a reply to the host” on page 34). The buffer is 64K bytes long and is mapped read-write. See “Mapped kernel

buffers and DMA-eligible buffers” on page 26.

- *method* is the same as *pGetReq->message*

Notes

A fiber that has called *xcGetRequest* must call *xcPutReply* to end the request before the fiber calls *xcGetRequest* again.

Return codes

Common return codes generated by this routine are:

GETREQUEST_SUCCESS	The operation was successful.
GETREQUEST_FAILED	The operation failed.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.2.4 xcInitMappings - prepare memory mappings for driver buffers

xcInitMappings directs the coprocessor Communication driver to map into the calling application's address space the buffers the driver uses to hold replies generated by the application. This reduces the amount of copying required during communication with the host and so improves performance. See “Mapped kernel buffers and DMA-eligible buffers” on page 26 for details.

Failure to use this function and the buffers it maps can reduce communication throughput by as much as 50%.

Function prototype

```
int xcInitMappings( int fd,
                    xcthread_t **ppxcHandle,
                    uint32_t     userFlag,
                    uint32_t     appStacks );
```

Input

On entry to this routine:

fd is a file descriptor returned by *xcAttachWithCDUOption*.

userFlag is APP_USER.

appStacks is the number of bytes of data (for example, 1MB) required for the application's stacks.

Output

On successful exit from this routine:

ppxcHandle is a pointer to a handle used by the coprocessor hardware and firmware to track the MRBs, HW registers, and scratchpad space for the thread.

Notes

Each application thread must call *xcInitMappings* once before calling *xcGetRequest*.

Note: 128K is the minimum value for *appstacks*, and is in general sufficient for 1 fiber. The maximum is 4M.

Return codes

Common return codes generated by this routine are:

INITMAP_SUCCESS	The initialization was successful.
INITMAP_FAILED	The initialization was unsuccessful.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.2.5 xcKillMappings - release memory mappings for driver buffers

xcKillMappings directs the coprocessor Communication driver to unmap the buffers mapped into the calling application's address space by a call to *xcInitMappings*.

Function prototype

```
int xcKillMappings( xcthread_t *pxcHandle );
```

Input

On entry to this routine:

pxcHandle is a file descriptor returned by *xcInitMappings*.

Output

None.

Notes

This function need only be called by a thread if the thread is going to die but the process to which it belongs is not. If the process itself ends the embedded OS will automatically undo the mappings for the process's threads.

Return codes

Common return codes generated by this routine are:

KILLMAP_SUCCESS	The function was successful.
KILLMAP_FAILED	The function was unsuccessful.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.2.6 xcPutReply - send a reply to the host

xcPutReply sends a reply to a host application's request. This effectively ends the request.

Function prototype

```
int xcPutReply( putRep_t *pPutRep );
```

Input

On entry to this routine:

pPutRep contains the address of a reply block whose fields are initialized as follows:

- *pxcRequestBlock* is a pointer to the xcRequestBlock_t pointer returned from the call to xcGetRequest.
  Note that the xcRequestBlock contains the srcMRB, startMRB, and other data required for the driver to route this reply to the correct host caller.
- *numTLP* is the number of elements in the array referenced by *pTLV*. There must be one element for each buffer the host expects to receive in response to its request (i.e., one for the reply control block and one for the reply data, if present).
- *pTLV* points to an array of structures, each of which defines a buffer that the coprocessor wishes to return as part of the reply. The structure elements are:
  - *tagLen.dataLen* , the low-order three bytes of which specify the length in bytes of the buffer, The maximum size of a reply buffer is 65536.
  - *tagLen.tag[0]* , which specifies whether the buffer is the reply control block (**TAG_OCPRB**) or the reply data (**TAG_REPDAT**); and
  - *vptr*, which points to the first byte of the buffer. *PTLV[0].vptr* must be *pxcRequestBlock->pReplyUVirt, and pTLV[1].vptr* must be *pxcRequestBlock->pReplyDataUVirt*.
- *userDef* is returned to the host application in the *UserDefined* field of the host's request block. See “xcRequest - send request to coprocessor application” on page 11 for details.
- *status* is returned to the host application in the *Status* field of the host's request block. That field can also be set in some circumstances by the Communication driver on the coprocessor; a coprocessor application should therefore refrain from setting status to a value that can be returned by the Communication driver. See “xcRequest - send request to coprocessor application” on page 11 for details.

Output

On successful exit from this routine:

The following fields of **pPutRep* are changed as noted:

- *tagChkRC* indicates whether or not the tags in the array referenced by *pTLV* were valid. Possible values are:

0	Success.
T_OCPRB_NOTFOUND	Host did not supply a buffer for the reply control block. The

	reply is discarded and the host application's call to *xcRequest* returns an error.
T_REPDAT_NOTFOUND	Reply included reply data but host did not supply a buffer for reply data. The reply is discarded and the host application's call to *xcRequest* returns an error.
T_OCPRB_DATA_TRUNC	Host supplied a buffer for the reply control block but it was too short. The reply is sent and the reply control block supplied by the coprocessor application is truncated on the host.
T_REPDAT_DATA_TRUNC	Host supplied a buffer for reply data but it was too short. The reply is sent and the reply data supplied by the coprocessor application is truncated on the host.

- *tankMRB0* is the number of host requests pending on queue 0. See “Request priority” on page 4.
- *tankMRB1* is the number of host requests pending on queue 1.

Notes

tagLen is a union containing a 4-byte string (*tag*) and a 32-bit integer (*dataLen*). A coprocessor application should take care to set *dataLen* first, and then set *tag[0]* to the appropriate tag. The other order will cause the tag to be overwritten by an unused byte in *dataLen*.

A coprocessor application should also not use *dataLen* after the value of the entire *taglen* union has been established. The presence of the tag in *tag[0]* means that the *dataLen* field is no longer the length of the corresponding buffer. Only the low-order 3 bytes of the *dataLen* field contain length information.

If the Communication driver mapped output buffers for the xcGetRequest call that started this request, the application must use the buffers mapped by the Communication driver. In particular, if the call to *xcGetRequest* for the request that this call ends returned:

- *T*he *vptr* field in the *pTLV* entry whose tag is **TAG_OCPRB** must be set to the value *xcGetRequest* returned in *pGetReq->pReplyUVirt*.
- *pGetReq->pReplyDataUVirt != NULL,* the *vptr* field in the *pTLV* entry whose tag is **TAG_REPDAT** must be set to the value *xcGetRequest* returned in *pGetReq->pReplyDataUVift*.

Failure to follow these rules will cause a resource leak.

Return codes

Common return codes generated by this routine are:

PUTREPLY_SUCCESS	The operation was successful.
PUTREPLY_FAILED	The operation was unsuccessful.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.3 Hash Operation Functions

The functions described in this section allow a coprocessor application to ask the Symmetric Key Cipher Hash (SKCH) driver to compute a condensed representation of a block of data using various standard hash algorithms.

A coprocessor application must call *xcAttachWithCDUOption* and *xcInitMappings* before calling any of the functions in this section.

3.3.1 xcSha1 - SHA-1 hash

xcSha1 computes the hash of a block of data using the Secure Hash Algorithm (SHA-1).

Function prototype

```
unsigned int xcSha1( xcSha1_RB_t *pSHA1_rb );
```

Input

On entry to this routine:

pSHA1_rb contains the address of a SHA-1 request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *options* controls the operation of the function and must be set to the logical OR of constants from the following category:

Operating mode

options must include exactly one of the following constants:

SHA1_MSGPART_ONLY	The input data constitutes the entire block of data to be hashed. The hash value is computed and returned.
SHA1_MSGPART_FIRST	The input data constitutes the first portion of a block of data to be hashed. See “Chained operations” on page 44.
SHA1_MSGPART_MIDDLE	The input data constitutes an additional portion of a block of data to be hashed. See “Chained operations” on page 44.
SHA1_MSGPART_FINAL	The input data constitutes the final portion of a block of data to be hashed. See “Chained operations” on page 44.

- *source_length* contains the length in bytes of the input data.

  If *options* specifies **SHA1_MSGPART_FIRST** or **SHA1_MSGPART_MIDDLE**, *source_length* must be a multiple of 64.

- *pSource* points to a buffer containing the input data.
- *hash_length* contains the length of the pHash buffer. This must be at least 20 bytes.
- *pHash* contains a partial hash returned by a prior call to *xcSha1*. *pHash* is used only if *options* specifies **SHA1_MSGPART_MIDDLE** or **SHA1_MSGPART_FINAL**. See “Chained operations” on page 44.
- *running_length* contains the number of bytes of input that have been processed by prior calls to *xcSHA1*. See “Chained operations” on page 44.

  running_length must be 0 if *options* specifies **SHA1_MSGPART_FIRST** or **SHA1_MSGPART_ONLY**.

Output

On successful exit from this routine:

The following fields of **pSHA1_rb* are changed as noted:

- *pHash* contains the SHA-1 hash of the input data.

  If, when *xcSha1* was called, *options* specified **SHA1_MSGPART_MIDDLE** or **SHA1_MSGPART_FINAL**, *pHash* also incorporates the value of *pHash* on entry to the routine.

- *running_length* reflects the number of bytes of input that were hashed.

  If, when *xcSha1* was called, options specified **SHA1_MSGPART_ONLY** or **SHA1_MSGPART_FIRST**, *running_length* is the number of bytes of input that were hashed.

  If when *xcSha1* was called *options* specified **SHA1_MSGPART_MIDDLE** or **SHA1_MSGPART_FINAL**, *running_length* is the value of *running_length* on entry to the routine increased by the number of bytes of input that were hashed.

Notes

If *pSource* points to a DMA-eligible buffer the hash operation may complete more quickly than would otherwise be the case. See “Mapped kernel buffers and DMA-eligible buffers” on page 26 for details.

Return codes

Common return codes generated by this routine are:

DMGood (that is, 0)	The operation was successful.
DMBadParm	A parameter was invalid (e.g, *options* specifies **SHA1_MSGPART_FIRST** or **SHA1_MSGPART_MIDDLE** but *source_length* is not a multiple of 64, or *options* specifies **SHA1_MSGPART_MIDDLE** or **SHA1_MSGPART_LAST** but *running_length* is 0).
DMBadAddr	Part of the buffer defined by *source.data_ptr* and *source_length* is not readable (i.e., lies in unmapped memory).

Refer to *xc_err.h* for a comprehensive list of return codes.

3.3.2 xcSha2 - SHA-2 hash

xcSha2 computes the hash of a block of data using the Secure Hash Algorithm (SHA-2).

Function prototype

```
unsigned int xcSha2( xcSha2_RB_t *pSHA2_rb );
```

Input

On entry to this routine:

pSHA2_rb contains the address of a SHA-2 request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *options* controls the operation of the function and must be set to the logical OR of constants from the following categories:

Operating mode

options must include exactly one of the following constants:

SHA2_MSGPART_ONLY	The input data constitutes the entire block of data to be hashed. The hash value is computed and returned.
SHA2_MSGPART_FIRST	The input data constitutes the first portion of a block of data to be hashed. See “Chained operations” on page 44.
SHA2_MSGPART_MIDDLE	The input data constitutes an additional portion of a block of data to be hashed. See “Chained operations” on page 44.
SHA2_MSGPART_FINAL	The input data constitutes the final portion of a block of data to be hashed. See “Chained operations” on page 44.

Hash method

options must include exactly one of the following constants:

SHA224_METHOD	Compute a 224-bit hash of the input.
SHA256_METHOD	Compute a 256-bit hash of the input.
SHA384_METHOD	Compute a 384-bit hash of the input.
SHA512_METHOD	Compute a 512-bit hash of the input
SHA512_224_METHOD	Compute a 224-bit hash of the input using the SHA-512 method.
SHA512_256_METHOD	Compute a 256-bit hash of the input using the SHA-512 method.

- *source_length* contains the length in bytes of the input data.

  If *options* specifies **SHA2_MSGPART_FIRST** or **SHA2_MSGPART_MIDDLE**, *source_length* must be a multiple of 64.

- *pSource* points to a buffer containing the input data.
- *hash_length* is the number of bytes in the buffer pointer to by *pHash.* If *options* includes **SHA224_METHOD** or **SHA256_METHOD**, *hash_length* must be at least 32. Otherwise, *hash_length* must be at least 64.
- *pHash* is a pointer to a buffer at least *hashlength* bytes long. If *options* specifies **SHA2_MSGPART_MIDDLE** or **SHA2_MSGPART_FINAL,** **pHash* contains a partial hash returned by a prior call to *xcSha2*. See “Chained operations” on page 44.
- *KH* and *KL* form a 16-byte integer containing the number of bytes of input that have been processed by prior calls to *xcSHA2*. See “Chained operations” on page 44.

  KH is the high-order (most significant) portion of the integer and KL is the low-order (least significant) portion of the integer.

  KH and KL must be 0 if options specifies **SHA2_MSGPART_FIRST** or **SHA2_MSGPART_ONLY**.

- *magic* is 0xDECAF123.

Output

On successful exit from this routine:

The following fields of **pSHA2_rb* are changed as noted:

- *pHash* contains the hash of the input data, using the hash algorithm specified in options.

  If, when *xcSha2* was called, *options* specified **SHA2_MSGPART_MIDDLE** or **SHA2_MSGPART_FINAL**, *pHash* also incorporates the value of *pHash* on entry to the routine.

- *KH* and *KL* form a 16-byte integer reflecting the number of bytes of input that were hashed.

  If, when *xcSha2* was called, *options* specified **SHA2_MSGPART_ONLY** or **SHA2_MSGPART_FIRST**, the integer is the number of bytes of input that were hashed.

  If, when *xcSha2* was called, *options* specified **SHA2_MSGPART_MIDDLE** or **SHA2_MSGPART_FINAL**, the integer is the value of the integer on entry to the routine increased by the number of bytes of input that were hashed.

Notes

If *pSource* points to a DMA-eligible buffer the hash operation may complete more quickly than would otherwise be the case. See “Mapped kernel buffers and DMA-eligible buffers” on page 26 for details.

Return codes

Common return codes generated by this routine are:

DMGood (that is, 0)	The operation was successful.
DMNotAuth	The coprocessor application is not authorized to perform SHA operations (for example, because it has not called

	xcAttachWithCDUOption).
DMBadFlags	The options argument is not valid.
DMBadParm	A parameter was invalid (e.g, *options* specifies **SHA2_MSGPART_FIRST** or **SHA2_MSGPART_MIDDLE** but *source_length* is not a multiple of 64, or *options* specifies **SHA2_MSGPART_MIDDLE** or **SHA2_MSGPART_LAST** but *KH* and *KL* are both 0).
DMBadAddr	Part of the buffer defined by *source.data_ptr* and *source_length* is not readable (i.e., is not mapped).

Refer to *xc_err.h* for a comprehensive list of return codes.

3.3.3 xcSha3 - SHA-3 hash

xcSha3 computes the hash of a block of data using the Secure Hash Algorithm (SHA-3).

Function prototype

```
unsigned int xcSha3( xcSha3_RB_t *pSHA3_rb );
```

Input

On entry to this routine:

pSHA3_rb contains the address of a SHA-3 request block whose fields are initialized as follows:

- cmn is set as described in Common structure on page 21.
- *options* controls the operation of the function and must be set to the logical OR constants from the following categories:

SHA-3 Option

options must include **SHA23_METHOD**

Operating mode

options must include exactly one of the following constants:

SHA3_MSGPART_ONLY	The input data constitutes the entire block of data to be hashed. The hash value is computed and returned.
SHA3_MSGPART_FIRST	The input data constitutes the first portion of a block of data to be hashed. See “Chained operations” on page 44.
SHA3_MSGPART_MIDDLE	The input data constitutes an additional portion of a block of data to be hashed. See “Chained operations” on page 44.
SHA3_MSGPART_FINAL	The input data constitutes the final portion of a block of data to be hashed. See “Chained operations” on page 44.

Hash method

options must include exactly one of the following constants:

SHA224_METHOD	Compute a SHA 224 hash of the input.
SHA256_METHOD	Compute a SHA 256 hash of the input.
SHA384_METHOD	Compute a 384-bit hash of the input.
SHA512_METHOD	Compute a 512-bit hash of the input.

- *source_length* contains the length in bytes of the input data.

  If *options* specifies **SHA3_MSGPART_FIRST** or **SHA3_MSGPART_MIDDLE**, *source_length* must be a multiple of 64.

- *pSource* points to a buffer containing the input data.
- *hash_length* contains the length in bytes of the buffer pointed to by *pHash.* If *options* specifies **SHA3_MSGPART_FIRST** or **SHA3_MSGPART_MIDDLE** or **SHA3_MSGPART_FINAL** this must be at least 200 bytes.
- *pHash* contains a partial hash returned by a prior call to *xcSha3*. *pHash* is used only if *options* specifies **SHA3_MSGPART_MIDDLE** or **SHA3_MSGPART_FINAL**. See “Chained operations” on page 44.
- *KH* and *KL* form a 16-byte integer containing the number of bytes of input that have been processed by prior calls to *xcSHA3*. See “Chained operations” on page 44.

  KH is the high-order (most significant) portion of the integer and KL is the low-order (least significant) portion of the integer.

  KH and KL must be 0 if options specifies **SHA3_MSGPART_FIRST** or **SHA3_MSGPART_ONLY**.

- *magic* is 0xDECAF123.

Output

On successful exit from this routine:

The following fields of **pSHA3_rb* are changed as noted:

- *pHash* contains the hash of the input data, using the hash algorithm specified in *options,* left-justified in the buffer.

  If, when *xcSha3* was called, *options* specified **SHA3_MSGPART_MIDDLE** or **SHA3_MSGPART_FINAL**, *hash_value* also incorporates the value of *hash_value* on entry to the routine.

- *KH* and *KL* form a 16-byte integer reflecting the number of bytes of input that were hashed.

  If, when *xcSha3* was called, *options* specified **SHA3_MSGPART_ONLY** or **SHA3_MSGPART_FIRST**, the integer is the number of bytes of input that were hashed.

  If, when *xcSha3* was called, *options* specified **SHA3_MSGPART_MIDDLE** or **SHA3_MSGPART_FINAL**, the integer is the value of the integer on entry to the routine increased by the number of bytes of input that were hashed.

Notes

If p*Source* points to a DMA-eligible buffer the hash operation may complete more quickly than would otherwise be the case. See “Mapped kernel buffers and DMA-eligible buffers” on page 26 for details.

3.3.4 Chained operations

A block of data to be hashed may be processed in a single operation. It may be necessary, however, to break the operation into several steps, each of which processes only a portion of the block. For example, an application may want to compute a hash that covers several discontiguous fields in a structure.

A chained operation is initiated by calling *xcSHAn* (where n is either 1, 2, or 3) with **SHAn_MSGPART_FIRST** specified in the options field of the SHA request block and the first piece of the block of data to hash in the buffer referenced by the *source.data_ptr* field of the request block. On return, the *hash_value* field of the request block contains the hash for the first piece of data and the *running_length* or *KH* and *KL* fields contain the number of bytes of data processed. The *hash_value* and length (*running_length*/*KH* and *KL*) fields must be preserved and passed to *xcSHAn* when the next piece of the block of data to hash is processed.

Subsequent pieces of the block are processed by calling *xcSHAn* with **SHAn_MSGPART_MIDDLE** specified in the options field of the SHA request block (**SHAn_MSGPART_FINAL** must be specified if the piece in question is the last). The piece is in the buffer referenced by the *source.data_ptr* field of the request block. The *hash_value* and length (*running_length*/*KH* and *KL*) fields must contain the values returned in those fields by the call to *xcSHAn* that processed the previous piece of the block. The function hashes the piece and updates the *hash_value* and length (*running_length*/*KH* and *KL*) fields appropriately.

For SHA-2 and SHA-3 operations, the options field of the SHA request block must specify the same hash method (**SHA224_METHOD** or **SHA256_METHOD**) for each piece of the block of data to be hashed.

3.4 AES Operation Functions

The functions described in this section allow a coprocessor application to ask the Symmetric Key Cipher Hash (SKCH) driver to perform various encryption, decryption, MAC generation, and key wrapping and unwrapping operations using the Advanced Encryption Standard (AES) algorithm.

A coprocessor application must call *xcAttachWithCDUOption* before calling any of the functions in this section.

3.4.1 xcAES - AES encryption / decryption / MAC

xcAES enciphers and deciphers an arbitrary amount of data using the AES (Advanced Encryption Standard) algorithm. Data can be enciphered in either Cipher Block Chaining (CBC) mode or Electronic Code Book (ECB) mode. *xcAES* can also generate a message authentication code (MAC). Keys may be 128, 192, or 256 bits in length.

Function prototype

```
int xcAES( xcAES_RB_t *pAES_rb );
```

Input

On entry to this routine:

pAES_rb contains the address of an AES request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *options* controls the operation of the function and must be set to the logical OR of constants from the following categories:

Operation

options must include exactly one of the following constants:

AES_ENCRYPT	Encrypt the input.
AES_DECRYPT	Decrypt the input.
AES_MAC	Generate a message authentication code (MAC) for the input.

If **AES_MAC** is specified, **AES_ECB_MODE** must not be specified. **AES_CBC_MODE** will be assumed.

Key length

options must include exactly one of the following constants:

AES_128BIT_KEY	The key length is 128 bits (16 bytes).
AES_192BIT_KEY	The key length is 192 bits (24 bytes).
AES_256BIT_KEY	The key length is 256 bits (32 bytes).

Chaining mode

options must include exactly one of the following constants:

AES_CBC_MODE	Use outer Cipher Block Chaining (CBC) mode.
AES_ECB_MODE	Use Electronic Code Book (ECB) mode.

AES_ECB_MODE must not be specified if **AES_MAC** is specified. **AES_CBC_MODE** will be assumed.

- *key* points to a buffer containing the key to use for the operation (an item of type *xcAES_key_t*). The length of the key is specified by the Key Length bit in *options*. A key that is shorter than the buffer is left-justified within the buffer (i.e., the first byte of the key resides in the first byte of the buffer, regardless of the key length).
- *init_v* points to a buffer containing the initial vector for the operation (an item of type *xcAES_vector_t*) if *options* specifies **AES_CBC_MODE** and is unused otherwise.
- *term_v* points to a buffer in which an item of type *xcAES_vector_t* can be stored.
- *source_length* is the length in bytes of the data to be processed by *xcAES*. *source_length* must be a multiple of 16. If *source_length* is 0, either *prepad_length* or *postpad_length* must be non-zero.
- *pSource* points to a buffer containing the data to be processed by *xcAES*. The “input data” to *xcAES* consists of the contents of the *prepad* buffer (if any) followed by the contents of this buffer (*pSource)* followed by the contents of the *postpad* buffer (if any).
- *destination_length* is the length in bytes of the buffer referenced by *pDestination*. *destination_length* must be at least as large as the *source_length* + *prepad_length* + *postpad_length,* unless *options* contains **AES_MAC**.
- *pDestination* points to a writeable buffer.
- *prepad* is a pointer to a buffer containing data to be prepended to the data in *pSource* by *xcAES*. This may be a null pointer if *prepad_length* is 0.
- *prepad_length* is the length in bytes of the data to process prior to that in *pSource.* This must be 0 or 16.
- *postpad_length* is the length iin bytes of the *postpad* buffer. This must be 0, 16, or 32.
- *postpad* is a pointer to a buffer containing data to be processed following the processing of the data in *pSource.* If *postpad_length* is 0, this may be a null pointer.

Output

On successful exit from this routine:

The following fields of **pAES_rb* are changed as noted:

- *term_v* contains

- the message authentication code (MAC) generated from the input data if *options* specifies **AES_MAC**.
- the initialization vector for the next AES operation if options specifies **AES_CBC_MODE**[2] .

term_v is undefined otherwise.

- The buffer referenced by *pDestination* contains
  - The input data encrypted using *key* (and *init_v* if *options* specifies **AES_CBC_MODE**) if options specifies **AES_ENCRYPT**.
  - The input data decrypted using *key* (and *init_v* if *options* specifies **AES_CBC_MODE**) if options specifies **AES_DECRYPT**.

  The buffer referenced by *pDestination* is undefined otherwise.

Notes

The length of the input data may be less than *destination_length*. In this case, any excess bytes at the end of the output buffer are not affected by *xcAES*.

If *pSource* and/or *pDestination* point to a DMA-eligible buffer the AES operation may complete more quickly than would otherwise be the case. See “Mapped kernel buffers and DMA-eligible buffers” on page 26 for details.

The buffers defined by *pSource/source_length* and *pDestination/destination_length* should not overlap.

Return codes

Common return codes generated by this routine are:

DMGood (that is, 0)	The operation was successful.
DMNotAuth	The coprocessor application is not authorized to perform AES operations (for example, because it has not called *xcAttachWithCDUOption*).
DMBadFlags	The options argument is not valid.
DMBadParm	The length of the input data is invalid (for example, not a multiple of 16), the length of the input data exceeds the length of the output buffer, or *source_length* is zero and no padding data was specified.
DMBadAddr	Part of the buffer defined by *pSource* and *source_length* is not readable, part of the buffer defined by *pDestination* and *destination_length* is not writeable, or a field in the request block cannot be accessed.

Refer to *xc_err.h* for a comprehensive list of return codes.

2 If *options* specifies both **AES_MAC** and **AES_CBC_MODE**, the MAC and the initialization vector are the same value.

3.4.2 xcAESKeyWrap* and xcAESKeyUnwrap* - AES key wrapping

The functions described in this section allow an application to wrap an AES key and to unwrap a wrapped AES key. Two distinct wrapping standards are supported:

- ANSI ASC X9.102 (June 2008) (http://www.x9.org)
- NIST (November 16, 2001) (*http://csrc.nist.gov/groups/ST/toolkit/documents/kms/key-wrap.pdf*)

Wrapping operation

In general, a wrapping operation imbeds the key to be wrapped within a formatted buffer, then uses a second key (the "wrapping key") to perform a cryptographic operation that transforms the formatted buffer. Although the cryptographic operation in question is often simply encryption or decryption, both the X9.102 and NIST standards use a more complex algorithm. See either standard for details (both standards use the same algorithm).

Under both standards the length of the wrapped key matches the length of the corresponding formatted buffer.

Sound cryptographic practice requires that the wrapping key be at least as cryptographically strong as any key it is used to wrap. The functions described in this section neither check nor enforce this.

X9.102 formatted buffer

The formatted buffer used by X9.102 as input to the wrapping operation is shown below:

ICV (6 bytes)	PadLen (1 byte)	Hlen (1 byte)	H (Hlen bytes) or H1 (4 bytes)/H2 (Hlen-4 bytes)	Key to be wrapped	Padding (Padlen bits, all zeros)

ICV is the "Integrity Check Value" and consists of six bytes of 0xA6.

PadLen specifies the number of bits of padding at the end of the formatted buffer; between 0 and 63 bits are added to ensure the total length in bits of the formatted buffer is a multiple of 64.

The user may supply a (possibly zero-length) string of "associated data". This string may be incorporated directly into the formatted buffer (copied into H). Or it may be hashed and its hash incorporated into the formatted buffer (copied into H2; H1 contains the options supplied to the hash routine).

The X9.102 standard specifies that a single wrapping key should not be used to wrap more than 2^{48} distinct (i.e., different) X9.102 formatted buffers. The functions described in this section neither check nor enforce this.

NIST formatted buffer

The formatted buffer used by NIST as input to the wrapping operation is shown below:

ICV (8 bytes)	Key to be wrapped

ICV is the "Integrity Check Value" and consists of eight bytes of 0xA6.

The length in bits of the key to be wrapped must be a multiple of 64.

3.4.3 xcAESKeyWrapX9102

xcAESKeyWrapX9102 wraps an AES key according to the X9.102 standard. This function allows the caller to supply up to 255 bytes of associated data. This data is incorporated into the formatted buffer without change (see “X9.102 formatted buffer” on page 48):

Function prototype

```
int xcAESKeyWrapX9102( xcAESKW_X9102_t *pAESKWrap );
```

Input

On entry to this routine:

pAESKWrap contains the address of an AES key wrap request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *pWrapKey* points to a buffer containing the wrapping key (an item of type *xcAES_key_t*), i.e., the key to use to wrap the AES key that is the object of the operation.
- *wrapKeyLen* specifies the length in bits of the wrapping key. *wrapKeyLen* must be 128, 192, or 256.
- *pAsData* points to a buffer containing the associated data for the X9.102 formatted buffer. See “X9.102 formatted buffer” on page 48 for details.

  pAsData may be NULL.
- *asDataLen* is the length in bytes of the buffer referenced by *pAsData*. If *pAsData* is NULL, *asDataLen* must be 0.
- *pKeyData* points to a buffer containing the AES key to be wrapped.
- *keyDataLen* is the length in bits of the key referenced by *pKeyData*.

  If *keyDataLen* is not a multiple of 8, the key occupies the leftmost bits of the buffer referenced by *pKeyData* (for example, if *keyDataLen* is 15, the least-significant bit in the last byte of the buffer refereced by *pKeyData* is not used).
- *pWrapData* points to a writeable buffer large enough to hold the wrapped key. See “X9.102 formatted buffer” on page 48 for details.
- *pWrapDataLen* points to a writeable buffer that contains the length in bytes of the buffer referenced by *pWrapData*.

Output

On successful exit from this routine:

The following fields of **pAESKWrap* are changed as noted:

- **pWrapDataLen* contains the actual length in bytes of the wrapped key.

- The buffer defined by *pWrapData* contains the wrapped key.

Return codes

Common return codes generated by this routine are:

DMGood (that is, 0)	The operation was successful.
DMNotAuth	The coprocessor application is not authorized to perform AES key wrapping operations (for example, because it has not called *xcAttachWithCDUOption*).
DMBadFlags	An input argument was invalid (for example, *pAESKWrap* is NULL or *pAESKWrap->hashOption* specifies more than one algorithm or a buffer is too small).

Refer to *xc_err.h* for a comprehensive list of return codes.

3.4.4 xcAESKeyWrapX9102Hash

xcAESKeyWrapX9102Hash wraps an AES key according to the X9.102 standard. This functions allows the caller to supply up to 65535 bytes of associated data. This data is hashed. The options passed to the hash routine and the resulting hash are incorporated into the formatted buffer (see “X9.102 formatted buffer” on page 48):

Function prototype

```
int xcAESKeyWrapX9102Hash( xcAESKW_X9102hash_t *pAESKWrap );
```

Input

On entry to this routine:

pAESKWrap contains the address of an AES key wrap request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *pWrapKey* points to a buffer containing the wrapping key (an item of type *xcAES_key_t*), i.e., the key to use to wrap the AES key that is the object of the operation.
- *wrapKeyLen* specifies the length in bits of the wrapping key. *wrapKeyLen* must be 128, 192, or 256.
- *pAsData* points to a buffer containing the associated data for the X9.102 formatted buffer. See “X9.102 formatted buffer” on page 48 for details.

  pAsData may be NULL.
- *asDataLen* is the length in bytes of the buffer referenced by *pAsData*. If *pAsData* is NULL, *asDataLen* must be 0.
- *hashOption* specifies how the buffer referenced by *pAsData* is to be hashed prior to incorporation into the X9.102 formatted buffer *hashOption* may take one of the following values:
  - **SHA224_METHOD** - Compute a SHA 224 hash of the buffer referenced by pAsData.
  - **SHA256_METHOD** - Compute a SHA 256 hash of the buffer referenced by pAsData.

  If *pAsData* is NULL and *asDataLen* is 0, *xcAESKeyWrap9201Hash* computes the hash of a zero-length string.
- *pHashAsData* points to a writeable buffer large enough to hold the result of the hash operation.
- *pHashAsDataLen* points to a writeable buffer that contains the length in bytes of the buffer referenced by *pHashAsData*.
- *pKeyData* points to a buffer containing the AES key to be wrapped.
- *keyDataLen* is the length in bits of the key referenced by *pKeyData*.

  If *keyDataLen* is not a multiple of 8, the key occupies the leftmost bits of the buffer referenced by *pKeyData* (for example, if *keyDataLen* is 15, the least-significant bit in the last byte of the buffer refereced by *pKeyData* is not used).

- *pWrapData* points to a writeable buffer large enough to hold the wrapped key. See “X9.102 formatted buffer” on page 48 for details.
- *pWrapDataLen* points to a writeable buffer that contains the length in bytes of the buffer referenced by *pWrapData*.

Output

On successful exit from this routine:

The following fields of **pAESKWrap* are changed as noted:

- **pHashAsDataLen* contains the actual length in bytes of the hash that was incorporated into the X9.102 formatted buffer.
- **pWrapDataLen* contains the actual length in bytes of the wrapped key.
- The buffer defined by *pHashAsData* contains the hash that was incorporated into the X9.102 formatted buffer.
- The buffer defined by *pWrapData* contains the wrapped key.

Notes

The value of *pAESKWrap->hashOption* is placed into the H1 field of the X9.102 formatted buffer in big-endian order.

Return codes

Common return codes generated by this routine are:

DMGood (that is, 0)	The operation was successful.
DMNotAuth	The coprocessor application is not authorized to perform AES key wrapping operations (for example, because it has not called *xcAttachWithCDUOption*).
DMBadFlags	An input argument was invalid (for example, *pAESKWrap* is NULL or *pAESKWrap->hashOption* specifies more than one algorithm or a buffer is too small).

Refer to *xc_err.h* for a comprehensive list of return codes.

3.4.5 xcAESKeyUnwrapX9102

xcAESKeyUnwrapX9102 unwraps an AES key wrapped according to the X9.102 standard. This function allows the caller to supply up to 255 bytes of associated data. This data must match without change the data that was incorporated into the X9.102 formatted buffer prior to the wrapping operation.(see “X9.102 formatted buffer” on page 48):

Function prototype

```
int xcAESKeyUnwrapX9102( xcAESKUW_X9102_t *pAESKUWrap );
```

Input

On entry to this routine:

pAESKUWrap contains the address of an AES key unwrap request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *pUnwrapKey* points to a buffer containing the unwrapping key (an item of type *xcAES_key_t*), i.e., the key to use to unwrap the wrapped AES key.that is the object of the operation.
- *unwrapKeyLen* specifies the length in **bits** of the unwrapping key. *unwrapKeyLen* must be 128, 192, or 256.
- *pAsData* points to a buffer containing the associated data that the agent that wrapped the key is expected to have put into the X9.102 formatted buffer. See “X9.102 formatted buffer” on page 48 for details.

  pAsData may be NULL.
- *asDataLen* is the length in bytes of the buffer referenced by *pAsData*. If *pAsData* is NULL, *asDataLen* must be 0.
- *pWrapData* points to a buffer containing the wrapped AES key.
- *wrapDataLen* is the length in bytes of the buffer referenced by *pWrapData*.
- *pClearKey* points to a writeable buffer large enough to hold the unwrapped key.
- *pClearKeyLen* points to a writeable buffer that contains the length in **bits** of the buffer referenced by *pClearKey*.

Output

On successful exit from this routine:

The following fields of **pAESKUWrap* are changed as noted:

- **pClearKeyLen* contains the actual length in **bits** of the unwrapped key.

The buffer defined by *pClearKey* contains the unwrapped key. If **pClearKeyLen* is not a multiple of 8, the key occupies the leftmost bits of the buffer referenced by *pClearKey* (for example, if **pClearKeyLen* is 15, the least-significant bit in the last byte of the buffer referenced by *pClearKey* is not used).

Notes

The X9.102 formatted buffer produced by the unwrapping operation is verified using the X9.102 integrity check method:

1. Each of the first six bytes of the buffer must have the value 0xA6.
2. The pad length must be less than or equal to 63.
3. The length of the buffer must be large enough to contain all the required elements.
4. For *xcAESKeyUnwrapX9102*, the associated data in the buffer must match the contents of the buffer defined by *pAESKUWrap->pAsData/pAESKUWrap->asDataLen*.
5. All of the pad bits must be zero.

If any of these conditions is not satisfied, the function returns **DMBadX9102KUW**.

Return codes

Common return codes generated by this routine are:

DMGood (that is, 0)	The operation was successful.
DMNotAuth	The coprocessor application is not authorized to perform AES key wrapping operations (for example, because it has not called *xcAttachWithCDUOption*).
DMBadFlags	An input argument was invalid (for example, *pAESKWrap* is NULL or a buffer is too small).
DMBadX9102KUW	Unwrap operation did not produce a valid X9.102 formatted buffer.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.4.6 xcAESKeyUnwrapX9102Hash

xcAESKeyUnwrapX9102Hash unwrap an AES key wrapped according to the X9.102 standard. This function allows the caller to supply up to 65535 bytes of associated data. This data is hashed. The options passed to the hash routine and the resulting hash must match the corresponding items that were incorporated into the X9.102 formatted buffer prior to the wrapping operation. (see “X9.102 formatted buffer” on page 48):

Function prototype

```
int xcAESKeyUnwrapX9102Hash( xcAESKUW_X9102hash_t *pAESKUWrap );
```

Input

On entry to this routine:

pAESKUWrap contains the address of an AES key unwrap request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *pUnwrapKey* points to a buffer containing the unwrapping key (an item of type *xcAES_key_t*), i.e., the key to use to unwrap the wrapped AES key.that is the object of the operation.
- *unwrapKeyLen* specifies the length in **bits** of the unwrapping key. *unwrapKeyLen* must be 128, 192, or 256.
- *pAsData* points to a buffer containing the associated data that the agent that wrapped the key is expected to have put into the X9.102 formatted buffer. See “X9.102 formatted buffer” on page 48 for details.

  pAsData may be NULL.
- *asDataLen* is the length in bytes of the buffer referenced by *pAsData*. If *pAsData* is NULL, *asDataLen* must be 0.
- *hashOption* specifies how the buffer referenced by *pAsData* is to be hashed prior to comparison with the appropriate portion of the X9.102 formatted buffer. *hashOption* may take one of the following values:
  - **SHA224_METHOD** - Compute a SHA 224 hash of the buffer referenced by *pAsData*.
  - **SHA256_METHOD** - Compute a SHA 256 hash of the buffer referenced by *pAsData*.

  If *pAsData* is NULL and *asDataLen* is 0, *xcAESKeyUnwrap9201Hash* computes the hash of a zero-length string.
- *pWrapData* points to a buffer containing the wrapped AES key.
- *wrapDataLen* is the length in bytes of the buffer referenced by *pWrapData*.
- *pClearKey* points to a writeable buffer large enough to hold the unwrapped key.
- *pClearKeyLen* points to a writeable buffer that contains the length in **bits** of the buffer referenced by *pClearKey*.

Output

On successful exit from this routine:

The following fields of **pAESKUWrap* are changed as noted:

- **pClearKeyLen* contains the actual length in **bits** of the unwrapped key.

The buffer defined by *pClearKey* contains the unwrapped key. If **pClearKeyLen* is not a multiple of 8, the key occupies the leftmost bits of the buffer referenced by *pClearKey* (for example, if **pClearKeyLen* is 15, the least-significant bit in the last byte of the buffer refereced by *pClearKey* is not used).

Notes

The X9.102 formatted buffer produced by the unwrapping operation is verified using the X9.102 integrity check method:

1. Each of the first six bytes of the buffer must have the value 0xA6.
2. The pad length must be less than or equal to 63.
3. The length of the buffer must be large enough to contain all the required elements.
4. The associated data in the buffer must match the value of *pAESKUWrap->hashOption* (in big-endian order) concatenated with the hash value (computed using the algorithm dictated by *pAESKUWrap->hashOption*) of the buffer defined by *pAESKUWrap->pAsData/pAESKUWrap->asDataLen*.
5. All of the pad bits must be zero.

If any of these conditions is not satisfied, the function returns **DMBadX9102KUW**.

Return codes

Common return codes generated by this routine are:

DMGood (that is, 0)	The operation was successful.
DMNotAuth	The coprocessor application is not authorized to perform AES key wrapping operations (for example, because it has not called *xcAttachWithCDUOption*).
DMBadFlags	An input argument was invalid (for example, *pAESKWrap* is NULL or *pAESKWrap->hashOption* specifies more than one algorithm or a buffer is too small).
DMBadX9102KUW	Unwrap operation did not produce a valid X9.102 formatted buffer.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.4.7 xcAESKeyWrapNIST

xcAESKeyWrapNIST wraps an AES key according to the NIST standard (see “NIST formatted buffer” on page 48 for details):

Function prototype

```
int xcAESKeyWrapNIST( xcAESKW_NIST_t *pAESKWrap );
```

Input

On entry to this routine:

pAESKWrap contains the address of an AES key wrap request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *pWrapKey* points to a buffer containing the wrapping key (an item of type *xcAES_key_t*), i.e., the key to use to wrap the AES key.that is the object of the operation.
- *wrapKeyLen* specifies the length in **bits** of the wrapping key. *wrapKeyLen* must be 128, 192, or 256.
- *pKeyData* points to a buffer containing the AES key to be wrapped.
- *keyDataLen* is the length in **bits** of the key referenced by *pKeyData*. *keyDataLen* must be a multiple of 64.
- *pWrapData* points to a writeable buffer large enough to hold the wrapped key. See “NIST formatted buffer” on page 48 for details.
- *pWrapDataLen* points to a writeable buffer that contains the length in bytes of the buffer referenced by *pWrapData*.

Output

On successful exit from this routine:

The following fields of **pAESKWrap* are changed as noted:

- **pWrapDataLen* contains the actual length in bytes of the wrapped key.
- The buffer defined by *pWrapData* contains the wrapped key.

Return codes

Common return codes generated by this routine are:

DMGood (that is, 0)	The operation was successful.
DMNotAuth	The coprocessor application is not authorized to perform DES operations (for example, because it has not called *xcAttachWithCDUOption*).
DMBadFlags	An input argument was invalid (for example, *pAESKWrap* is

	NULL or a buffer has an invalid length).

Refer to *xc_err.h* for a comprehensive list of return codes.

3.4.8 xcAESKeyUnwrapNIST

xcAESKeyUnwrapNIST unwraps an AES key wrapped according to the NIST standard (see “NIST formatted buffer” on page 48 for details):

Function prototype

```
int xcAESKeyUnwrapNIST( xcAESKUW_NIST_t *pAESKUWrap );
```

Input

On entry to this routine:

pAESKUWrap contains the address of an AES key unwrap request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *pUnwrapKey* points to a buffer containing the unwrapping key (an item of type *xcAES_key_t*), i.e., the key to use to unwrap the wrapped AES key.that is the object of the operation.
- *unwrapKeyLen* specifies the length in **bits** of the wrapping key. *unwrapKeyLen* must be 128, 192, or 256.
- *pWrapData* points to a buffer containing the wrapped AES key.
- *wrapDataLen* is the length in bytes of the buffer referenced by *pWrapData*. *wrapDataLen* must be a multiple of 64.
- *pClearKey* points to a writeable buffer large enough to hold the unwrapped key.
- *pClearKeyLen* points to a writeable buffer that contains the length in **bits** of the buffer referenced by *pClearKey*.

Output

On successful exit from this routine:

The following fields of **pAESKUWrap* are changed as noted:

- **pClearKeyLen* contains the actual length in **bits** of the unwrapped key.
- The buffer defined by *pClearKey* contains the unwrapped key.

Notes

The NIST formatted buffer produced by the unwrapping operation is verified using the NIST integrity check method:

- Each of the first eight bytes of the buffer must have the value 0xA6.

If any of these conditions is not satisfied, the function returns **DMBadNISTKUW**.

Return codes

Common return codes generated by this routine are:

DMGood (that is, 0)	The operation was successful.
DMNotAuth	The coprocessor application is not authorized to perform DES operations (for example, because it has not called *xcAttachWithCDUOption*).
DMBadFlags	An input argument was invalid (for example, *pAESKWrap* is NULL or a buffer has an invalid length).
DMBadNISTKUW	Unwrap operation did not produce a valid NIST formatted buffer.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.5 DES Operation Functions

The functions described in this section allow a coprocessor application to ask the Symmetric Key Cipher Hash (SKCH) driver to perform various encryption, decryption, and MAC generation operations using the Data Encryption Standard (DES) algorithm.

A coprocessor application must call *xcAttachWithCDUOption* before calling any of the functions in this section.

3.5.1 xcTDES - triple DES encryption/decryption/MAC

xcTDES enables an application to encrypt, decrypt, or MAC data using a triple-length DES key.

xcTDES enciphers and deciphers an arbitrary amount of data using the DES (Data Encryption Standard) algorithm. Data can be enciphered in either Cipher Block Chaining (CBC) mode or Electronic Code Book (ECB) mode. *xcTDES* can also generate a message authentication code (MAC). Three separate DES keys are used.

Function prototype

```
unsigned int xcTDES( xcTDES_RB_t *pTDES_rb );
```

Input

On entry to this routine:

pTDES_rb contains the address of a TDES request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *options* controls the operation of the function and must be set to the logical OR of constants from the following categories:

Operation

options must include exactly one of the following constants:

DES_ENCRYPT	Encrypt the input.
DES_DECRYPT	Decrypt the input.
DES_MAC	Generate a message authentication code (MAC) for the input.

If **DES_MAC** is specified, **DES_ECB_MODE** must not be specified. **DES_CBC_MODE** will be assumed.

Chaining mode

options must include exactly one of the following constants:

DES_CBC_MODE	Use outer Cipher Block Chaining (CBC) mode.
DES_ECB_MODE	Use Electronic Code Book (ECB) mode.

DES_ECB_MODE must not be specified if **DES_MAC** is specified. **DES_CBC_MODE** will be assumed.

- *key1*, *key2*, and *key3* point to buffers containing the keys to use for the operation (each an item of type *xcDES_key_t*). If *options* specifies **DES_ENCRYPT** or **DES_MAC**, the input is encrypted with the contents of *key1*, the result is decrypted with the contents of *key2*, and that result is encrypted

with *key3*. If *options* specifies **DES_DECRYPT**, the input is decrypted with the contents of *key3*, the result is encrypted with the contents of *key2*, and that result is decrypted with the contents of *key1*.

- *init_v* points to a buffer containing the initial vector for the operation (an item of type *xcDES_vector_t*) if options specifies **DES_CBC_MODE** and is unused otherwise.
- *term_v* points to a buffer in which an item of type *xcDES_vector_t* can be stored.
- *source_length* is the length in bytes of the data to be processed by *xcTDES*. *source_length* must be a multiple of 8. If *source_length* is 0, at least one of *prepad_length* and *postpad_length* must be non-zero.
- *pSource* points to a buffer containing the data to be processed by *xcTDES*. The “input data” to *xcTDES* consists of the contents of this buffer plus any pre- or post-padding specified by options.
- *destination_length* is the length in bytes of the buffer referenced by *pDestination*. *destination_length* must be at least as large as the length of the input data.
- *pDestination* points to a writeable buffer.
- *prepad_length* is the length in bytes of the data in the buffer pointed to by *prepad.* This must be 0 or 8.
- *prepad* is a pointer to a buffer to be processed before the data in *pSource.*

  If *prepad_length* is zero, this may be a null pointer.
- *postpad_length* is the length in bytes of the data in *postpad.* This should be 0, 8, or 16.
- *postpad* is a buffer containing data which is to be processed after the data in *pSource.*

  If *postpad_length* is zero, *postpad* may be a null pointer.

Output

On successful exit from this routine:

The following fields of **pTDES_rb* are changed as noted:

- *term_v* contains
  - the message authentication code (MAC) generated from the input data if options specifies **DES_MAC**[3]**.**
  - the initialization vector for the next TDES operation if options specifies **DES_CBC_MODE**.

  term_v is undefined otherwise.
- The buffer referenced by *pDestination* contains
  - The input data processed using *key1*, *key2*, and *key3* (and *init_v* if *options* specifies **DES_CBC_MODE**) if *options* specifies **DES_ENCRYPT** or **DES_DECRYPT**.

  The buffer referenced by *pDestination* is undefined otherwise.

Notes

The length of the input data may be less than *destination_length*. In this case, any excess bytes at the end of the output buffer are not affected by *xcTDES*.

3 If *options* specifies both **DES_MAC** and **DES_CBC_MODE**, the MAC and the initialization vector are the same value.

If *pSource* and/or *pDestination* point to a DMA-eligible buffer the DES operation may complete more quickly than would otherwise be the case. See “Mapped kernel buffers and DMA-eligible buffers” on page 26 for details.

The buffers defined by *pSource/source_length* and *pDestination/destination_length* should not overlap.

Return codes

Common return codes generated by this routine are:

DMGood (that is, 0)	The operation was successful.
DMNotAuth	The coprocessor application is not authorized to perform DES operations (e.g., because it has not called *xcAttachWithCDUOption*).
DMBadFlags	The options argument is not valid.
DMBadParm	The length of the input data is invalid (e.g., not a multiple of 8), the length of the input data exceeds the length of the output buffer, or *source_length* is zero and no padding option was specified.
DMBadAddr	Part of the buffer defined by *pSource* and *source_length* is not readable, part of the buffer defined by *pDestination* and *destination_length* is not writeable, or a field in the request block cannot be accessed.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.5.2 xcDES - DES encryption/decryption/MAC

xcDES enables an application to encrypt, decrypt, or MAC data using a single-length DES key.

xcDES enciphers and deciphers an arbitrary amount of data using the DES (Data Encryption Standard) algorithm. Data can be enciphered in either Cipher Block Chaining (CBC) mode or Electronic Code Book (ECB) mode. *xcDES* can also generate a message authentication code (MAC).

Function prototype

```
unsigned int xcDES( xcDES_RB_t *pTDES_rb );
```

Input

On entry to this routine:

pDES_rb contains the address of a DES request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *options* controls the operation of the function and must be set to the logical OR of constants from the following categories:

Operation

options must include exactly one of the following constants:

DES_ENCRYPT	Encrypt the input.
DES_DECRYPT	Decrypt the input.
DES_MAC	Generate a message authentication code (MAC) for the input.

If **DES_MAC** is specified, **DES_ECB_MODE** must not be specified. **DES_CBC_MODE** will be assumed.

Chaining mode

options must include exactly one of the following constants:

DES_CBC_MODE	Use outer Cipher Block Chaining (CBC) mode.
DES_ECB_MODE	Use Electronic Code Book (ECB) mode.

DES_ECB_MODE must not be specified if **DES_MAC** is specified. **DES_CBC_MODE** will be assumed.

- *key* points to a buffer containing the key to use for the operation (an item of type *xcDES_key_t*).
- *init_v* points to a buffer containing the initial vector for the operation (an item of type *xcDES_vector_t*) if *options* specifies **DES_CBC_MODE** and is unused otherwise.
- *term_v* points to a buffer in which an item of type *xcDES_vector_t* can be stored.

- *source_length* is the length in bytes of the data to be processed by *xcTDES*. *source_length* must be a multiple of 8. If *source_length* is 0, at least one of *prepad_length* and *postpad_length* must be non-zero.
- *pSource* points to a buffer containing the data to be processed by *xcDES*. The “input data” to *xcDES* consists of the contents of the *prepad* buffer followed by the contents of this buffer followed by the contents of the *postpad* buffer.
- *destination_length* is the length in bytes of the buffer referenced by *pDestination*. *destination_length* must be at least as large as the length of the input data.
- *pDestination* points to a writeable buffer.
- *prepad_length* is the length of data in the *prepad* buffer. This should be 0 or 8.
- *prepad* is a pointer to a buffer of data which is to be processed before the data in *pSource.*

  If *prepad_length i*s zero, this may be a null pointer.
- *postpad_length* is the length of data in the *postpad* buffer. This should be 0, 8 or 16.
- *postpad* is a pointer to a buffer of data which is to be processed after the data in *pSource.*

  If *postpad_length* is zero, this may be a null pointer.

Output

On successful exit from this routine:

The following fields of **pDES_rb* are changed as noted:

- *term_v* contains
  - the message authentication code (MAC) generated from the input data if *options* specifies **DES_MAC.**
  - the initialization vector for the next DES operation if *options* specifies **DES_CBC_MODE**[4] .

  term_v is undefined otherwise.
- The buffer referenced by *pDestination* contains
  - The input data encrypted using *key* (and *init_v* if *options* specifies **DES_CBC_MODE**) if *options* specifies **DES_ENCRYPT**.
  - The input data decrypted using *key* (and *init_v* if *options* specifies **DES_CBC_MODE**) if options specifies **DES_DECRYPT**.

  The buffer referenced by *pDestination* is undefined otherwise.

Notes

The length of the input data may be less than *destination_length*. In this case, any excess bytes at the end of the output buffer are not affected by *xcDES*.

If *pSource* and/or *pDestination* point to a DMA-eligible buffer the DES operation may complete more quickly than would otherwise be the case.

The buffers defined by *pSource/source_length* and *pDestination/destination_length* should not overlap.

4 If *options* specifies both **DES_MAC** and **DES_CBC_MODE**, the MAC and the initialization vector are the same value.

Return codes

Common return codes generated by this routine are:

DMGood (that is, 0)	The operation was successful.
DMNotAuth	The coprocessor application is not authorized to perform DES operations (e.g., because it has not called *xcAttachWithCDUOption*).
DMBadFlags	The options argument is not valid.
DMBadParm	The length of the input data is invalid (e.g., not a multiple of 8), the length of the input data exceeds the length of the output buffer, or *source_length* is zero and no padding option was specified.
DMBadAddr	Part of the buffer defined by *pSource* and *source_length* is not readable, part of the buffer defined by *pDestination* and *destination_length* is not writeable, or a field in the request block cannot be accessed.

Refer to xc_err.h for a comprehensive list of return codes.

3.5.3 xcDES3Key - eight-byte triple DES

xcDES3Key enciphers or deciphers eight bytes of data using the DES (Data Encryption Standard) algorithm in Electronic Code Book (ECB) mode and a triple-length DES key.

The input data is assumed to be a single-length DES key, hence the function's name.

Function prototype

```
unsigned int xcDES3Key( xcDES3Key_RB_t *pDES3Key_rb );
```

Input

On entry to this routine:

pDES3Key_rb contains the address of a DES request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *options* controls the operation of the function and must be set to **DES_ENCRYPT** (encrypt the input) or **DES_DECRYPT** (decrypt the input).
- *key1*, *key2*, and *key3* contain the keys to use for the operation. If *options* specifies **DES_ENCRYPT**, the input is encrypted with the contents of *key1*, the result is decrypted with the contents of *key2*, and that result is encrypted with *key3*. If *options* specifies **DES_DECRYPT**, the input is decrypted with the contents of *key3*, the result is encrypted with the contents of *key2*, and that result is decrypted with the contents of *key1*.
- *key_in* contains the data to be processed by *xcDES3Key*.

Output

On successful exit from this routine:

The following fields of **pDES3Key_rb* are changed as noted:

key_out contains

- *key_in* processed using *key1*, *key2*, and *key3* according to the *options*.

Return codes

Common return codes generated by this routine are:

DMGood (that is, 0)	The operation was successful.
DMNotAuth	The coprocessor application is not authorized to perform DES operations (e.g., because it has not called *xcAttachWithCDUOption*).
DMBadFlags	The options argument is not valid.
DMBadAddr	A field in the request block cannot be accessed.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.5.4 xcEDE3_3DES – Legacy encryption mode

xcEDE3_3DES enciphers or deciphers data using the DES (Data Encryption Standard) algorithm in EDE3 mode (which differs from standard TDES mode) and a triple-length DES key.

Function prototype

```
unsigned int xcEDE3DES( int                fd,
                        xcEDE3DES3_RB_t *pEDE3DES_rb );
```

Input

On entry to this routine:

pEDE3DES_rb contains the address of a DES request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *options* controls the operation of the function and must be set to **DES_ENCRYPT** (encrypt the input) or **DES_DECRYPT** (decrypt the input).
- *key1*, *key2*, and *key3* contain the keys to use for the operation. If *options* specifies **DES_ENCRYPT**, the first block of input is encrypted with the contents of *key1*, the result is decrypted with the contents of *key2*, and that result is encrypted with *key3*. This resulting block is then encrypted with the 2nd block of input data and the process is repeated. If *options* specifies **DES_DECRYPT**, the first block of the input is decrypted with the contents of *key3*, the result is encrypted with the contents of *key2*, and that result is decrypted with the contents of *key1*. The decrypted block is then XORed with the second block of input data and the process is continued.
- *count* is the number of bytes of data in the *input* and *output* buffers.
- *input* contains the data to be processed by *xcEDE3DES*.
- *output* is a pointer to a writeable buffer.

Output

On successful exit from this routine:

The following fields of **pDES3Key_rb* are changed as noted:

output contains

- *input* processed using *key1*, *key2*, and *key3* if *options* specifies **DES_ENCRYPT**.

Return codes

Common return codes generated by this routine are:

DMGood (that is, 0)	The operation was successful.
DMNotAuth	The coprocessor application is not authorized to perform DES operations (e.g., because it has not called *xcAttachWithCDUOption*).
DMBadFlags	The options argument is not valid.

DMBadAddr	A field in the request block cannot be accessed.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.6 CMAC Operation Functions

The functions described in this section allow a coprocessor application to as the Symmetric Key Cryptographic Hash (SKCH) driver to perform either the AES or DES algorithms and create or verify a Message Authentication Code or to wrap a key using TR-31.

3.6.1 cmacGenerateNIST – Generate a Cipher-based Message Authentication Code according to the NIST specification

cmacGenerateNIST computes the C Message Authentication Code of a block of data using the NIST Cipher-based Message Authentication Code (as specified in NIST SP 800-38B).

Function prototype

```
unsigned int cmacGenerateNIST( cmacGen_NIST_t *pCmacGen );
```

Input

On entry to this routine:

pCmacGen contains the address of a cmacGen_NIST_t request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *options* controls the operation of the function and must be set to the logical OR of constants from the following categories:

Operating mode

options must include exactly one of the following constants:

CMAC_MSGPART_ONLY	This is the complete message to be MACed.
CMAC_MSGPART_FIRST	This is the first part of the message to be MACed; more parts will follow.
CMAC_MSGPART_MIDDLE	This a a second (or later) part of the message. The results of prior calls to cmacGenNIST are in the *pIV* parameter.
CMAC_MSGPART_FINAL	This is the final call to generate a MAC. The results of prior calls to cmacGenNIST are in the *pIV* parameter, and the final result should be placed in *pOutMAC.*

Key type and size

options must include exactly one of the following constants:

DES_EDE2	Calculate a CMAC using a double length TDEA key.
DES_EDE3	Calculate a CMAC using a triple-length TDEA key.
AES_128BIT_KEY	Calculate a CMAC using a 128-bit AES key.
AES_192BIT_KEY	Calculate a CMAC using a 192-bit AES key.
AES_256BIT_KEY	Calculate a CMAC using a 256-bit AES key.

- *keyLen* contains the length of the key in bytes.
- *pKey* is a pointer to the raw key, from which the subkeys key1 and key2 are derived.
- *inMsgLen* contains the length in bits of the input data.

  If *options* specifies **CMAC_MSGPART_FIRST** or **CMAC_MSGPART_MIDDLE**, *inMsgLen* must be a multiple of the block size of the underlying algorithm. (AES 16 bytes or DES 8 bytes).
- *pInMsg* points to a buffer containing the input data.
- *IVLen* is the length of the initialization vector in bytes. This must be the block size for this algorithm (TDES or AES).
- *pIV* is a pointer to the initialization vector for this call. If *options* includes **CMAC_MSGPART_FIRST** or **CMAC_MSGPART_ONLY,** this buffer must be filled with 0x00 bytes. Otherwise, this buffer should contain the output (*pOutMAC*) from the previous call to the function.
- *pOutMACLen* is a pointer to a buffer containing the number of bytes in the buffer pointed to by *pOutMAC.* If *options* includes **DES_EDE2** or **DES_EDE3,** **pOutMACLen* must be at least 8 bytes long. Otherwise, **pOutMACLen* must be at least 16 bytes long.
- *pOutMAC* is a pointer to a buffer at least **pOutMACLen* bytes long.

Output

On successful exit from this routine:

The following fields of **pCmacGen* are changed as noted:

- *pOutMAC* contains the MAC of the input data, using the key specified in *pKey.*

  If, when *cmacGenNIST* was called, *options* specified **CMAC_MSGPART_MIDDLE** or **CMAC_MSGPART_FINAL**, *pOutMAC* also incorporates the value of *pIV* on entry to the routine.
- *pOutMACLen* is the number of bytes of data in *pOutMAC*.

Notes

If *pSource* points to a DMA-eligible buffer the hash operation may complete more quickly than would otherwise be the case. See “Mapped kernel buffers and DMA-eligible buffers” on page 26 for details.

Return codes

Common return codes generated by this routine are:

DMGood (that is, 0)	The operation was successful.
DMNotAuth	The coprocessor application is not authorized to perform AES or DES operations (for example, because it has not called *xcAttachWithCDUOption*).
DMBadFlags	The options argument is not valid.
DMBadParm	A parameter was invalid (e.g, *options* specifies **CMAC_MSGPART_FIRST** or **CMAC_MSGPART_ONLY** but *pIV* contains non-zero data).

DMBadAddr	Part of the buffer defined by *pInMsg* and *inMsgLen* is not readable (i.e., is not mapped).

3.6.2 cmacVerifyNIST – Verify a Cipher-Based Message Authentication Code

cmacVerifyNIST computes the C Message Authentication Code of a block of data using the Cipher-based Message Authentication Code (as specified in NIST SP 800-38B).

Function prototype

```
unsigned int cmacVerifyNIST( cmacVer_NIST_t *pCmacVer );
```

Input

On entry to this routine:

pCmacVer contains the address of a cmacVer_NIST_t request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *options* controls the operation of the function and must be set to the logical OR of constants from the following categories:

Operating mode

options must include exactly one of the following constants:

CMAC_MSGPART_ONLY	This is the complete message to be MACed.
CMAC_MSGPART_FIRST	This is the first part of the message to be MACed; more parts will follow.
CMAC_MSGPART_MIDDLE	This a a second (or later) part of the message. The results of prior calls to cmacGenNIST are in the *pIV* parameter.
CMAC_MSGPART_FINAL	This is the final call to generate a MAC. The results of prior calls to cmacGenNIST are in the *pIV* parameter, and the final result should be placed in *pOutMAC.*

Key type and size

options must include exactly one of the following constants:

DES_EDE2	Calculate a CMAC using a double length TDEA key.
DES_EDE3	Calculate a CMAC using a triple-length TDEA key.
AES_128BIT_KEY	Calculate a CMAC using a 128-bit AES key.
AES_192BIT_KEY	Calculate a CMAC using a 192-bit AES key.
AES_256BIT_KEY	Calculate a CMAC using a 256-bit AES key.

- *keyLen* contains the length of the key in bytes.
- *pKey* is a pointer to the raw key, from which the subkeys key1 and key2 are derived.
- *inMsgLen* contains the length in bits of the input data.

  If *options* specifies **CMAC_MSGPART_FIRST** or **CMAC_MSGPART_MIDDLE**, *inMsgLen* must be a multiple of the block size of the underlying algorithm. (AES 16 bytes or DES 8 bytes).
- *pInMsg* points to a buffer containing the input data.
- *IVLen* is the length of the initialization vector in bytes. This must be the block size for this algorithm (TDES or AES).
- *pIV* is a pointer to the initialization vector for this call. If *options* includes **CMAC_MSGPART_FIRST** or **CMAC_MSGPART_ONLY,** this buffer must be filled with 0x00 bytes. Otherwise, this buffer should contain the output (*pOutMAC*) from the previous call to the function.
- *pOutMACLen* is a pointer to a buffer containing the number of bytes in the buffer pointed to by *pOutMAC.* If *options* includes **DES_EDE2** or **DES_EDE3,** **pOutMACLen* must be at least 8 bytes long. Otherwise, **pOutMACLen* must be at least 16 bytes long.
- *pOutMAC* is a pointer to a buffer at least **pOutMACLen* bytes long.

Output

On successful exit from this routine:

The following fields of **pCmacGen* are changed as noted:

- *pOutMAC* contains the MAC of the input data, using the key specified in *pKey.*

  If, when *cmacGenNIST* was called, *options* specified **CMAC_MSGPART_MIDDLE** or **CMAC_MSGPART_FINAL**, *pOutMAC* also incorporates the value of *pIV* on entry to the routine.
- *pOutMACLen* is the number of bytes of data in *pOutMAC*.

Notes

If *pInMsg* points to a DMA-eligible buffer the hash operation may complete more quickly than would otherwise be the case. See “Mapped kernel buffers and DMA-eligible buffers” on page 26 for details.

Return codes

Common return codes generated by this routine are:

DMGood (that is, 0)	The operation was successful.
DMNotAuth	The coprocessor application is not authorized to perform AES or DES operations (for example, because it has not called *xcAttachWithCDUOption*).
DMBadFlags	The options argument is not valid.
DMBadParm	A parameter was invalid (e.g, *options* specifies **CMAC_MSGPART_FIRST** or **CMAC_MSGPART_ONLY** but *pIV* contains non-zero data).

DMBadAddr	Part of the buffer defined by *pInMsg* and *inMsgLen* is not readable (i.e., is not mapped).

3.7 Format Preserving Encryption Operation Functions

The function described in this section allows a coprocessor application to ask the Symmetric Key Cryptographic Hash (SKCH) driver to preform various operations to encrypt a string translated into a particular alphabet into another string in the same alphabet.

A coprocessor application must call *xcAttachWithCDUOption* before calling any of the functions in this section.

3.7.1 Format Preserving Encryption alphabets

The format preserving encryption standard calls for translating the input data into a string of digits, each digit representing one character in the allowed alphabet. This string is then encrypted using a symmetric key (TDES or AES) and the resulting encrypted block is then read as a string of digits *modulo* the number of digits in the alphabet.

Thus, prior to calling xcVfpe, you must translate the provided string (whether it is in ASCII, EBCDIC, or BCD) into a string of digits representing your accepted alphabet. For example, “ABC” might be translated to 0x000102. The output of the xcVfpe function would also need to be translated back to your alphabet. For example, you might be returned a string 0x070105, and would perhaps translate it back to “GAE”.

Correct implementation of a VFPE scheme thus requires the definition of an alphabet (acceptable characters), a translation scheme from the alphabet to the string of digits, a maximum value for the keystream (Bmax) and the number of digits that may be encrypted by the keystream (k).

3.7.2 xcVfpe – Perform VISA Format Preserving Encryption

xcVfpe generates the encrypted (or decrypted) string of digits representing an alphabet in a Format Preserving Encryption scheme, according to the specification *Visa Merchant Data Secure with Point to Point Encryption (VMDS with P2PE) Hardware Security Module Guide Version 2.0.*

Function prototype

```
unsigned int xcVfpe( xcVFpe_RB_t *pxcVfpe_rb );
```

Input

On entry to this routine:

xcVfpe is a pointer to a VFPE request block whose fields have been initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *n_numCharInAlphabet* is the number of characters in the alphabet that is being translated.
- *b_cipherBlockSize* is the cipher block size of the algorithm to use. If *options* includes **VFPE_CIPHER_DES**, *b_cipherBlockSize* must be 8. If *options* includes **VFPE_CIPHER_AES**, *b_cipherBlockSize* must be 16.
- *options* must include exactly one of the following constants:

Operation

VFPE_ENCRYPT	Encrypt the input.
VFPE_DECRYPT	Decrypt the input.

Key type

- *options* must include exactly one of the following constants:

VFPE_CIPHER_DES	Use a TDES key
VFPE_CIPHER_AES	Use an AES key.

- *EncKeyBitLen* is the bit length of the key in *pEncKey.* If *options* includes **VFPE_CIPHER_DES,** this may be 128 or 192. If *options* includes **VFPE_CIPHER_AES**, this my be 128, 192, or 256.
- *pEncKey* is the clear raw key to use to encrypt or decrypt the data.
- *k_blkCharLen* is the number of characters in the alphabet which may be encrypted with a single block of encryption.
- *Bmax_len* is the number of characters in the string pointed to by *Bmax.*
- *Bmax* is the largest allowable value for keystream block B.
- *counterArrayLen* is the number of characters in the array pointed to by *pCounter.*

- *pCounter* is the counter used to track the use of blocks encrypted with this string. The counter is modified for each block of data that is encrypted with the key.
- *L_inTextLen* is the number of digits in the input text.
- *inText* is the string to be encrypted or decrypted. This should be a string of digits, as translated from the original alphabet/ascii/hex.
- *pL_outText* is a buffer containing the length of the *outText* buffer. This must be at least as large as the *L_inTextLen.*
- *outText* is a pointer to a buffer in which to store the output.

Output

On successful exit from this routine:

xcVfpe has been updated as follows:

- *pL_outText* has been updated with the actual number of digits encrypted or decrypted.
- *outText* contains the decrypted digit string.

Return codes

Common return codes generated by this routine are:

DMGood (that is, 0)	The operation was successful.
DMNotAuth	The coprocessor application is not authorized to perform AES or DES operations (for example, because it has not called *xcAttachWithCDUOption*).
DMBadFlags	The options argument is not valid.
DMBadParm	A parameter was invalid (e.g, *options* specifies **VFPE_CIPHER_DES** but *b_cipherBlockSize* is not 8).
DMBadAddr	Part of the buffer defined by i*nText* and *L_inTextLen* is not readable (i.e., is not mapped).

3.8 RSA Operation Functions

The functions described in this section allow a coprocessor application to ask the Public Key Algorithm (PKA) driver to perform various cryptographic operations using the Rivest-Shamir-Adleman (RSA) algorithm or the Elliptic Curve Cryptography (ECC) algorithm.

A coprocessor application must call *xcAttachWithCDUOption* before calling any of the functions in this section.

3.8.1 RSA key tokens

The PKA interface defines the *xcRsaKeyToken_t* type to hold information about RSA public and private keys. The interface also defines the *xcPKCSKeyToken_t* type to hold information about RSA private keys stored in PKCS#1 CRT form. An item of either type consists of a descriptive header followed by a buffer containing information about the values of the various elements of the key. For example, the key token for an RSA public key includes the modulus n and the public exponent e. The header indicates which elements are present and gives the length of and a pointer to each element. Elements are stored in big-endian order: the byte at the lowest address contains the most significant byte of the element.

The fields of the key token for an RSA public key are set as follows:

- *type* is **RSA_PUBLIC_MODULUS_EXPONENT**.
- *tokenLength* is the length in bytes of the *key* token.
- *n_BitLength* is the length in bits of the modulus *n*.
- *n_Length* is the length in bytes of the modulus *n*.
- *n_Ptr* points to the first (most significant) byte of the modulus *n*.
- *e_Length* is the length in bytes of the public exponent *e*.
- *e_Ptr* points to the first (most significant) byte of the public exponent *e*.

The remaining length and pointer fields are ignored and should be set to zero.

The PKA interface supports six kinds of key tokens for an RSA private key:

- **RSA_PRIVATE_MODULUS_EXPONENT**
- **RSA_X931_PRIVATE_MODULUS_EXPONENT**
- **RSA_PRIVATE_CHINESE_REMAINDER**
- **RSA_X931_PRIVATE_CHINESE_REMAINDER**
- **RSA_PKCS_PRIVATE_CHINESE_REMAINDER**
- **RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER**

The PKA interface uses a straightforward modular exponentiation approach to decrypt ciphertext or wrap an X9.31 encapsulated hash, as appropriate, using a xcRsaKeyToken_t key token whose fields are set as follows:

- *type* is **RSA_PRIVATE_MODULUS_EXPONENT** (decrypt ciphertext) or **RSA_X931_PRIVATE_MODULUS_EXPONENT** (wrap encapsulated hash).
- *tokenLength* is the length in bytes of the key token.

- *n_BitLength* is the length in bits of the modulus n. *n_BitLength* cannot exceed 4096

  If *type* is **RSA_X931_PRIVATE_MODULUS_EXPONENT**, *n_BitLength* must be greater than or equal to1024 and a multiple of 256
- *n_Length* is the length in bytes of the modulus *n*.
- *n_Ptr* points to the first (most significant) byte of the modulus *n*.
- *e_Length* is the length in bytes of the public exponent *e*.
- *e_Ptr* points to the first (most significant) byte of the public exponent i.
- *x.d_Length* is the length in bytes of the private exponent *d* [5].
- *y.d_Ptr* points to the first (most significant) byte of the private exponent *d*.
- *r_Length* is the length in bytes of the blinding value *r* [6].
- *r_Ptr* points to the first (most significant) byte of the blinding value *r*.
- *r1Length* is the length in bytes of the blinding value r^{-1}, which is the inverse of *r* modulo *n*.
- *r1Ptr* points to the first (most significant) byte of the blinding value r^{1}.

The remaining length and pointer fields are not used and should be set to zero.

The PKA interface uses an approach based on the Chinese Remainder Theorem to decrypt ciphertext or wrap an X9.31 encapsulated hash, as appropriate, using a *xcRSAKeyToken_t* key token whose fields are set as follows:

- *type* is **RSA_PRIVATE_CHINESE_REMAINDER** (decrypt ciphertext) or **RSA_X931_PRIVATE_CHINESE_REMAINDER** (wrap encapsulated hash).
- *tokenLength* is the length in bytes of the key token.
- *n_BitLength* is the length in bits of the modulus *n*. *n_BitLength* cannot exceed 4096.

  If *type* is **RSA_X931_PRIVATE_CHINESE_REMAINDER**, *n_BitLength* must be 1024+256k for nonnegative integer k.
- *n_Length* is the length in bytes of the modulus *n*.
- *n_Ptr* points to the first (most significant) byte of the modulus *n*.
- *e_Length* is the length in bytes of the public exponent *e*.
- *e_Ptr* points to the first (most significant) byte of the public exponent *e*.
- *x.p_Length* is the length in bytes of the prime number *p* [7].
- *y.p_Ptr* points to the first (most significant) byte of the prime number *p*. The value of p must be greater than the value of *q*.
- *q_Length* is the length in bytes of the prime number *q*.
- *q_Ptr* points to the first (most significant) byte of the prime number *q*. The value of *q* must be less than the value of *p*.

---

5 *d* is the inverse of the public exponent *e* modulo *(p-1)(q-1)*

6 r = R^e mod *n* and r^{-1} is the inverse of *r* modulo *n* where *R* is a random number less than the modulus *n*.

7 *n = pq*

- *dp_Length* is the length in bytes of *dp = d* mod *(p-1)*, where *d* is the private exponent.
- *dpPtr* points to the first (most significant) byte of *dp*.
- *dqLength* is the length in bytes of *dq = d* mod *(q-1)*, where *d* is the private exponent.
- *dqPtr* points to the first (most significant) byte of *dq*.
- *apLength* is the length in bytes of $ap = q^{p-1} \bmod n$.
- *apPtr* points to the first (most significant) byte of *ap*.
- *aqLength* is the length in bytes of *aq = n + 1 - ap*.
- *aqPtr* points to the first (most significant) byte of *aq*.
- *r_Length* is the length in bytes of the blinding value *r* [8].
- *r_Ptr* points to the first (most significant) byte of the blinding value *r*.
- *r1Length* is the length in bytes of the blinding value r^{-1}, which is the inverse of r modulo n.
- *r1Ptr* points to the first (most significant) byte of the blinding value r^{1}.

The PKA Driver also uses a (different) approach based on the Chinese Remainder Theorem to decrypt ciphertext or wrap an X9.31 encapsulated hash, as appropriate, using a *xcPKCSKeyToken_t* key token whose fields are set as follows:

- *type* is **RSA_PKCS_PRIVATE_CHINESE_REMAINDER** (decrypt ciphertext) or **RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER** (wrap encapsulated hash).
- *tokenLength* is the length in bytes of the key token.
- *n_BitLength* is the length in bits of the modulus n. *n_BitLength* cannot exceed 4096.

  If *type* is **RSA_X931_PRIVATE_CHINESE_REMAINDER**, *n_BitLength* must be 1024+256k for nonnegative integer *k*.
- *n_Length* is the length in bytes of the modulus *n*.
- *n_Ptr* points to the first (most significant) byte of the modulus *n*.
- *e_Length* is the length in bytes of the public exponent *e*.
- *e_Ptr* points to the first (most significant) byte of the public exponent *e*.
- *x.p_Length* is the length in bytes of the prime number *p* [9].
- *y.p_Ptr* points to the first (most significant) byte of the prime number *p*. The value of p must be greater than the value of *q*.
- *q_Length* is the length in bytes of the prime number *q*.
- *q_Ptr* points to the first (most significant) byte of the prime number *q*. The value of *q* must be less than the value of *p*.
- *dp_Length* is the length in bytes of *dp = d* mod *(p-1)*, where *d* is the private exponent.
- *dpPtr* points to the first (most significant) byte of *dp*.

---

8 $r = R^e \bmod n$ and r^{-1} is the inverse of *r* modulo *n* where *R* is a random number less than the modulus *n*.

9 *n = pq*

- *dqLength* is the length in bytes of $dq = d \bmod (q-1)$, where *d* is the private exponent.
- *dqPtr* points to the first (most significant) byte of *dq*.
- *qInvLength* is the length in bytes of $q^{-1} \bmod p$.
- *qInvPtr* points to the first (most significant) byte of $q^{-1} \bmod p$.
- *notDefined1* and *notDefined2* are reserved and should be set to zero.
- *r_Length* is the length in bytes of the blinding value *r* [10].
- *r_Ptr* points to the first (most significant) byte of the blinding value *r*.
- *r1Length* is the length in bytes of the blinding value r^{-1}, which is the inverse of *r* modulo *n*.
- *r1Ptr* points to the first (most significant) byte of the blinding value r^{-1}.

Use of a private key of type **RSA_PRIVATE_CHINESE_REMAINDER**, **RSA_X931_PRIVATE_CHINESE_REMAINDER**, **RSA_PKCS_PRIVATE_CHINESE_REMAINDER**, or **RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER** can improve performance with no loss of security.

Note that an RSA private key token includes the public exponent. This portion need not be present when the token is used as a private key.

If *n_BitLength* is not a multiple of 8, any excess high-order bits in the modulus are treated as zeros (that is, n is essentially padded on the left with zeros, regardless of the actual bits that appear in the key token). If *n_BitLength* is 4096, *e_Length* should not be greater than 3.

3.8.2 xcRSAKeyGenerate - generate an RSA keypair

xcRSAKeyGenerate generates a key token for an RSA keypair, i.e., a token containing information about both the public and the private key.

Function prototype

```
unsigned int xcRSAKeyGenerate( xcRSAKeyGen_RB_t *pRSAKeyGen_rb );
```

Input

On entry to this routine:

pRSAKeyGen_rb contains the address of a RSA key generation request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *key_type* specifies which kind of private key token is generated and must be one of the following constants:

  RSA_PRIVATE_MODULUS_EXPONENT

  RSA_PRIVATE_CHINESE_REMAINDER

  RSA_X931_PRIVATE_MODULUS_EXPONENT

  RSA_X931_PRIVATE_CHINESE_REMAINDER

10 $r = R^e \bmod n$ and r^{-1} is the inverse of *r* modulo *n* where *R* is a random number less than the modulus *n*.

RSA_PKCS_PRIVATE_CHINESE_REMAINDER

RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER

- *mod_size* specifies the desired length in bits of the modulus *n*. *mod_size* must be less than or equal to 4096.

  If *key_type* is **RSA_X931_PRIVATE_MODULUS_EXPONENT**, **RSA_X931_PRIVATE_CHINESE_REMAINDER**, or **RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER**, *mod_size* must be 1024+256k for nonnegative integer *k*.

- *public_exp* determines how the value of the public exponent *e* is chosen and must be one of the following constants:

RSA_EXPONENT_RANDOM	Choose a pseudo-random number containing *mod_size* bits that meets the standards described in ANSI X9.31
RSA_EXPONENT_FIXED	Use the value of *e* in the RSA key token referenced by *key_token*
RSA_EXPONENT_2	Set *e* = 2
RSA_EXPONENT_3	Set *e* = 3
RSA_EXPONENT_65537	Set *e* = 65537 (0x10001)

  RSA_EXPONENT_3 and **RSA_EXPONENT_65537** provide support for certain standards that require specific public exponents (for example, secure electronic transactions). These are recommended values for 4096-bit keys.

  If *public_exp* is **RSA_EXPONENT_FIXED**, the public exponent must be odd unless *key_type* is **RSA_X931_PRIVATE_MODULUS_EXPONENT**, **RSA_X931_PRIVATE_CHINESE_REMAINDER**, or **RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER**.

  If *public_exp* is **RSA_EXPONENT_2**, *key_type* must be **RSA_X931_PRIVATE_MODULUS_EXPONENT**, **RSA_X931_PRIVATE_CHINESE_REMAINDER**, or **RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER**.

- *key_token* must contain the address of a writeable buffer in which an item of type *xcRSAKeyToken_t* can be stored.

  If *public_exp* is **RSA_EXPONENT_FIXED**, *key_token->tokenlength* must be valid and the buffer defined by *key_token->e_Ptr* and *key_token->e_Length* must contain the public exponent *e*.

  The fields of **key_token* must be initialized as shown below. Note that a particular field is only used if appropriate for a key whose type is *pRSAKeyGen_rb->key_type*. For example, *x.p_Length* is not used if *pRSAKeyGen_rb->key_type* is **RSA_PRIVATE_MODULUS_EXPONENT**.

  - *tokenLength* is the length in bytes of the buffer referenced by *key_token*.
  - *n_BitLength* is the desired length in bits of the modulus *n*.
  - *n_Length* is the length in bytes of the buffer referenced by *n_Ptr*. *n_Length* must be greater than or equal to ((*n_BitLength* + 7) & ~7) >> 3.

- *e_Length* is the length in bytes of the buffer referenced by *e_Ptr*. *e_Length* must be large enough to accommodate the public exponent.
- *x.p_Length* is the length in bytes of the buffer referenced by *y.p_Ptr*. *x.p_Length* must be greater than or equal to *n_Length*/2.
- *x.d_Length* is the length in bytes of the buffer referenced by *y.d_Ptr*. *x.d_Length* must be greater than or equal to *n_Length*.
- *q_Length* is the length in bytes of the buffer referenced by *q_Ptr*. *q_Length* must be greater than or equal to *n_Length*/2.
- *dpLength* is the length in bytes of the buffer referenced by *dpPtr*. *dpLength* must be greater than or equal to *x.p_Length*.
- *dqLength* is the length in bytes of the buffer referenced by *dqPtr*. *dqLength* must be greater than or equal to *q_Length*.
- *apLength* is the length in bytes of the buffer referenced by *apPtr*. If *pRSAKeyGen_rb->key_type* is not **RSA_PKCS_***, *apLength* must be greater than or equal to *n_Length*. If *pRSAKeyGen_rb->key_type* is **RSA_PKCS_***, *apLength* must be greater than or equal to *p_Length*. In the latter case, *key_token* actually points to a structure of type *xcPKCSKeyToken_t* and the field referenced by *apLength* is actually *qInvLength*.
- *aqLength* is the length in bytes of the buffer referenced by *aqPtr*. If *pRSAKeyGen_rb->key_type* is not **RSA_PKCS_***, *aqLength* must be greater than or equal to *n_Length*. If *pRSAKeyGen_rb->key_type* is **RSA_PKCS_***, *aqLength* is not used. In the latter case, *key_token* actually points to a structure of type *xcPKCSKeyToken_t* and the field referenced by *aqLength* is actually *notDefined1*.
- *r_Length* is the length in bytes of the buffer referenced by *r_Ptr*. *r_Length* must be greater than or equal to *n_Length*.
- *r1Length* is the length in bytes of the buffer referenced by *r1Ptr*. *r1Length* must be greater than or equal to *n_Length*.
- *n_Ptr*, *e_Ptr*, *y.p_Ptr* or *y.d_Ptr*, *q_Ptr*, *dpPtr*, *dqPtr*, *apPtr*, *aqPtr*, *r_ptr*, and *r1Ptr* must point to writeable buffers whose lengths are given by the corresponding **Length* fields (e.g., the length of the buffer referenced by *n_Ptr* is given by *n_Length*).

- *key_size* points to a writable buffer in which an item of type unsigned long can be stored. **key_size* must be the length in bytes of the buffer referenced by *key_token*.
- *regen_size* is the length in bytes of the buffer referenced by *regen_data*.

  If *regen_data* is NULL, *regen_size* must be zero.

  If the high-order bit of *regen_data* is set, *xcRSAKeyGenerate* performs 7 rounds of the Miller-Rabin primality test for each candidate 101g-bit prime (used to create *p* and *q*) and for each of *p* and *q*. This meets the current ANSI X9.31 requirements. If the high-order bit of *regen_data* is clear, the number of rounds is 8.

  (If *regen_data* is not used, *xcRSAKeyGenerate* performs 38 rounds of the Miller-Rabin primality test for each candidate 101-bit prime and 7 rounds for each of *p* and *q*.)
- *regen_data* may be NULL (and should be NULL when generating keys in the course of normal operations).

  If *regen_data* is not NULL, it points to a string of bits used to seed the PKA driver's pseudo-

random number generator, which is used to generate the prime numbers *p* and *q* (and the public exponent e if *public_exp* is **RSA_EXPONENT_RANDOM**). The bit string should contain at least 160 bits of entropy to ensure the keys generated from the seed are cryptographically sound. Use of *regen_data* ensures reproducible results and thus assists testing and benchmarking.

In normal operations (i.e., when *regen_data* is NULL), the PKA Driver obtains its random numbers from the RNG Driver. If *regen_data* is not NULL, the string it references should contain at least 160 bits of entropy to ensure the keys generated from the seed are cryptographically sound.

Output

On successful exit from this routine:

The following fields of **pRSAKeyGen_rb* are changed as noted:

- **key_token* contains a key token for an RSA private key. *pRSAKeyGen_rb->key_token->type* is set to the value of *pRSAKeyGen_rb->key_type*. The various buffers defined by the other fields of **key_token* are set as shown below. Note that a particular buffer is only used if appropriate for a key whose type is *key_token->type*. For example, the buffer defined by *y.p_Ptr/x.p_Length* is not used if *key_token->type* is **RSA_PRIVATE_MODULUS_EXPONENT**.
  - The buffer defined by *n_Ptr/n_Length* contains the modulus *n*.
  - The buffer defined by *e_Ptr/e_Length* contains the public exponent *e*.
  - The buffer defined by *y.p_Ptr/x.p_Length* contains the prime *p*.
  - The buffer defined by *y.d_Ptr/x.d_Length* contains the private exponent *d*.
  - The buffer defined by *q_Ptr/q_Length* contains the prime *q*.
  - The buffer defined by *dpPtr/dpLength* contains $dp = d \bmod (p-1)$.
  - The buffer defined by *dqPtr/dqLength* contains $dq = d \bmod (q-1)$.
  - The buffer defined by *apPtr/apLength* contains
    - $ap = q^{p-1} \bmod n$ if *key_token->type* is not **RSA_PKCS_*** or
    - $q^{-1} \bmod p$ if *pRSAKeyGen_rb->key_type* is **RSA_PKCS_***. (In this case, *key_token* actually points to a structure of type *xcPKCSKeyToken_t* and the buffer referenced by *apPtr/apLength* is actually the buffer referenced by *qInvPtr/qInvLength*.)
  - The buffer defined by *aqPtr/aqLength* contains $aq = n + 1 - ap$.
  - The buffer defined by r*_Ptr/r_Length* contains the blinding value *r,* if one was supplied on input. The coprocessor does not generate blinding values.
  - The buffer defined by *r1Ptr/r1Length* contains the blinding value r^{-1}, which is the inverse of *r* modulo *n*.
- *key_size* contains the length in bytes of the key token (i.e., *key_token->tokenLength*).

Notes

A key token for an RSA public key can be generated from the key token for the corresponding RSA private key by copying the buffers defined by *n_PTR/n_Length* and *e_PTR/e_Length*, copying the *n_Length*, *e_Length*, and *n_BitLength* fields, and setting the public key token's type field to to **RSA_PUBLIC_MODULUS_EXPONENT**.

None of the buffers defined to hold a piece of the generated key should overlap any of the buffers defined to hold a different piece of the generated key.

Return codes

Common return codes generated by this routine are:

PKAGood (that is, 0)	The operation was successful.
PKANotAuth	The coprocessor application is not authorized to perform PKA operations (e.g., because it has not called *xcAttachWithCDUOption*).
PKABadParm	An argument is not valid.
PKANoSpace	The operation failed due to lack of space (for example, the buffer referenced by *pRSAKeyGen_rb->key_token* is not large enough to hold the token generated by the call or there is no free memory available to the PKA driver).

Refer to *xc_err.h* for a comprehensive list of return codes.

3.8.3 xcRSA - encipher/decipher data or wrap/unwrap X9.31 encapsulated hash

xcRSA enciphers or deciphers a block of data using the RSA algorithm or wraps or unwraps an X9.31 encapsulated hash.

Function prototype

```
unsigned int xcRSA( xcRSA_RB_t *pRSA_rb );
```

Input

On entry to this routine:

pRSA_rb contains the address of a RSA request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *options* controls the operation of the function and must be set to the logical OR of constants from the following categories:

Public or private key

options must include exactly one of the following constants:

RSA_PUBLIC	Perform the operation using the public key from the key token (that is, *output* = $input^{e}$ mod *n*).
RSA_PRIVATE	Perform the operation using the private key from the key token (for example, *output* = $input^{d}$ mod *n*).

If **RSA_PRIVATE** is specified, **RSA_DECRYPT** must also be specified. If **RSA_PUBLIC** is specified, **RSA_ENCRYPT** must also be specified.

RSA_PRIVATE must be specified to wrap an X9.31 encapsulated hash. **RSA_PUBLIC** must be specified to unwrap an X9.31 encapsulated hash.

If **RSA_PRIVATE** is specified, *key_token->type* must not be **RSA_PUBLIC_MODULUS_EXPONENT**.

Blinding operation

Certain implementations of the RSA algorithm are vulnerable to a timing attack. The blinding values *r* and r^{-1} are used to defeat such attacks.

The implementation of the RSA algorithm in the PCIe Cryptographic Coprocessor is not subject to timing attacks, and the blinding values are included in the key token for compatibility with earlier implementations of the PKA interface.

options may include exactly one of the following constants:

RSA_DONT_BLIND	Perform the operation without using the blinding values.
RSA_BLIND_NO_UPDATE	Perform the operation using the blinding values and replace

	the blinding values in the key token.
RSA_BLIND_UPDATE	Perform the operation using the blinding values but do not replace the blinding values in the key token.

The presence or absence of these options has no effect on the function, they are simply included for compatibility with earlier implementations.

ANSI X9.31 operation

options must include **RSA_X931_OPERATION** if *pRSA_rb->key_token->key_type* is **RSA_X931_PRIVATE_MODULUS_EXPONENT**, **RSA_X931_PRIVATE_CHINESE_REMAINDER**, or **RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER**.

- *key_token* points to a buffer containing the RSA key token for the key to be used in the operation.
- *key_size is* the length in bytes of the RSA key token referenced by *key_token* (that is, *key_token->tokenLength*).
- *data_in* points to a buffer that contains the input data.

  If *options* specifies **RSA_X931_OPERATION**, the buffer is assumed to contain a valid X9.31 encapsulated hash. The encapsulated hash must be wrapped if *options* specifies **RSA_PUBLIC** and must not be wrapped if options specifies **RSA_PRIVATE**.
- *data_size* is the length in bits of input data and must be equal to the modulus bit length (i.e., *key_token->n_BitLength*).
- *data_out* points to a writeable buffer.
- *output_size* is the length in bytes of the buffer referenced by *data_out*.

Output

On successful exit from this routine:

The buffer defined by *pRSA_rb->data_out/pRSA_rb->output_size* contains:

- The input data transformed using the public key from *pRSA_rb->key_token* if *pRSA_rb->options* specifies **RSA_PUBLIC**.
- The input data transformed using the private key from *pRSA_rb->key_token* if *pRSA_rb->options* specifies **RSA_PRIVATE**.

pRSA_rb->output_size is the length in bytes of the transformed data.

The blinding values *r* and r^{-1} in **(pRSA_rb->key_token)* are replaced if *pRSA_rb->options* specifies **RSA_BLIND_NO_UPDATE**.

Notes

Buffer overlap

The buffers defined by *data_in/data_size* and *data_out/output_size* should not overlap.

Buffer length not equal to modulus length

If the length of the input data or the output buffer is less than the length of the modulus n (that is, if

pRSA_rb->data_size < *pRSA_rb->key_token->n_BitLength* or if

pRSA_rb->output_size < *pRSA_rb->key_token->n_Length)*,

xcRSA returns **PKABadParm**.

If the length of the input data or the output buffer is greater than the length of the modulus n (that is, if

pRSA_rb->data_size > *pRSA_rb->key_token->n_BitLength* or if

pRSA_rb->output_size > *pRSA_rb->key_token->n_Length)*,

xcRSA processes the rightmost bytes of the input data and places its result in the rightmost bytes of the output buffer. For example,

```
char             inbuffer[256];
char             outbuffer[256];
xcRSA_RB_t       RSARB;
xcRSAKeyToken_t *pToken;
...
pToken->n_BitLength = 1024;
xcRSAKeyGenerate(...); /* Generate 1024-bit RSA keypair */
...
RSARB.data_in     = inbuffer;
RSARB.data_out    = outbuffer;
RSARB.data_size   = 256*8; /* Input data and output buffer are 2048 bits
*/
RSARB.output_size = 256;
xcRSA(&RSARB);
/*
xcRSA processes inbuffer[128] through inbuffer[255] and places the
result in outbuffer[128] through outbuffer[255].
outbuffer[0] through outbuffer[127] are set to 0x00.
*/
```

X9.31 support

The X9.31 signature generation process incorporates three steps:

1. The message is hashed.
2. The hash is encapsulated.
3. The encapsulated hash is wrapped to generate the signature.

xcRSA performs the third step as indicated by the X9.31 specification if

- *options* specifies **RSA_PRIVATE** and **RSA_X931_OPERATION**,
- *key_token->key_type* is **RSA_X931_PRIVATE_MODULUS_EXPONENT**, **RSA_X931_PRIVATE_CHINESE_REMAINDER**, or **RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER** (and had that value when the key was generated), and
- the buffer referenced by *data_in* contains a valid X9.31 encapsulated hash.

The first two steps are the application's responsibility.

Similarly, the signature verification process incorporates four steps:

1. The signature is opened (or unwrapped) to produce an encapsulated hash.

2. The format of the encapsulated hash is verified.
3. The hash value is extracted from the encapsulated hash.
4. The message is hashed and the value is compared to the extracted hash.

xcRSA performs the first step as dictated by the X9.31 specification if

- *options* includes **RSA_PUBLIC** and **RSA_X931_OPERATION**,
- *key_token->key_type* is **RSA_PUBLIC** (and the key itself corresponds to the private key used to generate the signature), and
- the buffer referenced by *data_in* contains a valid X9.31 signature.

The last three steps are the application's responsibility.

Return codes

Common return codes generated by this routine are:

PKAGood (that is, 0)	The operation was successful.
PKABadAddr	A pointer in the request block or the key token is invalid.
PKABadParm	An argument is not valid. Many structural deficiencies in the request block or key token can generate this error.
PKANoSpace	The operation failed due to lack of space (for example, there is no free memory available to the PKA driver).
PKARangeOverflow	The last *pRSA_rb->key_token->n_Length* bytes of the buffer described by *pRSA_rb->data_in*, when interpreted as a big-endian integer, exceed the value of the modulus *n*.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.9 ECC Operation Functions

The functions described in this section allow a coprocessor application to ask the Public Key Algorithm (PKA) driver to perform various cryptographic operations using the Elliptic Curve Cryptography (ECC) algorithm.

A coprocessor application must call *xcAttachWithCDUOption* before calling any of the functions in this section.

3.9.1 ECC key tokens

The PKA interface defines the *xcEccKeyToken_t* type to hold information about ECC (Elliptic Curve Cryptography) public and private keys. An item of type *xcEccKeyToken_t* consists of a descriptive header followed by a buffer containing information about the values of the various elements of the key. For example, the key token for an ECC public key includes the public part Q, public point *p*, the public point *y*, the curve type and curve length. The header indicates which elements are present and gives the length of and a pointer to each element. Elements are stored in big-endian order: the byte at the lowest address contains the most significant byte of the element.

The fields of the key token for an ECC public key are set as follows:

- *key_type* is **ECC_PUBLIC**.
- *tokenLength* is the length in bytes of the key token.
- *curve_type* is 0 for Prime or 1 for Brainpool.
- *pLength* is the curve length in bits.(160, 192, 224, 256, 320, 384, 512, or 521).
- *qLen* is the length in bytes of the public part Q.
- *pQ* points to the first (most significant) byte of the public part Q.
- *xLen* is the length of of the public point x.
- *px* points to the first (most significant) byte of the public point *x*.
- *yLen* is the length of of the public point y.
- *py* points to the first (most significant) byte of the public point *y*.

The remaining fields are ignored and should be set to zero.

The fields of the key token for a ECC private key are set as follows:

- *type* is **ECC_PRIVATE**.
- *key_token_length* is the length in bytes of the key token.
- *pLength*, q*Len*, *pq,* x*Len*, *px*, *yLen*, and *py* are set in the same manner as for an ECC public key.
- *dLen* is the length in bytes of the private part d,
- *pd* points to the first (most significant) byte of the private key *d*.

Note that an ECC private key token includes information about the corresponding ECC public key.

3.9.2 xcECCKeyGenerate - generate ECC keypair

xcECCKeyGenerate generates a key token for an ECC private key. The token includes information that defines the corresponding ECC public key. The user may specify values for the ECC private key, in which case the routine will return a new ECC public key.

Function prototype

```
unsigned int xcECCKeyGenerate( xcECCKeyGen_RB_t *pECCKeyGen_rb );
```

Input

On entry to this routine:

pECCKeyGen_rb contains the address of an ECC key generation request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *key_type* specifies the desired ECC key curve type and bit length. The curve type is within the most significant bytes of the field (*key_token->curveType* << ***SHIFT_KEY_TYPE****)* while the ECC token type (**ECC_PUBLIC** or **ECC_PRIVATE**)is within the least-significant bytes of the field.
- *key_size* contains the length in bytes of the buffer referenced by *key_token*. *key_size* must be greater than or equal to *sizeof(xcECCKeyToken_t)* + the sum of the *Len fields in *key_token*.
- *curve_size* contains the length in bits of the curve for the requested key.
- *curve_type* contains 0 for a Prime key or 1 for a Brainpool key.
- *key_token* points to a writeable buffer in which an item of type *xcECCKeyToken_t* can be stored.

  The fields of **key_token* must be initialized as follows:

  - *pLength* is the curve length in bits. This must match the curve type specified in *curve_type.*
  - *xcECCKeyGenerate* normally generates *d, Q, x, and y* randomly. If *key_type* specifies a key of **ECC_PUBLIC,** the value of *d* is instead taken from the appropriate field of **key_token*.
  - *dLen* contains the length in bytes of the buffer referenced by *pd*. This must be at least (*pLength* + 7)/8, that is, the number of bytes needed to hold *pLength* bits.
  - *qLen* contains the length in bytes of the buffer referenced by *pQ*. This must be at least ((((*pLength* +7)/8) * 2) + 1).
  - *xLen* contains the length in bytes of the buffer referenced by *px. xLen* must be greater than or equal to (*pLength* + 7)/8, that is, the number of bytes needed to hold *pLength* bits.
  - *yLen* contains the length in bytes of the buffer referenced by *py*. *yLen* must be greater than or equal to *(pLength* + 7)/8, that is, the number of bytes needed to hold *pLength* bits.
  - *x_length* must contain the length in bytes of the buffer referenced by *x_Ptr*. *x_length* must be greater than or equal to (*pLength* + 7)/8, that is, the number of bytes needed to hold *pLength* bits.
  - *pd*, *pQ*, *py*, and *px* must point to writeable buffers whose lengths are given by the corresponding **Len* fields.

    If *options* specifies **ECC_PUBLIC**,

    - the buffer defined by *pd/dLen* contains the value to be used for the private part *d*,
- *regen_data_size* is the length in bytes of the buffer referenced by *regen_data*. *regen_data_size* must be 0.
- *regen_data* may be NULL.

Output

On successful exit from this routine, the following fields of **(pECCKeyGen_rb->key_token)* are changed as noted:

- *key_type* is unchanged from the input value.
- *tokenLength* is set to *sizeof(xcECCKeyToken_t)*
- *curve_type* is set to one of 0 for Prime or 1 for Brainpool.
- *dLen* is set to the length in bytes of the public part d and the buffer defined by *pd/dLen* contains the public part d.

  If *pECCKeyGen_rb->key_type* specifies **ECC_PRIVATE**, this represents no change from the values of *dLen* and the buffer defined by *pd/dLen* on entry to the routine.
- *qLen* is set to the length in bytes of the public part Q and the buffer defined by *pQ/qLen* contains the public part *q*.
- *yLen* is set to the length in bytes of the public point *y* and the buffer defined by *py/yLen* contains the public point *y*.
- *xLen* is set to the length in bytes of the private point *x* and the buffer defined by *px/xLen* contains the private point *x*.

Notes

A key token for a ECC public key can be generated from the key token for the corresponding ECC private key by copying the buffer in the private key defined by *pd/dLen*, copying the *key_type, curve_type,* and *pLength* fields, and setting the key token's type field to **ECC_PUBLIC**.

None of the buffers defined to hold a piece of the generated key should overlap any of the buffers defined to hold a different piece of the generated key.

Return codes

Common return codes generated by this routine are:

PKAGood (that is, 0)	The operation was successful.
PKABadParm	An argument is not valid.
PKANoSpace	The buffer referenced by *pECCKeyGen_rb->key_token* is not large enough to hold the token generated by the call.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.9.3 xcECC - sign data or verify signature for data

xcECC generates a digital signature for or verifies that a specified digital signature is correct for an arbitrary amount of data using the ECC algorithm.

Function prototype

```
unsigned int xcECC( xcECC_RB_t *pECC_rb );
```

Input

On entry to this routine:

pECC_rb contains the address of a ECC request block whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *options* controls the operation of the function and must be set to the logical OR of constants from the following categories:

Sign or verify

options must include exactly one of the following constants:

ECDSA_SIGN	Compute the ECC signature for the input data.
ECDSA_VERIFY	Verify that the signature for the input data is correct.
ECC_ECDH	Perform an Elliptic Curve Diffie-Hellman computation

Key Type

options must include exactly one of the following constants:

ECC_PUBLIC	Use the public key to verify a signature.
ECC_PRIVATE	Use the private key to generate a signature.

If **ECDSA_SIGN** is specified, *pECC_rb->key_token* must contain a private key section *d*. If **ECC_VERIFY** is specified, *pECC_rb->key_token* must contain a public key section Q.

- *key_token* points to a buffer containing the ECC key token for the key to be used in the operation.

If *options* specifies **ECDSA_SIGN**, **key_token* must be the token for an ECC private key (e.g., *key_token->pd* must be non-null).

If *options* specifies **ECDSA_VERIFY**, **key_token* must be the token for an ECC public key (e.g., *key_token->pQ* must be present).

- *key_size* is the length in bytes of the buffer referenced by *key_token* (i.e, *key_size* equals *key_token->tokenLength*).
- *data_out_size* is the length in bytes of the buffer referenced by *data_out.*
- *data_out* points to a buffer in which the signature can be stored.

If *options* specifies **ECDSA_SIGN**, the buffer must be writeable.

If *options* specifies **ECDSA_VERIFY**, the buffer defines the signature that is to be verified.

- *data_in_size* is the length in bytes of the buffer referenced by *data_in.*
- *data_in* points to a buffer that contains the input data.

Output

On successful exit from this routine,

- If *options* specifies **ECDSA_VERIFY**, *xcECC* returns **PKAGood** if the signature verifies and **PKAEccVerifyFail** if the signature does not verify.
- If *options* specifies **ECDSA_SIGN**, *xcECC* returns **PKAGood** and the buffer defined by *data_out* contains the requested signature. In this case, *sdata_out_size* is changed to reflect the actual length in bytes of the signature.
- If *options* specifies **ECC_ECDH**, *xcECC* returns **PKAGood** and the buffer defined by *data_out* contains the calculated shared secret 'Z'. In this case, *sdata_out_size* is changed to reflect the actual length in bytes of the shared secret.

Return codes

Common return codes generated by this routine are:

PKAGood (that is, 0)	The operation was successful.
PKAEccVerifyFail	*pECC_rb->options* specifies **ECDSA_VERIFY** but the signature does not verify.
PKABadParm	An argument is not valid.
PKANoSpace	The operation failed due to lack of space.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.10 Large Integer Modular Math Operation Functions

The functions described in this section allow a coprocessor application to ask the Public Key Algorithm (PKA) driver to perform specific modular operations on large integers. Currently, the following operations are supported:

Modular multiplication (*C = A x B mod N*)

Modular exponentiation ($C = A^B \bmod N$)

Modular reduction (*C = A mod N*)

Modular inversion (*C = INV(A) mod N)*

Multiplication (*C = A x B)*

A coprocessor application must call *xcAttachWithCDUOption* before calling any of the functions in this section.

3.10.1 Large integers

A large integer is described by a structure of type *xcModMath_Int_t*. The fields of this structure are:

- *bytesize*, which specifies the length in bytes of the buffer that contains the large integer. *bytesize* must be less than or equal to **MODM_MAXBYTES**.
- *bitsize*, which specifies the number of bits in the large integer. *bitsize* must be less than or equal to 8**bytesize*.
- *buffer*, which points to the buffer that contains the large integer.

A large integer is stored in big-endian order (*buffer[0]* is the most significant byte of the integer) and occupies the first (*bitsize* + 7)/8 bytes of the buffer that contains it. A large integer is always nonnegative (that is, there is no sign bit).

A large integer that is passed as an input argument to the large integer modular math functions may contain leading zero bits (that is, the most significant bit of the integer may be zero). Any bits in the most significant byte that are not part of the large integer are ignored.

A large integer that is generated as an output by the large integer modular math functions does not contain leading zero bits (that is, the most significant bit of the integer is one). Any bits in the most significant byte that are not part of the large integer are set to zero.

3.10.2 xcModMath - perform modular computations

xcModMath performs one of the following operations on large integers:

- Modular multiplication (*C = A x B mod N*)
- Modular exponentiation (*C = A^B mod N*)
- Modular reduction (*C = A mod N*)
- Multiplication *(C = A x B)*
- Modular inversion (*C = INV(A) mod B)*

Function prototype

```
unsigned int xcModMath( xcModMath_t *pModMath );
```

Input

On entry to this routine:

pModMath is a pointer to a Modular Math request block whose fields are set as follows:

- *cmn* is set as described in Common structure on page 21.
- *cmd* controls the operation of the function and must be set to the logical OR of constants from the following categories:

  Operation

- *cmd* must include exactly one of the following constants:

MODM_MULT	Compute C = A x B mod N
MODM_EXP	Compute C = A^B mod N
MODM_MOD	Compute C = A mod N
MULTPLY	Compute C = A x B
MULTINV	Compute C = 1/A mod N

  Large integer byte order

- *cmd* must include **MODM_BIG**.
- *numInts* is the number of elements in the array referenced by *pModMath_ints*. If *cmd* specifies **MODM_MULT** or **MODM_EXP**, *numInts* must be at least 4. If *cmd* specifies **MODM_MOD**, **MULTIPLY,** or **MULTINV,** *numInts* must be at least 3.
- *pModMath_ints* points to an array of large integer descriptors. Its elements are as follows:
  - *pModMath_ints[MODM_C]* is the descriptor for *C*, the result of the operation. The buffer defined by *pModMath_ints[MODM_C].bytesize* and *pModMath_ints[MODM_C].*buffer must be large enough to hold the result of the operation, i.e., must be as large as the modulus *N*.

pModMath_ints[MODM_C].bitsize is not used.

- *pModMath_ints[MODM_N]* is the descriptor for *N*, the modulus.
- *pModMath_ints[MODM_A]* is the descriptor for *A*, the first (or only) operand.

  If *cmd* specifies **MODM_MULT** or **MODM_EXP** or **MULTINV**, the value of *A* must be less than the value of the modulus *N*.

  If *cmd* specifies **MODM_EXP**, the value of *A* must not be zero.
- *pModMath_ints[MODM_B]* is the descriptor for *B*, the second operand.

  If *cmd* specifies **MODM_MULT**, the value of *B* must be less than the value of the modulus *N*.

  If *cmd* specifies **MODM_EXP**, the value of *B* must not be zero.

  If *cmd* specifies **MODM_MOD**, *pModMath_ints[MODM_B]* is not used.

Output

On successful exit from this routine, *pModMath_ints[MODM_C].bitsize* contains the number of bits in the result and the buffer defined by *pModMath_ints[MODM_C].buffer/ pModMath_ints[MODM_C].bytesize* contains the value of the result.

Notes

None of the buffers defined to hold a large integer used or produced in the operation should overlap any of the buffers defined to hold a different large integer used or produced in the operation.

Return codes

Common return codes generated by this routine are:

PKAGood (that is, 0)	The operation was successful.
PKABadParm	An argument is not valid. For example, cmd does not specify MODM_MULT, MODM_EXP or MODM_MOD or an invalid operation was requested (that is, 00 mod N).
PKANoSpace	The operation failed due to lack of space.
PKABadAddr	One of the large integers supplied as inputs is invalid (for example, bitsize or bytesize exceeds the maximum or buffer is NULL).
PKARangeOverflow	The buffer provided to hold the result of the operation is not large enough.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.11 Random Number Generator Operation Functions

The functions described in this section allow a coprocessor application to request services from the Determinisitic Random Bit Generator (DRBG) Driver, which obtains random bits from a pseudo-random number generator that is regularly reseeded by the coprocessor's hardware noise source. The random bits meet the standards described in Deterministic Random Bit Generator (DRBG) NIST Special Publication 800-90A.

A coprocessor application must call *xcAttachWithCDUOption* before calling any of the functions in this section.

3.11.1 xcDRBGinstantiate – instantiate a DRBG random number generator

The *xcDRBGinstantiate* function acquires entropy input combined with a nonce and optional personalization string to create a seed from which the internal state is created. The use of a hash derivation function, as defined in section 10.4.1 of NIST SP800-90A, is used to create the internal state using calls to an AES cryptographic function. Known answer tests (KATs) on the “Instantiate” function will be performed by each operational instantiation. The coprocessor maintains a state table for each instantiated random number generator, and returns the handle of the state table to the caller.

Note: See NIST publication SP800-90A for details about hash derivation:
http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf

Function prototype

```
int xcDRBGinstantiate( xcDRBGinstan_t *pDRBGinst );
```

Input

On entry to this routine:

pDRBGinst is a pointer to a structure with fields set as follows:

- *cmn* is set as described in Common structure on page 21.
- *mechanism* is set to the constant **DRBG_CTR_MECH_AES256**.
- *securStren* is set to the constant **DRBG_SSTR_256** for 256-bit security strength.
- *pPersStr* is a pointer to the personalization string (which contributes to the seed material) or NULL if no string is to be supplied.
- *persStrLen* is 4 bytes indicating the length of *pPersStr.* This must be 0 if *pPersStr* is a NULL pointer.
- *predRFlag* when set to a value of 1, provides the option of selecting prediction resistance during calls to *xcDRBGgenerate*. A value of 0 precludes prediction resistance during *xcDRBGgenerate* calls. The value supplied here is logically AND’ed with the Prediction Resistance Request value provided in *xcDRBGgenerate*. If the result is 1, a re-seed will be performed during *xcDRBGgenerate*, thereby providing prediction resistance.

Notes:

- Instantiating the DRBG is a computationally expensive operation. Suggested use is to pre-allocate the DRBG instance early in your application and to use that DRBG throughout your application. There is usually no need to instantiate more than one DRBG.
- Customers should not instantiate more than nine DRBGs in the same application. Again, one DRBG is typically all that is neede.

Output

On successful exit from this routine, *pDRBGinst->handle* is the handle for the instantiated random number generator.

Return codes

Common return codes generated by this routine are:

RN_Success (that is, 0)	The operation was successful.
RN_Invalid	One of the parameters in the structure pointed to by the *pDRBGinst* parameter is invalid.
RN_NotWorking	DRBG not working: DRBG was working when the request was received, but the operation failed.
RN_None	DRBG was offline when the request was received.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.11.2 xcDRBGgenerate – generate random number

The *xcDRBGgenerate* function is used to generate pseudo-random bits after instantiation or reseeding. This function:

1. Invokes the reseed function to obtain sufficient entropy if the end of seed-life has been reached OR (prediction resistance is required AND the prediction resistance flag is set in the internal state selected by the state handle);
2. Generates the requested pseudo-random bits using the generate algorithm and the internal state array indexed by the state handle;
3. Updates the internal state;
4. Returns the requested pseudo-random bits to the consuming application;
5. Uses the SHA256 hash function and Hashgen function. The Hashgen function is defined in section 10.4.1 of NIST SP800-90A;
6. Performs at most only one reseed. If the caller has requested prediction resistance at the exact moment that the seed has reached end of life, then only one reseed to take place, not two;
7. Implements the flowchart according to Figure 8, section 10.1.1.2 of NIST SP800-90A; and
8. Performs Known Answer Tests (KATs) on the generate function before the first usage of the function in operational mode (i.e. the first use ever), and at reasonable intervals (at maximum number of requests between reseeds).

Function prototype

```
int xcDRBGgenerate( xcDRBGgenerate_t *pDRBGgen );
```

Input

On entry to this routine:

pDRBGgen is a pointer to a structure with fields set as follows:

- *cmn* is set as described in Common structure on page 21.
- *pAddStr* is a pointer to additional data for reseeding. If no additional data is provided, this may be NULL.
- *addStrLen* is the length of the additional input in bytes. If *pAddStr* is NULL, this value must be 0.
- *handle* is the value returned from the *xcDBRGinstantiate* call in the *pDBRGinstan->handle* field.
- *reqNumBits* is the number of bits requested to be generated. It must be less than *maxBitsPerReq,* or 262144 (32 Kbytes).
- *reqStrength* must be set to the constant **DRBG_SSTR_256**, for 256-bit security strength.
- *predRRq* must be set to one of the following constants:

DRBG_PRR_ON	1	Reseeding is requested. If the prediction resistance byte in the internal state was set during the *xcDBRGinstantiate* call with input parameter *predRFlag* set to 1, the generator will be reseeded before the random bits are generated. Selecting this option will slow performance. If the *predRFlag* was set to 0 during the *xcDRBGinstantiate* call, then setting *DRBG_PRR_ON* to 1 will cause the *RN_Invalid* return code to be returned
DRBG_PRR_OFF	0	Reseeding not requested. The bit generator will not be reseeded

		UNLESS the previous reseed has reached the end of its life cycle.

- *options* must include exactly one of the following parity bit constants:

RANDOM_RANDOM	Generate the requested number of random bytes. Disregard parity.
RANDOM_ODD_PARITY	Generate the requested number of random bytes, then set or clear the least significant bit of each byte as necessary so that the number of bits set in each byte is odd.
RANDOM_EVEN_PARITY	Generate the requested number of random bytes, then set or clear the least significant bit of each byte as necessary so that the number of bits set in each byte is even.

- In addition, *options* may include any of the following source and filter constants:

RANDOM_SW	Obtain random bits from a pseudo-random number generator (PRNG). The PRNG is seeded from the hardware noise source. This option is the default and does not have to be specified. It is allowed for legacy purposes. The legacy option *RANDOM_HW* is no longer supported since retrieving random bits directly from the hardware is a violation of the NIST SP800-90A standard. Using *RANDOM_HW* will be ignored and a random number will be returned from the PRNG.
RANDOM_NOT_WEAK	Random numbers that are weak, semi-weak, or possibly weak when used as DES keys are not returned. The number is checked after any requested parity bits are generated.

- *pOutRNbits* is a pointer to a buffer to hold the generated random number. If this buffer is larger than is required to hold the number of bits requested, the random number will be left-justified within the buffer. Bits to the right of the generated bits WILL NOT BE MODIFIED.
- *outRNLen* is the size in bytes of the buffer to hold pseudo random bits. This size may be larger than required to hold the number of bits requested.

Output

On successful exit from this routine, *pDRBGgen->pOutRNbits* contains the generated random number.

Return codes

Common return codes generated by this routine are:

RN_Success (that is, 0)	The operation was successful.
RN_Invalid	One of the parameters in the structure pointed to by the

	pDRBGgen parameter is invalid.
RN_NotWorking	DRBG not working: DRBG was working when the request was received, but the operation failed
RN_None	DRBG was offline when the request was received.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.11.3 xcDRBGreseed – reseed an instance of the RNG

The *xcDRBGreseed* function reseeds the random number generator. It acquires new entropy input with sufficient entropy to support the security strength and combines the entropy with internal state variables and any optional additional input that is provided to create a new seed and a new internal state.

Reseeding can be:

- explicitly requested by the caller using this API,
- performed “under the covers” when a prediction resistance request parameter is provided to *xcDRBGgenerate* function AND the prediction resistance flag in the internal state is *on*, or
- triggered by the generate function "under the covers" when a predetermined number of generate requests have been made (i.e., at the end of the seed-life).

Function prototype

```
int xcDRBGreseed( xcDRBGreseed_t *pDRBGrsd );
```

Input

On entry to this routine:

pDRBGrsd is a pointer to a structure with fields set as follows:

- *cmn* is set as described in Common structure on page 21.
- *pAddStr* is a pointer to additional data for reseeding the RNG. If this pointer is NULL, the RNG will be reseeded entirely from the hardware device.
- *addStrLen* is the length of the additional input in bytes. If *pAddStr* is NULL, this value must be 0. The maximum additional length allowed is 8192 bytes. Exceeding this limit will result in a return code of *RN_Invalid* and an event log entry of “xcDRBGreseed: Additional string length too large.” to be created.
- *handle* is the state handle identifying the internal state for the instantiation the caller wishes to use (the value returned from the *xcDBRGinstantiate* call in the *pDBRGinstan->handle* field).

Output

This function has no outputs. On return from this function, the state of the RNG has been reseeded.

Return codes

Common return codes generated by this routine are:

RN_Success (that is, 0)	The operation was successful.
RN_Invalid	One of the parameters in the structure pointed to by the *pDRBGrsd* parameter is invalid.
RN_NotWorking	DRBG not working: DRBG was working when the request was received, but the operation failed
RN_None	DRBG was offline when the request was received.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.11.4 xcDRBGuninstantiate – uninstantiate DRBG

The *xcDRBGunintantiate* function erases the contents of the stored state for the given handle and makes the space available for the next instantiation of the RNG.

Function prototype

```
int xcDRBGuninstantiate( xcDRBGuninstantiate_t *pDRBGuninst );
```

Input

On entry to this routine:

pDRBGuninst is a pointer to a structure whose fields are initialized as follows:

- *cmn* is set as described in Common structure on page 21.
- *handle* is the handle returned from a previous *xcDRBGinstantiate* call, indicating which DRBG is to be uninstantiated.

Output

This function has no outputs. On successful exist, the state indicated by *xcDRBGuninst->handle* has been cleared and returned to the list for the next call to *xcDRBGinstantiate.*

Return codes

Common return codes generated by this routine are:

RN_Success (that is, 0)	The operation was successful.
RN_Invalid	One of the parameters in the structure pointed to by the *pDRBGuninst* parameter is invalid.
RN_NotWorking	DRBG not working: DRBG was working when the request was received, but the operation failed

Refer to *xc_err.h* for a comprehensive list of return codes.

3.12 Configuration functions

The functions described in this section allow a coprocessor application to interact with the 4767 driver to configure certain processor features or obtain information about the coprocessor.

A coprocessor application must call *xcAttachWithCDUOption* before calling any of the functions in this section.

3.12.1 Privileged operations

Some of the functions described in this section can only be performed by users and/or applications that have root authority.

3.12.2 Date/time

Support for querying and modifying the time-of-day is provided through the Linux kernel. This support is provided within the generic Linux kernel that is provided on the PCIe Cryptographic Coprocessor. Call the function *settimeofday*() to set the clock.

3.12.3 xcGetConfig - get coprocessor configuration

xcGetConfig obtains information about the coprocessor.

Function prototype

```
unsigned int xcGetConfig( xcGetConfigRB_t *pConfig_rb );
```

Input

On entry to this routine:

pConfig_rb points to a Get Config request block whose fields are set as follows:

- *cmn* is set as described in Common structure on page 21.
- *pAdptInfo* is a pointer to a buffer in which a data item of type xcAdapterInfo_t may be stored.
- *pSzAdptInfo* is a buffer in which the length of the buffer pointed to by *pAdptInfo* is stored.

Output

On successful exit from this routine:

**pConfig_rb->pAdptInfo* contains as much information about the coprocessor as could be returned in the buffer provided. If the buffer was sufficiently large, this is a full *xcAdapterInfo_t* structure whose fields are set as indicated below. If the buffer was too small, the structure is truncated.

**ipConfig_rb->nfo_length* contains the length of the full *xcAdapterInfo_t* structure. If this is larger than the buffer originally provided, the application can acquire a suitably-sized buffer and repeat the call.

The fields of the *xcAdapterInfo_t* structure are set as follows:

- *sid.ID* is **STRUCT_xcAdapterInfo**.
- *sid.len* is the length of this structure.
- *FpgaRevId* is the revision of the code.
- *ASICRevId* is the revision of the Application Specific Integrated Circuit.
- *VPD* contains the coprocessor Vital Product Data. Its fields are set as described in the function “xcGetAdapterData - retrieve identification data from a coprocessor”on page 15.
- *CardRevId* is the base card FPGA ID.
- *POST_Version* indicates which version of the coprocessor power-on self test (POST) microcode is installed. This microcode operates in three phases (POST0, POST1, and POST2), so *POST_Version* contains three fields. Each field contains the version and release of the corresponding phase of POST.
- *MiniBoot_Version* indicates which version of the miniboot microcode is installed. Miniboot initializes the coprocessor operating system and controls updates to software in flash memory. Miniboot operates in two phases (MiniBoot0 and MiniBoot1), so *MiniBoot_Version* contains two fields. Each field contains the version and release of the corresponding phase of miniboot.
- *OS_Name* contains the unquoted ASCII characters "Linux ".
- *OS_Version* indicates which version of the operating system is installed.

- *CPU_Speed* is the speed in megahertz of the coprocessor CPU. *This field is in little-endian byte order.*
- *HardwareStatus* contains the current state (active high) of the hardware tamper bits (see the various **HW_*** constants in *xc_types.h*).
- *AdapterID* is a unique serial number incorporated in the coprocessor chip that implements the real-time clock and the BBRAM. It can be used to distinguish the physical coprocessor from all others but is unrelated to the serial number in *VPD.sn*.
- *flashSize* is the size of the coprocessor's flash memory. The unit of measurement is 1M (that is, *flashSize* == 1 implies 1 megabyte of flash). *This field is in little-endian byte order.*
- *bbramSize* is the size of the coprocessor's battery-backed RAM. The unit of measurement is 1K (that is, *bbramSize* == 16 implies 16 kilobytes of BBRAM).
- *dramSize* is the size of the coprocessor's regular (non-battery-backed) random access memory (RAM). The unit of measurement is 1K (that is, *dramSize* = 128 implies 128 kilobytes of RAM).
- *reserved* is undefined.

Return codes

Common return codes generated by this routine are:

SCCGood (that is, 0)	The operation was successful.
SCCBadFun	Invalid function.
SCCBadFlags	Invalid options.
SCCBadParm	Invalid parameter.
SCCNotFound	Device driver access error.
SCCNotAuthorized	Invalid requestor authority.
SCCGenErr	General error.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.12.4 xcClearILatch - clear coprocessor intrusion latch

The coprocessor provides an input pin (the "intrusion latch") to which an external device can be connected. For example, the user might connect a sensor that detects unauthorized attempts to open the case of the host in which the coprocessor is installed.

An application with appropriate privileges can determine the state of the intrusion latch by calling *xcGetConfig*. Neither the coprocessor operating system nor the microcode that monitors attempts to compromise the coprocessor's secure environment take any action when the intrusion latch is triggered. *xcClearILatch* resets the coprocessor intrusion latch.

Function prototype

```
unsigned int xcClearILatch( xcClearILatchRB_t *pILatch );
```

Input

On entry to this routine:

pILatch is a pointer to a Clear Intrusion Latch buffer whose fields are set as follows:

- cmn is set as described in Common structure on page 21.

Output

None.

Notes

An application must run as root to call *xcClearILatch*.

Return codes

Common return codes generated by this routine are:

SCCGood (that is, 0)	The operation was successful.
SCCGenErr	General error.

Refer to *xc_err.h* for a comprehensive list of return codes.

3.12.5 xcClearLowBatt - clear coprocessor low battery warning latch

The coprocessor includes batteries that allow the coprocessor to detect certain attempts to compromise its physical integrity. If the batteries are allowed to drain completely, the coprocessor clears its secrets and resets itself as if it had detected an attempt to tamper with the secure hardware. The coprocessor is then in a permanently disabled state and cannot be used again. The coprocessor therefore monitors the battery voltage and triggers the low battery warning latch if it drops below a certain value [11].

An application with appropriate privileges can determine the state of the low battery warning latch by calling *xcGetConfig* . Neither the coprocessor operating system nor the microcode that monitors attempts to compromise the coprocessor's secure environment takes any action when the low battery warning is triggered. When the batteries are replaced, the low battery warning latch is reset automatically.

Note: This function is deprectated.

Function prototype

```
unsigned int xcClearLowBatt( xcClearLowBatt_t *pxcBattery );
```

Input

On entry to this routine:

pxcBattery is a pointer to a Clear Low Battery structure whose fields have been set as follows:

- cmn is set as described in Common structure on page 21.

Output

None.

Notes

An application must run as root to call *xcClearLowBatt*.

Return codes

Common return codes generated by this routine are:

HDDGood (that is, 0)	The operation was successful.

Refer to *xc_err.h* for a comprehensive list of return codes.

11 The precise value is chosen to provide a reasonable expectation that the low battery warning latch will be triggered at least one month before the batteries are exhausted.

3.13 Outbound authentication functions

The functions described in this section allow a coprocessor application to request services from the Outbound Authentication (OA) daemon, which supports cryptographic operations and data structures that allow the coprocessor application to authenticate itself to another agent and to engage in a wide range of cryptographic protocols. In particular, a coprocessor application can use these functions to:

- Prove to another agent that the coprocessor on which the application is running has not been tampered with.
- Provide another agent a list of all the software that has ever been loaded on the coprocessor that could have revealed the application's secrets or compromised the authentication scheme.
- Report in a manner that cannot be forged (unless the authentication scheme has been compromised) the status of the coprocessor, including its serial number and the identity of the software it contains.
- Perform general cryptographic operations (encryption, decryption, signing, and verification) and engage in cryptographic protocols (for example, key exchange) using keys whose validity is assured by the authentication scheme.

The remainder of this introduction describes certain aspects of the coprocessor architecture that form the basis of the authentication scheme and provides an overview of the authentication scheme. For a thorough overview of the coprocessor's security goals and a description of the security architecture, refer to *Building a High-Performance, Programmable Secure Coprocessor, Research Report RC21102* published by the IBM T.J. Watson Research Center in February, 1998. A revised version of this paper appeared in Computer Networks 31:831-860, April 1999.

3.13.1 Coprocessor architecture

The nonvolatile memory on a coprocessor is partitioned into four segments, each of which can contain program code and sensitive data:

- Segment 0 contains one portion of Miniboot, the most privileged software in the coprocessor. Miniboot implements, among other things, the protocols that ensure nothing is loaded into the coprocessor without the proper authorization. The code in segment 0 is in ROM.
- Segment 1 contains another portion of Miniboot. The code in segment 1 is saved in flash. The division of Miniboot into a ROM portion and a flash portion preserves flexibility while guaranteeing a basic level of security.
- Segment 2 contains the coprocessor operating system (Linux). This code is saved in flash.
- Segment 3 contains the coprocessor application. This code is saved in flash.

A segment's sensitive data is either saved in high-speed-erase battery-backed RAM (HSE BBRAM) or is encrypted [12] and saved in regular BBRAM or in flash. The coprocessor incorporates special hardware (independent of the CPU and whose operation cannot be affected by software) that prevents the operating system and any application (that is, code in segments 2 and 3) from modifying sensitive information in flash or reading secrets in BBRAM.

One of the data items Miniboot saves in BBRAM in segment 0 is a 32-bit "boot counter." The boot counter is initialized to zero during manufacture; the Miniboot code in segment 0 increments the boot counter each time the coprocessor boots. The authentication scheme uses the boot counter as a timestamp in

12 Miniboot generates an AES key for this purpose and saves it in HSE BBRAM. The encryption (on write) and decryption (on read) are performed transparently by the filesystem.

many contexts.

Information that identifies the code loaded in a segment is also saved in the segment. This information includes:

- The identity of the owner of the segment, that is, the party responsible for the software that is loaded in the segment. Owner identifiers are two bytes long. IBM owns segment 1 and issues an owner identifier to any party that is developing code to be loaded into segment 2. An owner of segment 2 issues an owner identifier to any party that is developing code that is to be loaded into segment 3 under the segment 2 owner's authority (that is, while the segment 2 owner owns segment 2).
- The name (an arbitrary string no longer than 80 bytes), revision number (a two-byte integer), and SHA-256 hash of the software in the segment. The hash that covers a segment is computed by the software in segment 1.

3.13.2 Overview of the authentication scheme

Initialization

During manufacture, a coprocessor generates a random RSA keypair (the "Device Keypair") and exports the public key. The factory incorporates the Device Public Key into a certificate and signs the certificate using the private half of a keypair owned and controlled by the factory (an "IBM Class Root Keypair"). The coprocessor imports and saves this certificate and a certificate containing the IBM Class Root Public Key. The latter certificate is signed using the private half of a keypair owned and controlled by IBM (an "IBM Root Keypair") [13].

Updates to segment 1

Whenever the software in segment 1 is updated, the software in segment 1 that is about to be replaced:

1. Generates a new random ECC keypair (a "Transition Keypair").
2. Incorporates the new Transition Public Key and information that identifies the new segment 1 software into a certificate and signs the certificate using the private half of the active segment 1 keypair. If this is the first time the software in segment 1 has been updated, the active segment 1 keypair is the Device Keypair. Otherwise the active segment 1 keypair is the Transition Keypair created the last time segment 1 was updated.
3. Deletes the private half of the active segment 1 keypair and makes the new Transition Keypair the active segment 1 keypair.

The result is a chain of certificates that links the IBM Class Root Certificate and the most recently created Transition Certificate. If an adversary tampers with the coprocessor, the coprocessor clears the active segment 1 private key. Any subsequent attempt to assert the coprocessor has not been tampered with fails because the adversary does not possess any of the private keys used to create the certificate chain. The adversary also does not possess the IBM Root Private Key and so cannot forge an IBM Class Root Certificate. The adversary therefore cannot sign a nonce with an existing key or create a new key that is linked to the IBM Class Root Certificate to do so.

The certificate chain also identifies every piece of software that has ever been loaded into segment 1. Although a malicious or defective program loaded into segment 1 can reveal its own Transition Private Key (and so compromise any certificates that are subsequently generated), such a program cannot mask its presence because its identity is incorporated into a certificate using a private key whose value the program never knows. Once the program's behavior is recognized, a host can treat a certificate chain that

13 The value of the IBM Root Public Key appears in “IBM root public key” on page 144.

includes the program with the suspicion it warrants [14].

Changes to segments 2 and 3

The software in segment 1 also manipulates the certificate chain when changes are made to segment 2 or to segment 3. The specific actions segment 1 performs depend on whether the changes affect the sensitive data saved in segment 3 BBRAM. Certain operations dictate that segment 1 clear segment 3 BBRAM; others do not [15].

The authentication scheme defines an "epoch" to be the maximum possible lifetime of a piece of sensitive data in segment 3 BBRAM. An epoch begins when an event occurs that loads runnable code into segment 3 (or leaves any code that is already in segment 3 in a runnable state) and causes segment 1 to erase the contents of segment 3 BBRAM. An epoch ends the next time segment 3 BBRAM is cleared for any reason (for example, because the software in segment 3 or the software in segment 2 has been unloaded or has been reloaded in a manner that clears BBRAM).

The authentication scheme defines a "configuration" to be a period of time during which the software in a coprocessor does not change. A configuration begins when an event occurs that changes the software in any segment and that either loads runnable code into segment 3 or leaves any code that is already in segment 3 in a runnable state. A configuration ends the next time the software in any segment changes or when the code in segment 3 is no longer runnable. A configuration also ends if the epoch in which the configuration started ends [16].

An application can ask the OA library to create one or more keypairs the application can use to perform general cryptographic operations. The application can specify that the private half of the keypair in question is to be used only during the current configuration (a "Configuration Keypair") or is to be used for the duration of the current epoch (an "Epoch Keypair"). The OA daemon also creates a certificate for the keypair and signs it using the private half of an "Operating System" keypair.

Configuration start

When a configuration begins, the software in segment 1 creates:

1. An operating system keypair
2. A certificate that contains the public half of the keypair

The certificate is signed using the private half of the active segment 1 keypair.

Configuration end

When a configuration ends, the software in segment 1 erases:

---

14 Note that although a malicious program can attempt to hide by adopting the name and revision number of a benign program, the SHA-256 hash that is saved in segment 1 is computed by the previous occupant of segment 1 and cannot be forged.

15 Refer to “Using Signer and Packager” in the *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide* for a discussion of which operations clear segment 3 BBRAM.

16 The notions of “epoch” and “configuration” are actually more general than these definitions indicate. For example, certain actions can cause the sensitive data in segment 3 BBRAM to be erased without affecting any sensitive data in segment 2 BBRAM. In that case, the current “segment 3” epoch ends while the current “segment 2” epoch continues. Similarly, a change to the software in segment 3 begins a new “segment 3” configuration but does not affect the current “segment 2” configuration. The only context in which these distinctions might be of interest to an application on the host is when interpreting the epoch_start, config_start, and config_count fields in a layer name. See “Layer names and layer descriptors” on page 131 for details.

1. The private portion of any configuration keypairs the application has caused to be created
2. The private portion of the current operating system keypair

The certificates for such keypairs are retained (since there may still be sensitive data that was encrypted using the private half of one of the keypairs) but they are now "inactive."

Epoch end

When an epoch ends, the software in segment 1 erases:

1. Any configuration keypairs and epoch keypairs the application has caused to be created
2. Any operating system keypairs that have been created

The software in segment 2 subsequently erases the certificates that contain the public portions of the keypairs that the software in segment 1 erased.

Examples

Figure 2 Initial certificate chain shows the certificate chain after an application has been loaded into a coprocessor for the first time and has asked the OA daemon to create an Epoch Keypair. The figure also indicates which certificates contain a public key whose corresponding private key is also stored on the coprocessor (the Device Private Key is deleted when the first Transition Keypair is created).

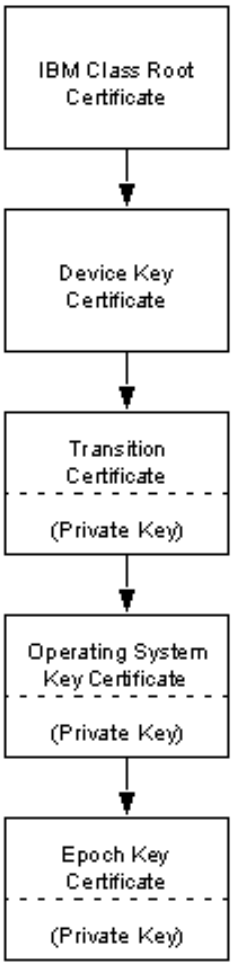

Figure 2 Initial certificate chain

Figure 3 Application generates configuration key shows the effect of a subsequent call by the application to the OA daemon to create a Configuration Keypair. The OA daemon adds the new certificate to the chain.

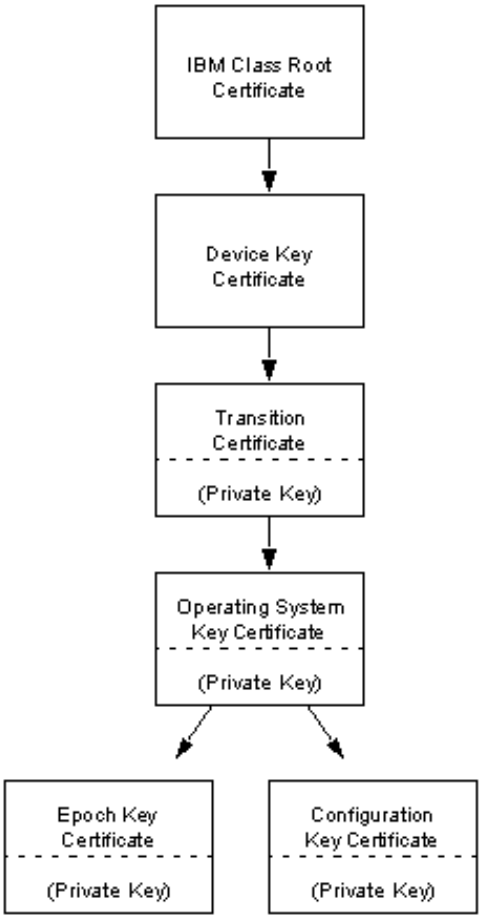

Figure 3 Application generates configuration key

Figure 4 illustrates the changes to the certificate chain caused by subsequently loading a new version of the operating system into the coprocessor in a manner that does not clear segment 3 BBRAM. This changes the configuration and so the private keys in the Operating System Keypair and the Configuration Keypair are deleted. This is appropriate since the configuration the Operating System Key Certificate names is no longer current and because Configuration Keypairs are by definition effective only during a single configuration. The private key in the Epoch Keypair is retained since the data in segment 3 BBRAM remains the same. The software in segment 1 creates a new Operating System Keypair and signs its certificate.

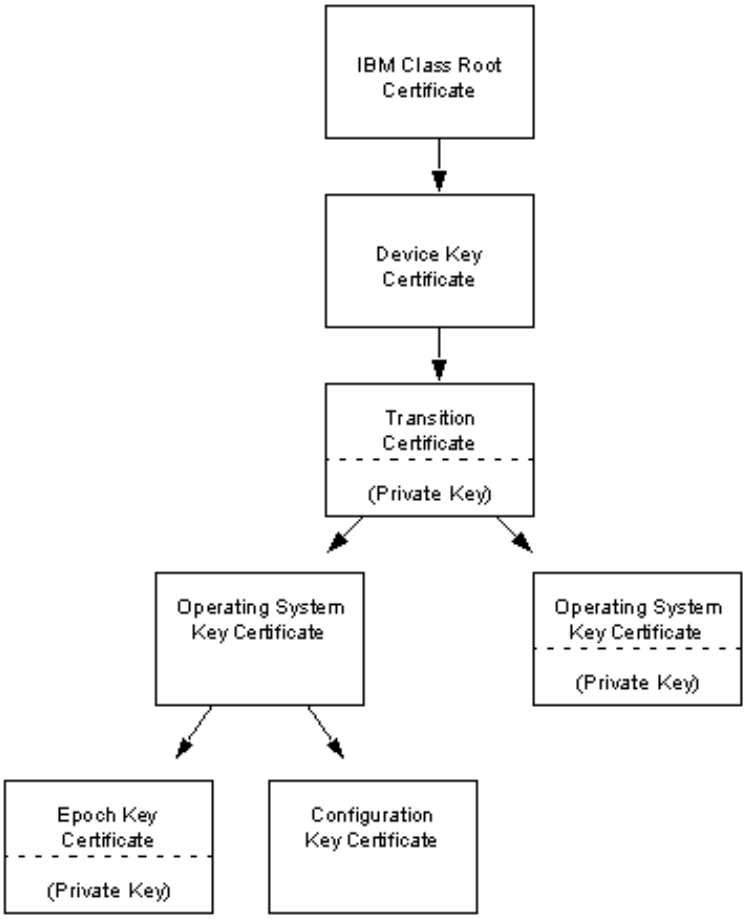

Figure 4 Operating system updated

Since the existing Configuration Keypair no longer has a private key, the application must ask the OA library to create a new Configuration Keypair if the application needs a private key. Figure 5 shows the new certificate chain. The application could generate another Epoch Keypair (whose certificate would be signed by the new Operating System Private Key), even though the epoch has not changed. One reason to do so (and to discontinue use of the original Epoch Private Key or delete the original Epoch Keypair entirely) is that it is easier to locate the current Operating System Key Certificate using the new Epoch Key Certificate rather than the old one, and the current Operating System Key Certificate is the one that identifies the new software in segment 2.

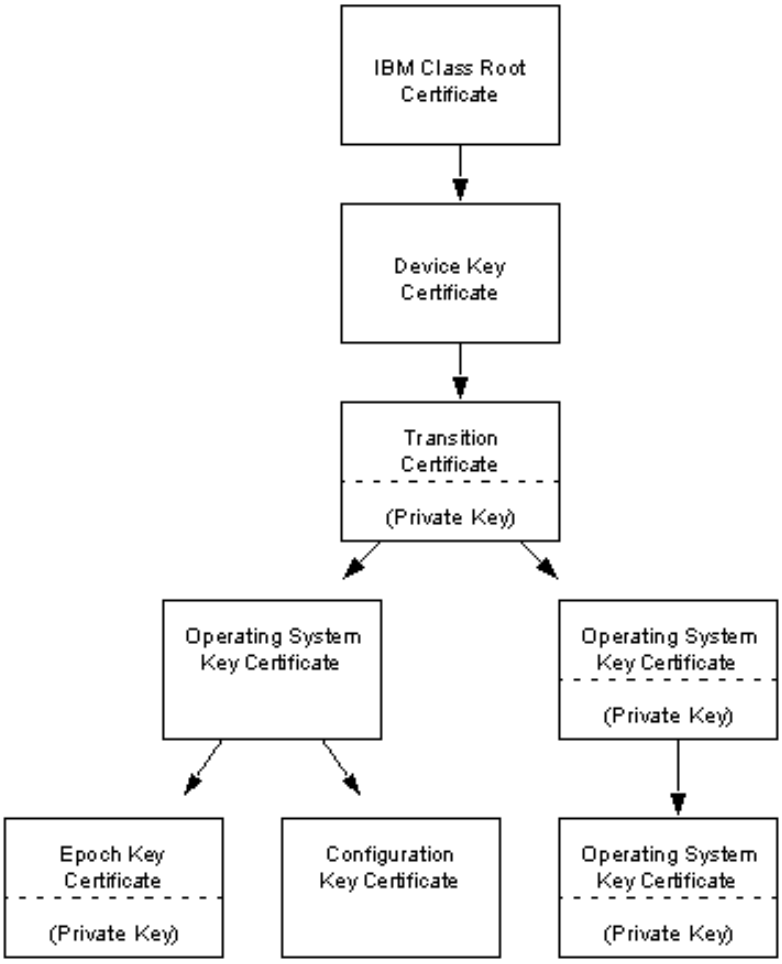

Figure 5 Application generates new configuration key

Figure 6 illustrates the changes to the certificate chain caused by subsequently updated segment 1 update in a manner that does not clear segment 3 BBRAM. The existing Configuration Private Key and Operating System Private Key are deleted. A new Transition Certificate and Operating System Certificate are added to the certificate chain and new private keys are created.

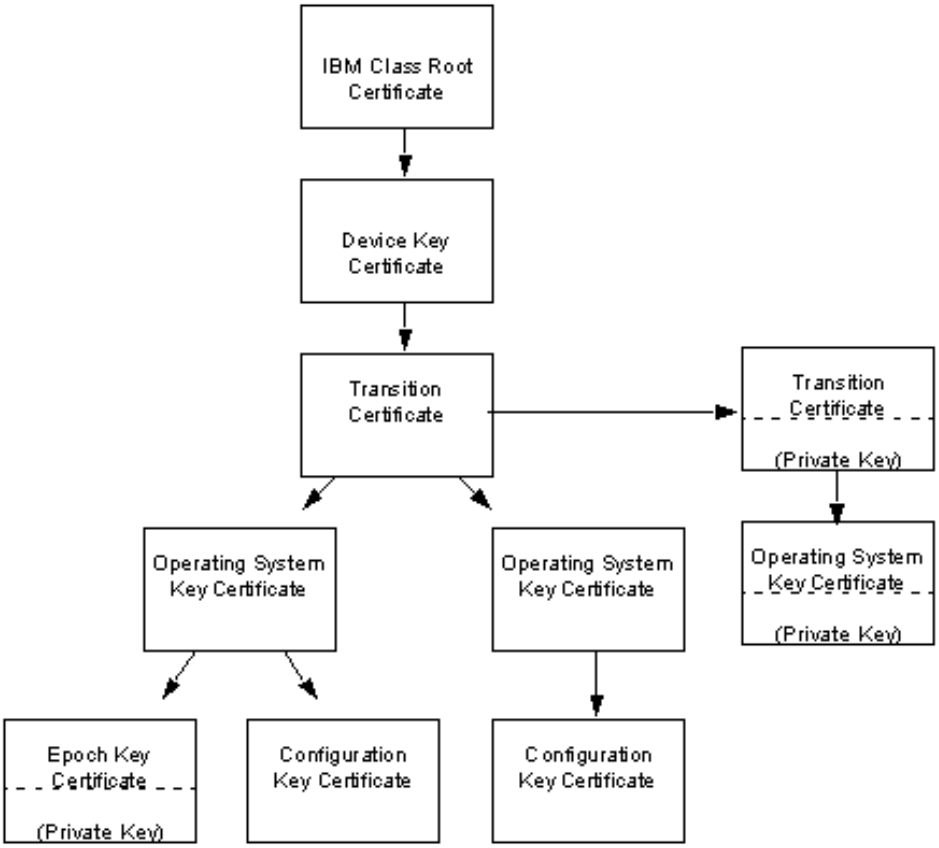

Figure 6 Miniboot updated

Figure 7 shows the effects of the following requests made by the application: creation of a new Epoch Keypair, creation of a new Configuration Keypair, and deletion of the original Epoch Keypair.

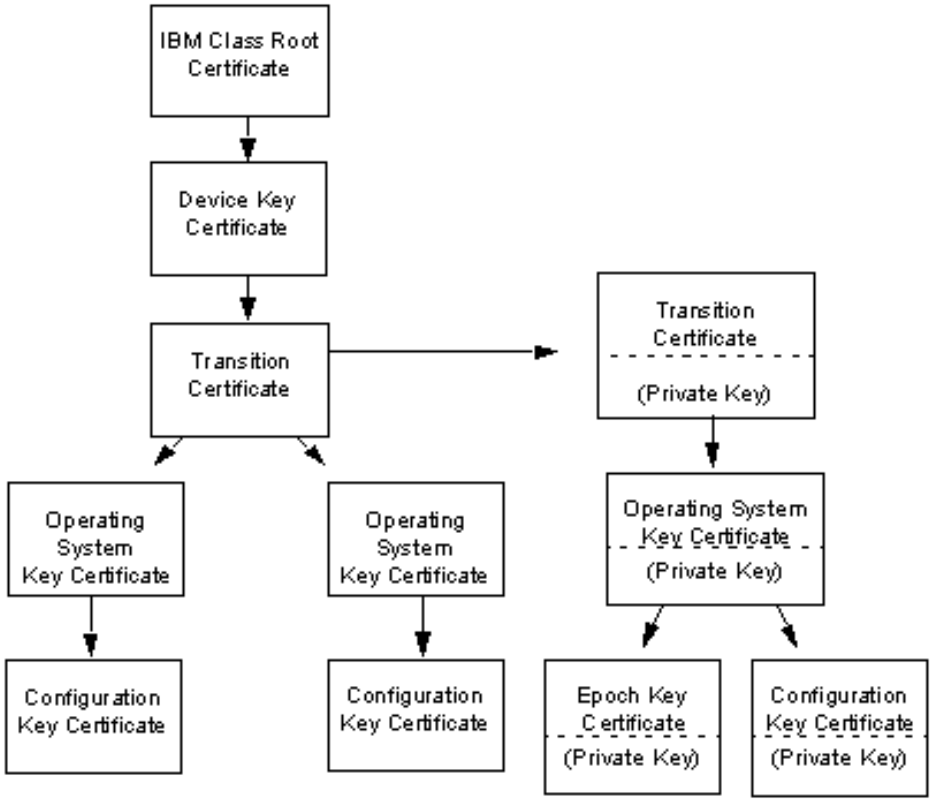

Figure 7 Configuration keypair and epoch keypair created

Figure 8 shows the changes to the certificate chain caused by loading an application from another vendor into segment 3. This operation perforce clears segment 3 BBRAM. This ends the current epoch, so all existing Operating System Certificates and any certificates created on behalf of the old application are deleted, as are any private keys that correspond to the public keys in those certificates. The start of a new epoch also marks the beginning of a new configuration, so the software in segment 1 creates a new Operating System Keypair and the corresponding certificate. Note that the current Operating System Certificate and private key are not the same as the current Operating System Certificate and private key in Figure 6 even though the two certificates have the same parent. The items shown in Figure 6 are deleted at the end of the epoch.

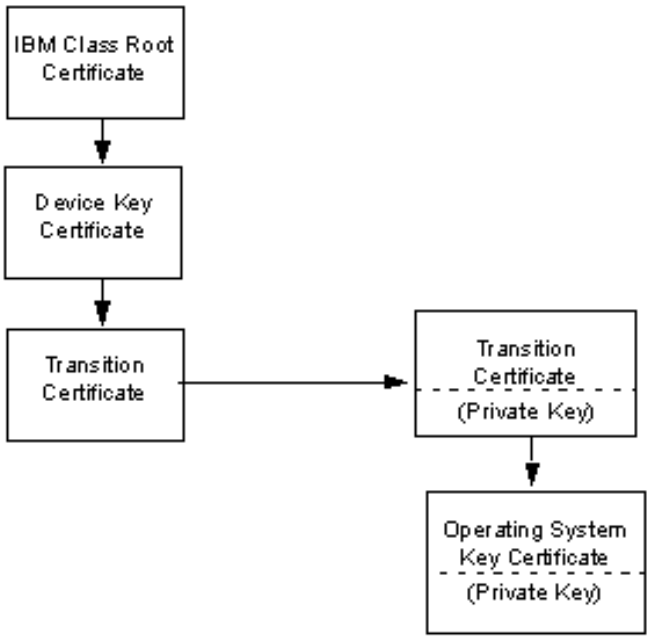

Figure 8 Foreign application loaded

3.13.3 OA certificates

The interface to the Outbound Authentication (OA) Driver defines the xcOA_CK0_Head_t and xcOA_CK0_Body_t types to hold information about an OA certificate. An OA certificate has a variable length and consists of two descriptive headers followed by a buffer containing the various elements of the certificate. Figure 9 shows the general structure of an OA certificate.

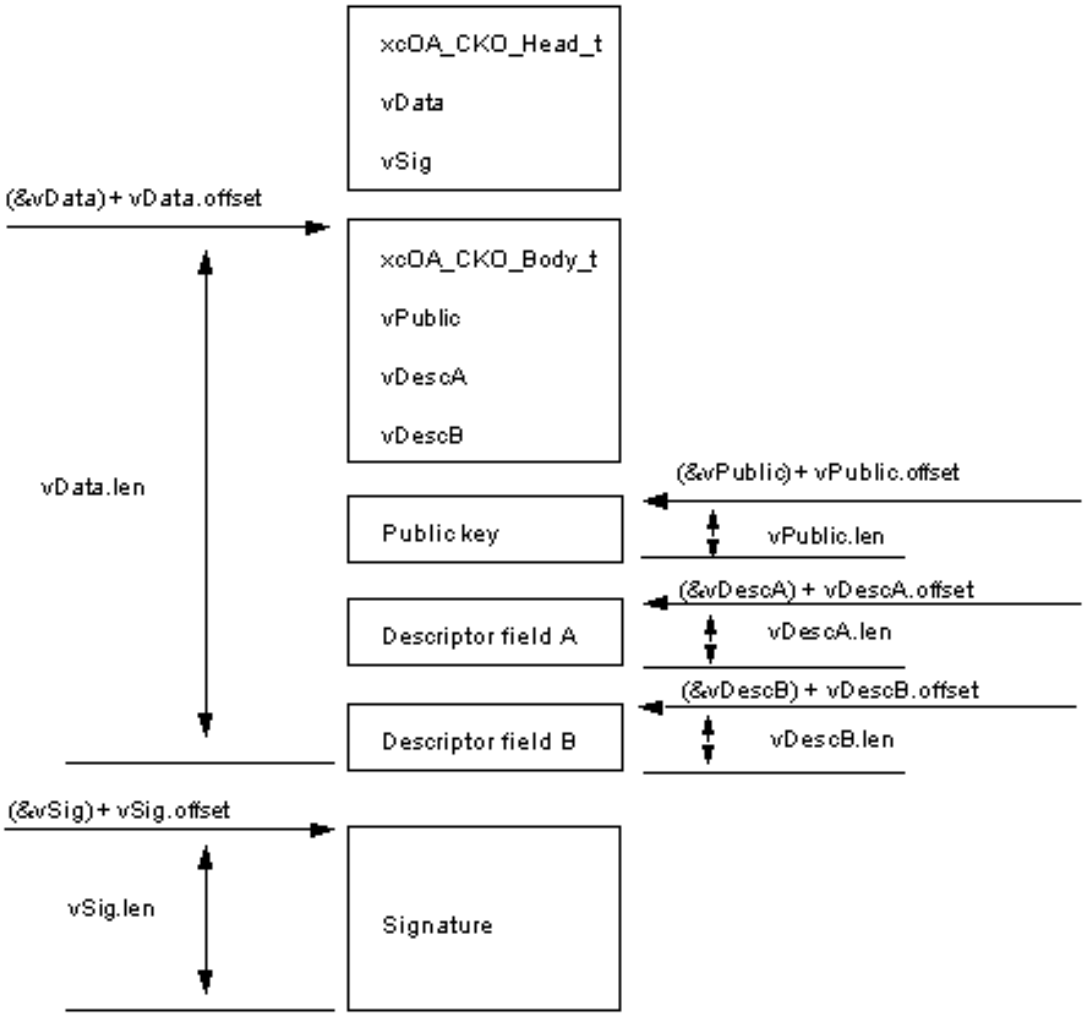

Figure 9 Structure of an OA certificate

For convenience, the following fields appear both in the *xcOA_CKO_Head_t* header and in the *xcOA_CKO_Body_t* header. The fields in the first header are easier to locate, but only the fields in the second header are part of the body of the certificate and hence covered by the cryptographic signature for the certificate. The following discussion describes the fields only once, with the understanding that they should have the same values in both headers.

- *cko_name*
- *cko_type*
- *parent_name*

Fields common to all certificates

The first descriptive header is an item of type *xcOA_CKO_Head_t*. Certain fields in this header are either

constant or interpreted in the same manner regardless of the type of certificate the header defines:

- *struct_id.name* is **XCOA_CKO_HEAD_T**.
- *struct_id.version* is the value to which **XCOA_CKO_HEAD_VER** is defined in the header file that defines the version of *xcOA_CKO_Head_t* that maps the header [17].
- *padbytes* is two bytes of zeros.
- *tData* is a data parsing indicator that defines how to parse the bytes in the *vData* data region and can have the following values:
  - **OA_NEW_CERT** identifies a *xcOA_CKO_Body_t*.
  - **OA_OLD_DEVICE_CERT** identifies an old format device certificate.
  - **OA_OLD_TRANS_CERT** identifies an old format transition certificate.
- *vData* specifies the offset and length of the body of the certificate:
  - *vData.offset* is the offset in bytes from the start of the *vData* field to the first byte of the body of the certificate, which begins with the second descriptive header (an item of type *xcOA_CKO_Body_t*).

    If *v* is an item of type *var_t*, the address of the item *v* describes is *((char)&(v))+v.offset*. By convention, if *v.offset* is zero the item *v* describes is empty or missing. Also by convention if *x* and *y* are *var_t* structures and y is part of the item *x* describes, the item *y* describes is also a part of the item *x* describes (that is, “nested” *var_t* structures describe nested items).
  - *vData.length* is the length in bytes of the body of the certificate [18].
- *vSig* specifies the offset and length of the cryptographic signature that covers the body of the certificate. The format of the signature depends on the value of the *tsig* field (see below). The fields of *vSig* are used in the same manner as the fields of *vData*.
- *tSig* specifies how the cryptographic signature that covers the body of the certificate is generated.

  If tSig is **ECC_COMPLIANT**, an ECC private key is used to generate the signature. The body of the certificate is processed as dictated by FIPS-186-3.

  A signature generated using a private key is stored as a simple (but potentially very large) binary integer. The block of data whose offset and length are specified in *vSig* contains the signature, which is stored in big-endian order: the byte at the lowest address is the most significant byte of the signature.
- *cko_status* is **OA_CKO_ACTIVE** if the coprocessor knows the value of the private key corresponding to the public key contained in the certificate and is **OA_CKO_INACTIVE** otherwise.
- *parent_name* is the name of the keypair whose private key is used to generate the signature that

17 Thus, for example, if the *struct_id.version* field in a structure of type *xcOATime_t* is not equal to **XCOATIME_VER**, the definition of *xcOATime_t* used to build the code that performs the comparison does not match the definition used to build the code that created the structure, and the code that performs the comparison must not attempt to parse the structure unless it has another way to know how the structure is mapped.

18 If *v* is an item of type *var_t*, the careful programmer will check that the region defined by *v.offset* and *v.length* is completely contained within the buffer or object that allegedly contains it.

covers the certificate and whose public key is used to verify the signature.

The contents of the remaining fields in the *xcOA_CKO_Head_t* header depend on which type of certificate the header defines.

The second descriptive header (which appears at the beginning of the body of the certificate) is an item of type *xcOA_CKO_Body_t*. Certain fields in this header are either constant or interpreted in the same manner regardless of the type of certificate the header defines:

- *struct_id.name* is **XCOA_CKO_BODY_T**.
- *struct_id.version* is the value to which **XCOA_CKO_BODY_VER** is defined in the header file that defines the version of *xcOA_CKO_Body_t* that maps the header. See “footnote 137” on page 126 for the description of *struct_id.version* for *xcOA_CKO_Head_t*.
- *padbytes* is two bytes of zeros.
- *tPublic* specifies which type of public key the certificate contains:

  If *tPublic* is **OA_RSA**, the public key is an RSA public key. The block of data whose offset and length are specified in *vPublic* begins with a structure of type *sccRSAKeyToken_t* that defines the elements of the public key (as described in “RSA key tokens” on page 81), which appear following the structure.

  If *tPublic* is **OA_ECC**, the public key is an ECC public key. The block of data whose offset and length are specified in *vPublic* begins with a structure of type xcECC*KeyToken_t*. This structure defines the elements of the public key (as described in “ key tokens” on page 93), which appear following the structure.
- *vPublic* specifies the offset and length of the public key the certificate contains. The fields of *vPublic* are used in the same manner as the fields of *xcOA_CKO_Head_t.vData*.
- *cko_name* identifies the keypair whose public key is contained in the certificate.
- *parent_name* identifies the keypair whose private key was used to create the cryptographic signature that covers the body of the certificate.

The contents of the remaining fields in the *xcOA_CKO_Body_t* header depend on which type of certificate the header defines.

The contents of the *cko_type*, *cko_name*, and *parent_name* fields in the *xcOA_CKO_Head_t* header are identical to the contents of the corresponding fields in the *xcOA_CKO_Body_t* header.

Keypair names in the two headers (*parent_name* and *cko_name*) are unique if the keypair is an IBM Root Keypair or an IBM Class Root Keypair. Keypairs of other types are generated by a coprocessor. Two keypairs generated by different coprocessors may have the same name but can be distinguished by using the *xcOA_CKO_Body_t.device_name* field. See “Keypair names” on page 130 for details.

IBM class root certificates

The type-dependent fields in the *xcOA_CKO_Head_t* and *xcOA_CKO_Body_t* headers for an IBM Class Root Certificate are set as follows:

- *cko_type* is **OA_CKO_IBM_ROOT**.
- *cko_name* names an IBM Class Root Keypair. See “Keypair names” on page 130 for details.
- *parent_name* names an IBM Root Keypair. See “Keypair names” on page 130 for details.
- *device_name* is undefined.

- *vDescA* specifies the offset and length of a timestamp that indicates when the IBM Class Root Keypair whose public key is contained in the certificate was created. See “Timestamps” on page 132 for details. The fields of *vDescA* are used in the same manner as the fields of *xcOA_CKO_Head_t.vData*.
- *vDescB* specifies the offset and length of a structure that describes the IBM Class Root Keypair whose public key is contained in the certificate. See “Class root descriptions” on page 133 for details. The fields of *vDescB* are used in the same manner as the fields of *xcOA_CKO_Head_t.vData*.

Device key certificates

The type-dependent fields in the *xcOA_CKO_Head_t* and *xcOA_CKO_Body_t* headers for a Device Key Certificate are set as follows:

- *cko_type* is **OA_CKO_MB**.
- *cko_name* names a coprocessor-generated keypair. See “Keypair names” on page 130 for details.
- *parent_name* names an IBM Class Root Keypair. See “Keypair names” on page 130 for details.
- *device_name* uniquely identifies the coprocessor that generated the keypair whose public key is contained in the certificate. See “Device names and device descriptors” on page 130 for details.
- *vDescA* specifies the offset and length of a description of the coprocessor that generated the keypair whose public key is contained in the certificate. See “Device names and device descriptors” on page 130 for details. The fields of *vDescA* are used in the same manner as the fields of *xcOA_CKO_Head_t.vData*.
- *vDescB* specifies the offset and length of a layer descriptor that describes the miniboot software (that is, the software in segment 1) that was present in the coprocessor identified by *device_name* when that coprocessor created the keypair whose public key is contained in the certificate. See “Layer names and layer descriptors” on page 131 for details. The fields of *vDescB* are used in the same manner as the fields of *xcOA_CKO_Head_t.vData*.

Transition certificates

The type-dependent fields in the *xcOA_CKO_Head_t* and *xcOA_CKO_Body_t* headers for a Transition Certificate are set as follows:

- *cko_type* is **OA_CKO_MB**.
- *cko_name* names a coprocessor-generated keypair. See “Keypair names” on page 130 for details.
- *parent_name* names a keypair whose public key is contained in a Device Key Certificate or in a Transition Certificate. See “Keypair names” on page 130 for details.
- *device_name* uniquely identifies the coprocessor that generated the keypair whose public key is contained in the certificate. See “Device names and device descriptors” on page 130 for details.
- *vDescA* specifies the offset and length of a description of the coprocessor that generated the keypair whose public key is contained in the certificate. See “Device names and device descriptors” on page 130 for details. The fields of *vDescA* are used in the same manner as the fields of *xcOA_CKO_Head_t.vData*.
- *vDescB* specifies the offset and length of a layer descriptor that describes the miniboot software

(that is, the software in segment 1) that was present in the coprocessor identified by *device_name* when that coprocessor created the keypair whose public key is contained in the certificate. See “Layer names and layer descriptors” on page 131 for details. The fields of *vDescB* are used in the same manner as the fields of *xcOA_CKO_Head_t.vData*.

Operating system key certificates

The type-dependent fields in the *xcOA_CKO_Head_t* and *xcOA_CKO_Body_t* headers for an Operating System Key Certificate are set as follows:

- *cko_type* is **OA_CKO_SEG2_SEG3**.
- *cko_name* names a coprocessor-generated keypair. See “Keypair names” on page 130 for details.
- *parent_name* names a keypair whose public key is contained in a Device Key Certificate or in a Transition Certificate. See “Keypair names” on page 130 for details.
- *device_name* uniquely identifies the coprocessor that generated the keypair whose public key is contained in the certificate. See “Device names and device descriptors” on page 130 for details.
- *vDescA* specifies the offset and length of a layer descriptor that describes the operating system (that is, the software in segment 2) that was present in the coprocessor identified by *device_name* when that coprocessor that generated the keypair whose public key is contained in the certificate. See “Layer names and layer descriptors” on page 131 for details. The fields of *vDescA* are used in the same manner as the fields of *xcOA_CKO_Head_t.vData*.
- *vDescB* specifies the offset and length of a layer descriptor that describes the the application (that is, the software in segment 3) that was present in the coprocessor identified by *device_name* when that coprocessor created the keypair whose public key is contained in the certificate. See “Layer names and layer descriptors” on page 131 for details. The fields of *vDescB* are used in the same manner as the fields of *xcOA_CKO_Head_t.vData*.

Application key certificates

The type-dependent fields in the *xcOA_CKO_Head_t* and *xcOA_CKO_Body_t* headers for an Application Key Certificate are set as follows:

- *cko_type* is **OA_CKO_SEG3_CONFIG** if the public key the certificate contains is part of a Configuration Keypair and is **OA_CKO_SEG3_EPOCH** if the public key is part of an Epoch Keypair.
- *cko_name* names a coprocessor-generated keypair. See “Keypair names” on page 130 for details.
- *parent_name* names a keypair whose public key is contained in an Operating System Key Certificate. See “Keypair names” on page 130 for details.
- *device_name* uniquely identifies the coprocessor that generated the keypair whose public key is contained in the certificate. See “Device names and device descriptors” on page 130 for details.
- *vDescA* is reserved.
- *vDescB* specifies the offset and length of a a block of data supplied by the application to be associated with the certificate when the keypair was created. The fields of *vDescB* are used in the same manner as the fields of *xcOA_CKO_Head_t.vData*.

Keypair names

The interface to the OA daemon defines the *xcOA_CKO_Name_t* type to hold the name of a keypair. The contents of the fields in a *xcOA_CKO_Name_t* structure depend on which type of keypair the structure names.

IBM root keypairs

The fields in a *xcOA_CKO_Name_t* structure that names an IBM Root Keypair are set as follows:

- *name_type* is **OA_IBM_ROOT**.
- *index* is an integer that distinguishes the IBM Root Keypair named by the structure from all other IBM Root Keypairs.
- *creation_boot* is not used.

IBM class root keypairs

The fields in a *xcOA_CKO_Name_t* structure that names an IBM Class Root Keypair are set as follows:

- *name_type* is **OA_IBM_CLASS_ROOT**.
- *index* is an integer that distinguishes the IBM Class Root Keypair named by the structure from all other IBM Class Root Keypairs.
- *creation_boot* is not used.

Coprocessor-generated keypairs

The fields in a *xcOA_CKO_Name_t* structure that names a keypair that was generated on a coprocessor (that is, any keypair except an IBM Root Keypair or an IBM Class Root Keypair) are set as follows:

- *name_type* is **OA_STANDARD_NAME**.
- *index* is an integer that distinguishes the keypair named by the structure from all other keypairs generated by the same coprocessor that have the same value for *creation_boot*.
- *creation_boot* is the value the boot counter on the coprocessor that generated the keypair that the structure names had when the keypair was generated. See “Coprocessor architecture” on page 114 for details.

Note that the names of two keypairs generated on a single coprocessor are distinct, but that the name a keypair generated on one coprocessor may match the name of a keypair generated on another coprocessor. In general, the device_name field in an OA certificate must be used to distinguish keys generated on one coprocessor from keys generated on another coprocessor.

Device names and device descriptors

The interface to the OA daemon defines the *xcOADeviceName_t* type to hold the name of a particular coprocessor. The fields in a *xcOADeviceName_t* structure are set as follows:

- *struct_id.name* is **XCOADEVICENAME_T**.
- *struct_id.version* is the value to which **XCOADEVICENAME_VER** is defined in the header file that defines the version of *xcOADeviceName_t* that maps the header. See “footnote 137” on page 126 for the description of *struct_id.version* for *xcOA_CKO_Head_t*.
- *padbytes* is two bytes of zeros.

- *adapterID* is a serial number that uniquely identifies the coprocessor. It matches the value of the AdapterID field returned by *xcGetConfig*. See “xcGetConfig - get coprocessor configuration” on page 110 for details.

when_certified is a timestamp that indicates when the Device Key Certificate was loaded into the coprocessor during manufacture. See “Epochs and configurations” on page 132 for details.

Layer names and layer descriptors

The interface to the OA daemon defines the *xcOALayerName_t* type to hold an identifier that uniquely identifies the software loaded into a particular segment of a particular coprocessor. The fields of a layer name are set as follows:

- *struct_id.name* is **XCOALAYERNAME_T**.
- *struct_id.version* is the value to which **XCOALAYERNAME_VER** is defined in the header file that defines the version of *xcOALayerName_t* that maps the timestamp. See “footnote 137” on page 126 for for the description of *struct_id.version* for *xcOA_CKO_Head_t*.
- *padbytes* is two bytes of zeros.
- *epoch_start* marks the beginning of the epoch in which the software that the structure names was loaded. In particular, *epoch_start* is the value of the boot counter on the coprocessor into which the software that the structure names was loaded at the point the epoch in which the software that the structure names was loaded began. See “Epochs and configurations” on page 132 and “Overview of the authentication scheme” on page 115 for details.
- *config_start* marks the start of the configuration that includes the software that the structure names. In particular, *config_start* is the value the boot counter on the coprocessor into which the software the structure names was loaded at the point the software that the structure names was loaded. See “Overview of the authentication scheme” on page 115 for details.
- *config_count* specifies how many configurations there have been during the epoch whose beginning *epoch_start* defines. This includes the configuration that began when the software the structure names was loaded. See “Overview of the authentication scheme” on page 115 for details.

The interface to the OA daemon defines the *xcOALayerDesc_t* type to hold a description of the software loaded into a particular segment of a particular coprocessor. The fields of a layer description are set as follows:

- *struct_id.name* is **XCOALAYERDESC_T**.
- *struct_id.version* is the value to which **XCOALAYERDESC_VER** is defined in the header file that defines the version of *xcOALayerDesc_t* that maps the layer description. See “footnote 137” on page 126 for for the description of *struct_id.version* for *xcOA_CKO_Head_t*.
- *padbyte* is one byte of zeros.
- *layer_number* is 1 if the software the structure describes is loaded into segment 1, 2 if the software is loaded into segment 2, and 3 if the software is loaded into segment 3.
- *ownerID* is the owner identifier associated with the segment into which the software is loaded. See “Overview of the authentication scheme” on page 115 for details.
- *image_name* is the name associated with the software. See “Overview of the authentication scheme” on page 115 for details.

- *image_revision* is the revision number associated with the software. See “Overview of the authentication scheme” on page 115 for details.
- *image_hash* is the SHA-256 hash of the software. See “Overview of the authentication scheme” on page 115 for details.
- *layer_name* uniquely identifies the software.

Epochs and configurations

Epochs and configurations are measured with respect to a particular segment. Thus, the values of the recorded boot counter values in a layer 2 descriptor in an Operating System Certificate may differ from the corresponding values in the layer 3 descriptor in the same certificate. Consider the following sequence of operations:

1. The operating system is loaded into an empty coprocessor when the boot counter is 0x60c. This begins a new segment 2 epoch and a new segment 2 configuration. The segment 2 configuration count is initialized to 1.
2. An application is loaded into segment 3 for the first time when the boot counter is 0x60d. This begins a new segment 3 epoch and a new segment 3 configuration. The segment 3 configuration count is initialized to 1.
3. A newer version of the operating system is loaded into the coprocessor when the boot counter is 0x612. This begins a new segment 2 configuration and a new segment 3 configuration. Both configuration counts are incremented.
4. A new application is loaded into the coprocessor when the boot counter is 0x620. This begins a new segment 3 configuration and the segment 3 configuration count is incremented.

The Operating System Certificate created during step 4 will have a layer descriptor for segment 2 whose fields have the following values:

- *epoch_start* = 0x60c
- *config_start* = 0x612
- *config_count* = 2

and a layer descriptor for segment 3 whose fields have the following values:

- *epoch_start* = 0x60d
- *config_start* = 0x620
- *config_count* = 3

Timestamps

The interface to the OA daemon defines the *xcOATime_t* type to hold a timestamp. The fields of a timestamp are set as follows:

- *struct_id.name* is **xcOATIME_T**.
- *struct_id.version* is the value to which **XCOATIME_VER** is defined in the header file that defines the version of *xcOATime_t* that maps the timestamp. See “footnote 137” on page 126 for the description of struct_id.version for *xcOA_CKO_Head_t*.
- *year* is a BCD representation of the year (for example, 0x2000 represents the year 2000).

- *month* is a BCD representation of the month (for example, 0x12 represents December).
- *day* is a BCD representation of the day of the month (for example, 0x10 represents the 10th).
- *hour* is a BCD representation of the hour using a 24-hour clock (for example, 0x17 represents 5 p.m.).
- *minute* is a BCD representation of the minute (for example, 0x25 represents 25 minutes past the hour).

Timestamps created on a coprocessor are set to the date and time provided by the coprocessor's real-time clock, which should be synchronized with an external clock if an accurate timestamp is required.

Class root descriptions

The interface to the OA daemon defines the *xcOA_CKO_Descr_t* type to hold the description of an IBM Class Root Keypair. The fields in the *xcOA_CKO_Descr_t* structure are set as follows:

- *struct_id.name* is **XCOA_CKO_DESCR_T**.
- *struct_id.version* is the value to which **XCOA_CKO_DESCR_VER** is defined in the header file that defines the version of *xcOA_CKO_Descr_t* that maps the description. See “footnote 137” on page 126 for the description of *struct_id.version* for *xcOA_CKO_Head_t*.
- *cert_qualifier* is an integer that identifies the keypair. See the **OA_CLASS_ROOT_*** constants in *xc_oa_mb.h*.
- *descr* is a text description of the keypair.

3.13.4 xcOA – Request an OA Operation

xcOA will perform a request defined by the option provided. Optionally, it will check the status of the coprocessor, check the directory structure, return a certificate, generate a new key and certificate of the key, or use an existing key/certificate in an operation.

Function prototype

```
unsigned int xcOA( xcOA_RB_t *pxcOA_rb );
```

Input

On entry to this routine:

pxcOA_rb is a pointer to an item of type whose fields are initialized as described below:

- *cmn* is set as described in Common structure on page 21.
- *DRBG_handle* is the handle of the DRBG instance associated with this thread/fiber.
- *options* contains one of the following:

OA_OPT_RESTART	Starts or restarts the OA system, initializing the global variables to hold the certificate chains.
OA_OPT_RESET	Recopies the certificate chains and deletes the previous chains. This function will change the random number generator instance for the OA system.
OA_OPT_GENERATE	generates a new Application Keypair and an OA certificate containing the public half of the keypair. The certificate is signed using the private half of the current Operating System keypair.
OA_OPT_PRIVOP	directs the OA Driver to perform a cryptographic operation with an Application Key. The private key can be used to decrypt or sign a block of data, and the public key can be used to encrypt a block of data or verify a cryptographic signature.
OA_OPT_GETDIR	determines the total number of OA Certificates the OA daemon has saved. Information about the certificates can also be retrieved. The function will optionally return a directory of the certificates.
OA_OPT_GETCERT	either returns the length of an OA certificate in the certificate list or retrieves the certificate itself
OA_OPT_DELETE	deletes an Application Keypair and the OA certificate that contains the keypair's public key.
OA_OPT_STATUS	either returns information about the status of the coprocessor and the software (if any) that is loaded into each segment or returns the amount of space this status information would occupy.

The contents of the remaining fields of the *pxcOA_rb* depend on the *options* field.

For *pxcOA_rb->options* = ***OA_OPT_GETDIR***

- *pCount* points to a writeable buffer in which an item of type unsigned long can be stored.
- *pBuffer* either must be NULL or must point to a writeable buffer.
- *pLen* points to a writeable buffer. If pBuffer is not NULL, **pLen* is the length in bytes of the buffer referenced by *pBuffer*.

For *pxcOA_rb->options* = ***OA_OPT_GETCERT***

- **pCKO_name* is the name of the keypair whose public key is contained in the OA certificate of interest. See “Keypair names” on page 130 for details.
- *pBuffer* either must be NULL or must point to a writeable buffer.
- *pLen* points to a writeable buffer. If *pBuffer* is not NULL, **pLen i*s the length in bytes of the buffer referenced by *pBuffer.*

For *pxcOA_rb->options* = ***OA_OPT_GENERATE***

- *pOAGenRB points to an OA Generate request block whose fields are initialized as follows:*
  - *struct_id.name* is **XCOAGEN_RB_T**.
  - *struct_id.version* is the value to which **XCOAGEN_RB_VER** is defined in the header file that defines the version of *xcOAGen_RB_t* that maps the request block. See “footnote 137” on page 126 for the description of *struct_id*.version for *xcOA_CKO_Head_t*.
  - *padbytes* is two bytes of zeros.
  - *algorithm* specifies the crypto system used to generate the keypair and must be **OA_ECC** (to generate an ECC keypair).
  - *cko_type* specifies what kind of Application Keypair is generated and must be either **OA_CKO_SEG3_CONFIG** (to generate a configuration key) or **OA_CKO_SEG3_EPOCH** (to generate an epoch key).
  - *vSeg3Field* specifies the offset and length of a block of data to be stored in the new OA Certificate. The block is copied to the body of the certificate and the certificate's vDescB field describes the block's location and length.
    - *vSeg3Field*.offset is the offset in bytes from the start of the *vSeg3Field* field to the first byte of the block of data.

      If *v* is an item of type *var_t*, the address of the item *v* describes is *((char)&(v)) +v.offset*. By convention, if *v.offset* is zero the item *v* describes is empty or missing. Also by convention if *x* and *y* are *var_t* structures and *y* is part of the item *x* describes, the item *y* describes is also a part of the item *x* describes (that is, “nested” *var_t* structures describe nested items).
    - *vSeg3Field.length* is the length in bytes of the block of data [19].
  - *pCKO_name* points to a writeable buffer in which an item of type *xcOA_CKO_Name_t* can be stored.

19 If *v* is an item of type *var_t*, the careful programmer will check that the region defined by *v.offset* and *v.length* is completely contained within the buffer or object that allegedly contains it.

- If *pOAGenRB->algorithm* is **OA_ECC**, *lOAeccRB* is the length of the buffer at *pOAeccRB,* and **pOAeccRB* must be an ECC key generate request block (an item of type *xcECCKeyGen_RB_t*) whose *curve_type, curve_size, and key_token* fields are initialized as required by *xcECCKeyGenerate* (see “xcECCKeyGenerate - generate ECC keypair” on page 94 for details). The remaining fields in **pOAeccRB* are ignored.

For *pxcOA_rb->options* = ***OA_OPT_DELETE***

- **pCKO_name* is the name of the keypair to delete. *pCKO_name* must identify an Application Keypair. See “Keypair names” on page 130 for details.

For *pxcOA_rb->options* = ***OA_OPT_PRIVOP***

- **pCKOName* is the name of the keypair to be used in the cryptographic operation. **pCKOName* must identify an Application Keypair. See “Keypair names” on page 130 for details.
- *pPKPrivRB* points to a public key algorithm operation request block:
- The keypair identified by **pCKOName* must be an ECC keypair.
- **pOAeccRB* must be an ECC operation request block (an item of type *xcECC_RB_t*) whose *options*, *key_token, data_in, data_out*, *data_in_size*, *data_out_size*, and *key_size* fields are initialized as required by *xcECC* (see “xcECC - sign data or verify signature for data” on page 96 for details).
- The options field of the request block determines whether the cryptographic operation is performed using the public half of the keypair or the private half. The request block must conform to the key used in the operation as required by *xcECC*.
- *lPKPrivRB is the length in bytes of the request block referenced by pPKPrivRB.*

For *pxcOA_rb->options* = ***OA_OPT_STATUS***

- *pBuffer* may be NULL. If it is not NULL, it points to a writeable buffer.
- *pLen* points to a writeable buffer in which an item of type unsigned long can be stored. If *pBuffer* is not NULL, **pLen* is the length in bytes of the buffer referenced by *pBuffer.*

For *pxcOA_rb->options* = *OA_OPT_RESET*

- *No fields are required to be set on input to this function.*

For *pxcOA_rb->options* = *OA_OPT_RESTART*

- *DRBG_handle* is the handle of a valid random number generator instance.

Output

On successful exit from this routine:

For *pxcOA_rb->options* = ***OA_OPT_GETDIR***

- **pCount* is the number of OA certificates in the certificate list.
- If *pBuffer* is not NULL, the buffer it references contains an array of items of type *xcOA_DirItem_t* and **pLen* is the length in bytes of this array. The fields of the ith entry in the array are set as follows:
    - *struct_id.name* is **xcOA_DIRITEM_T**.

- *struct_id.version* is the value to which **XCOA_DIRITEM_VER** is defined in the header file that defines the version of *xcOA_DirItem_t* that maps the entry. See “footnote 137” on page 126 for the description of struct_id.version for *xcOA_CKO_Head_t*.
- *padbytes* is two bytes of zeros.
- *cko_name* identifies the keypair whose public key is contained in the OA Certificate the entry describes. See “Keypair names” on page 130 for details [20].
- *cko_type* specifies the keypair's type. *cko_type* is used to ensure the certificate is used appropriately.
- *algorithm* is **OA_RSA** if the keypair identified by cko_name is an RSA keypair and is **OA_ECC** if the keypair is an ECC keypair.
- *cko_status* is **OA_CKO_ACTIVE** if the private key in the keypair identified by cko_name exists and is **OA_CKO_INACTIVE** if the private key does not exist (for example, because the software configuration has changed since the keypair was created). The OA manager deletes the private key section and changes *cko_status* to **OA_CKO_INACTIVE** when the OA certificate is deactivated.
- *length* is the length in bytes of the OA Certificate the entry describes (that is, the minimum size of a buffer that could hold the OA Certificate).
- *parent_index* is the index within the array referenced by *pBuffer* of the entry that describes the OA Certificate that contains the public key corresponding to the private key that was used to create the cryptographic signature that covers the body of the OA Certificate the i th entry describes. If *parent_index* is negative, there is no such entry in the array referenced by pBuffer (for example, because the certificate is for an IBM Class Root key).
- If *pBuffer* is NULL, **pLen* is the length in bytes of a buffer that is just large enough to hold an array of items of type *xcOA_DirItem_t* that contains an entry for each OA certificate tin the certificate list.

For *pxcOA_rb->options* = ***OA_OPT_GETCERT***

- **pLen* is the length in bytes of the OA certificate of interest.
- *If pBuffer is not NULL, the buffer it references contains a copy of the desired OA certificate. See “OA certificates” on page 124 for details.*

For *pxcOA_rb->options* = ***OA_OPT_GENERATE***

- **(pOAGenRB->pCKO_name) identifies the newly generated Application Keypair. See “Keypair names” on page 130 for details.*

For *pxcOA_rb->options* = ***OA_OPT_DELETE***

- On successful exit from this routine, the keypair identified by **pCKO_name* and the OA Certificate that contains the keypair's public key have been deleted.

For *pxcOA_rb->options* = ***OA_OPT_PRIVOP***

- If the keypair identified by **pCKOName* is an ECC keypair and *((xcECC_RB_t *)pOAeccRB)-*

20 Since the keypair in question was perforce generated on the coprocessor on which the application that calls *xcOAGetDir* is running, *cko_name* is unambiguous regardless of what kind of keypair it names. There is no need for a Device Name.

>*options* specifies **ECC_SIGN**, *(((xcECC_RB_t *)pOAeccRB)->sig_token)* contains the digital signature produced by signing *(((xcECC_RB_t *)pOAeccRB)->data)* with the private half of the keypair identified by **pCKOName* .

- If the keypair identified by **pCKOName* is an ECC keypair and *((xcECC_RB_t *)pOAeccRB)->options* specifies **ECC_VERIFY***,* a return code of zero implies that the signature in *(((xcECC_RB_t *)pOAeccRB)->sig_token)* was produced by signing *(((xcECC_RB_t *)pOAeccRB)->data)* with the private half of the keypair identified by **pCKOName.*

For *pxcOA_rb->options* = ***OA_OPT_STATUS***

- **pLen* is the length in bytes of the status information. If *pBuffer* is not NULL, the buffer it references contains a structure of type *xcOAStatus_t* whose fields are set as follows:
  - *struct_id.name* is **XCOASTATUS_T**.
  - *struct_id.version* is the value to which **XCOASTATUS_VER** is defined in the header file that defines the version of *xcOAStatus_t* that maps the entry. See “footnote 137” on page 126 for the description of *struct_id.version* for *xcOA_CKO_Head_t*.
  - *padbytes* is two bytes of zeros.
  - *rom_status* contains information about the basic health of the coprocessor and the state of each segment. The fields of *rom_status* are set as follows:
    - *struct_id.name* and *struct_id.version* are zero.
    - *rsvd1* is not used.
    - *rom_version* is a version number stored in the coprocessor's ROM. This number matches the value of "ROM ver" reported by the CLU ST command.
    - *page1_certified* is nonzero, indicating that the coprocessor possesses a Device Keypair and an OA certificate for the keypair signed by the appropriate IBM Class Root private key.
    - *rsvd2* is not used.
    - *boot_count_right* is the current value of the coprocessor's boot counter. See “Coprocessor architecture” on page 114 for details.
    - *adapterID* is a serial number that uniquely identifies the coprocessor. It matches the value of the *AdapterID* field returned by *xcGetConfig* . See “xcGetConfig - get coprocessor configuration” on page 110 for details.
  - *vpd* is a description of the coprocessor. The first 128 bytes match the value of the AMCC_EEPROM, HdwRigolettoID, HdwOtelloECID, and EthernetMAC fields returned by xcGetConfig. The next 128 bytes match the value of the VPD field returned by *xcGetConfig*. See “xcGetConfig - get coprocessor configuration ” on page 110for details.
    - *init_state* is 1.
    - *seg2_state* and *seg3_state* indicate the status of segment 2 and segment 3, respectively. Possible values are:
      0 UNOWNED
      1 OWNED_BUT_UNRELIABLE
      2 RUNNABLE
      3 RUNNABLE_BUT_UNRELIABLE
      Refer to Chapters 2 and 5 of the *IBM PCIe Cryptographic Coprocessor Custom*

Software Developer's Toolkit Guide for details.

- *owner2* and *owner3* are the owner identifiers associated with segment 2 and segment 3, respectively. An owner identifier is undefined if the corresponding segment is **UNOWNED**. Refer to Chapters 2 and 5 of the *IBM PCIe Cryptographic Coprocessor Custom Software Developer's Toolkit Guide* for details.
- *active_seg1* indicates which half of the memory dedicated to segment 1 will be overwritten the next time the software in segment 1 is reloaded. (This scheme permits segment 1 to be reloaded in an atomic fashion.)
- *rsvd3* is not used.
- *usr* is the number of times segment 3 has been updated since the last coprocessor reset.
- *vSeg_ids* is undefined.
- *free_space* indicates the amount of free code and system space in each segment. This is the total size in bytes of the segment minus the size in bytes of the code, public key, and other information that the system software in segment 1 has saved in the segment. The first entry in the array (that is, *free_space[0]*) specifies the amount of free space in segment 1, the second entry in the array specifies the amount of free space in segment 2, and the third entry in the array specifies the amount of free space in segment 3.
- *layer_name* is an array of identifiers that uniquely identify the software loaded into each segment of the coprocessor. See “Layer names and layer descriptors” on page 131 for details.
- The first entry in the array (that is, *layer_name[0]*) identifies the software in segment 1, the second entry in the array identifies the software in segment 2, and the third entry in the array identifies the software in segment 3.
- *device_name* identifies the coprocessor. See “Device names and device descriptors” on page 130 for details.

For *pxcOA_rb->options* = *OA_OPT_RESET*

- The OA certificate chains have been reset.

For *pxcOA_rb->options* = *OA_OPT_RESTART*

- The OA certificate system has been initialized.

Notes

Signature on new OA certificate

The cryptographic signature for the OA certificate generated by **OA_OPT_GENERATE** is created using the private key from the current Operating System Keypair (that is, the private key corresponding to the public key contained in the unique OA certificate whose *cko_type* field is **OA_CKO_SEG2_SEG3** and whose *cko_status* field is **CKO_ACTIVE**).

Return codes

Common return codes generated by this routine are:

OAGood (that is, 0)	The operation was successful.
OABadParm	An argument is not valid.
OANotAllowed	**pCKOName* does not identify an Application Keypair.
OANotFound	**pCKOName* does not identify any keypair.
OANoSpace	The operation failed due to lack of space.
PKAEccVerifyFail	The keypair identified by **pCKOName* is an ECC keypair and an ECC verify operation was requested but the signature failed to verify.

Refer to *xc_err.h* for a comprehensive list of return codes.

4 Error code formatting

Return codes for function calls follow the normal format:

oxWXYYzzzz

where:

W	Eight indicates a negative number; an error has occurred
X	Used by the error-generating module; usually zero
YY	Code number of the error-generating module
zzzz	Actual error code determined by the entity detecting the error

Common code combinations for WXYY:

Error Codes

8040	Host Device Driver
8140	Host Operating System
8207	File System
8240	POST Error
8340	MiniBoot 0
8440	MiniBoot 1

xC Error Codes

8041	xC Support Manager
8042	Comm Driver
8044	DES Driver
8045	PKA Driver
8046	RNG Driver
8048	OA Driver

Reserved for IBM Use

806x	CCA modules

Programmer Defined

8X8x	Used by applications

Note: A return code of zero indicates a successful operation.

5 DES weak, semi-weak, and possibly weak keys

xcRandomNumberGenerate will not return any of the 64-bit numbers in the following list if the options argument specifies **RANDOM_NOT_WEAK**.

```
01010101 01010101
01011F1F 01010E0E
0101E0E0 0101F1F1
0101FEFE 0101FEFE
011F0F1F 010E010E
011F1F01 010E0E01
011FE0FE 010EF1FE
011FFEE0 010EFEF1
01E001E0 01F101F1
01E01FFE 01F10EFE
01E0E001 01F1F101
01E0FE1F 01F1FE0E
01FE01FE 01FE01FE
01FE1FE0 01FE0EF1
01FEE01F 01FEF10E
01FEFE01 01FEFE01
1F01011F 0E01010E
1F011F01 0E010E01
1F01E0FE 0E01F1FE
1F01FEE0 0E01FEF1
1F1F0101 0E0E0101
1F1F1F1F 0E0E0E0E
1F1FE0E0 0E0EF1F1
1F1FFEFE 0E0EFEFE
1FE001FE 0EF101FE
1FE01FE0 0EF10EF1
1FE0E01F 0EF1F10E
1FE0FE01 0EF1FE01
1FFE01E0 0EFE01F1
1FFE1FFE 0EFE0EFE
1FFEE001 0EFEF001
1FFEFE1F 0EFEFE0E
E00101E0 F10101F1
```

```
E0011FFE F1010EFE
E001E001 F101F101
E001FE1F F101FE0E
E01F01FE F10E01FE
E01F1FE0 F10E0EF1
E01FE01F F10EF10E
E01FFE01 F10EFE01
E0E00101 F1F10101
E0E01F1F F1F10E0E
E0E0E0E0 F1F1F1F1
E0E0FEFE F1F1FEFE
E0FE011F F1FE010E
E0FE1F01 F1FE0E01
E0FEE0FE F1FEF1FE
E0FEFEE0 F1FEFEF1
FE0101FE FE0101FE
FE011FE0 FE010EF1
FE01E01F FE01F10E
FE01FE01 FE01FE01
FE1F01E0 FE0E01F1
FE1F1FFE FE0E0EFE
FE1FE001 FE0EF101
FE1FFE1F FE0EFE0E
FEE0011F FEF1010E
FEE01F01 FEF10E01
FEE0E0FE FEF1F1FE
FEE0FEE0 FEF1FEF1
FEFE0101 FEFE0101
FEFE1F1F FEFE0E0E
FEFEE0E0 FEFEF1F1
FEFEFEFE FEFEFEFE
```

6 IBM root public key

As of the date of this document, the key IBM uses to sign the certificates for the class keys used with the IBM 4767 PCIe Cryptographic Coprocessor is s 521-bit Prime ECC key whose public key Q is as follows:

```
04       (compression indicator)
01cf9238 348503b5 9d5fe207 467dda6e d13e6593 be423765 23bf2cd6 5fc37729
66fd90f2 10d4b960 46927418 037c8534 0d6e98d9 7551656e 89f8650f 9dec54d5
7c2d00dd fc3d7776 1bd913c9 16553453 4904c368 35dcdada 6dd8c8fd 6a226e7e
e36398df 81ec4c53 4996993e bd6ef421 410efabb 09286bb2 212cead6 2cfa1d0e
5d6f26e7
```

The most significant bit of the public key is 0x01 and the least significant is 0xe7.

7 Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectable rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, 10594, USA.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

7.1 Copying and distributing softcopy files

For online versions of this book, we authorize you to:

- Copy, modify, and print the documentation contained on the media, for use within your enterprise, provided you reproduce the copyright notice, all warning statements, and other required statements on each copy or partial copy.
- Transfer the original unaltered copy of the documentation when you transfer the related IBM product (which may be either machines you own, or programs, if the program's license terms permit a transfer). You must, at the same time, destroy all other copies of the documentation.

You are responsible for payment of any taxes, including personal property taxes, resulting from this authorization.

THERE ARE NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING THE WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Some jurisdictions do not allow the exclusion of implied warranties, so the above exclusion may not apply to you.

Your failure to comply with the terms above terminates this authorization. Upon termination, you must destroy your machine readable documentation.

7.2 Trademarks

IBM and AIX are trademarks of the IBM Corporation in the United States or other countries or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Windows is a registered trademark of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

8 List of abbreviations

ACP	Access Control Point
AES	Advanced Encryption Standard
ANSI	American National Standards Institute
API	Application program interface
ASCII	American National Standard Code for Information Exchange
BBRAM	battery-backed random access memory
CCA	Common Cryptographic Architecture
CMAC	Cipher-based Message Authentication Code (as specified in NIST SP 800-38B [8])
CLU	Coprocessor Load Utility
CPRB	Cooperative Processing Request Block
DES	Data Encryption Standard
ECC	Elliptic Curve Cryptography (algorithm)
FIPS	Federal Information Processing Standard
FPE	Format Preserving Encryption
IBM	International Business Machines Corporation
KEK	key encrypting key
MAC	message authentication code
PCI	peripheral component interconnect
PCIe	Peripheral component interconnect express
PCI-X	peripheral component interconnect extended
PIN	personal identification number
PKA	public key algorithm
RAM	random access memory
RNG	random number generator
RSA	Rivest-Shamir-Adleman (algorithm)
SHA	Secure Hash Algorithm
SRDI	Security Relevant Data Item
UDX	user-defined extension
VFPE	VISA Format Preserving Encryption
VPD	vital product data

9 Glossary

A

access control. Ensuring that the resources of a computer system can be accessed only by authorized users and in authorized ways.

access control point (ACP). A command that ensures that a certain resource of the cryptographic adapter can be accessed properly.

access method. A technique for moving data between main storage and input/output devices.

adapter. An electronic circuit board (expansion card) that a user can plug into an expansion slot to add memory or special features to a computer.

agent. (1) An application that runs within the IBM 4767 PCIe Cryptographic Coprocessor. (2) Synonym for *secure cryptographic coprocessor application*.

application program interface (API). A functional interface supplied by the operating system, or by a separate program, that allows an application program written in a high-level language to use specific data or functions of the operating system or that separate program.

authentication. In computer security, a process used to verify the user of an information system or protected resource.

authorize. In computer security, to permit a user to communicate with or make use of an object, resource, or function.

B

battery-backed random access memory (BBRAM). Random access memory that uses battery power to retain data while the system is powered off. The IBM 4767 PCIe Cryptographic Coprocessor uses BBRAM to store persistent data for IBM 4767 applications, as well as the coprocessor device key.

C

cipher block chain (CBC). A mode of operation that cryptographically connects one block of ciphertext to the next plaintext block.

ciphertext. Data that has been altered by any cryptographic process.

cleartext. Data that has not been altered by any cryptographic process.

Common Cryptographic Architecture (CCA). A comprehensive set of cryptographic services that furnishes a consistent approach to cryptography on major IBM computing platforms. Application programs can access these services through the CCA application program interface.

Common Cryptographic Architecture (CCA) API. The application program interface used to call CCA functions. The CCA API is described in the IBM 4767 CCA Basic Services Reference and Guide.

coprocessor. (1) A supplementary processor that performs operations in conjunction with another processor. (2) A microprocessor on an adapter that extends the address range of the processor in the host system, or adds specialized instructions to handle a particular category of operations; for example, an I/O coprocessor, math coprocessor, or a network coprocessor.

Coprocessor Load Utility (CLU). A program used to load validated code into the IBM 4767 PCIe Cryptographic Coprocessor.

Cryptographic Coprocessor (IBM 4767). An adapter that provides a comprehensive set of cryptographic functions to a workstation.

cryptographic node. A coprocessor that provides cryptographic services such as key generation and digital signature support.

cryptography. (1) The transformation of data to conceal its meaning. (2) In computer security, the principles, means, and methods used to so transform data.

D

data-encrypting key. (1) A key used to encipher, decipher, or authenticate data. (2) Contrast with key-encrypting key.

decipher. (1) To convert enciphered data into clear data. (2) Contrast with encipher.

Data Encryption Standard (DES). The NIST Data Encryption Standard, adopted by the U.S. Government as FIPS Publication 46, which allows only hardware implementation of the data encryption algorithm.

device driver. (1) A file that contains the code needed to use an attached device. (2) A program that enables a computer to communicate with a specific peripheral device; for example, a printer, videodisc player, or a CD drive.

E

encipher. (1) To scramble data or convert it to a secret code that masks its meaning. (2) Contrast with decipher.

enciphered data. (1) Data whose meaning is concealed from unauthorized users or observers. (2) See also ciphertext.

F

feature. A part of an IBM product that can be ordered separately from the essential components of the product.

flash memory. A specialized version of erasable programmable read-only memory (EPROM) commonly used to store code in small computers.

H

host. As regards to the IBM 4767 PCIe Cryptographic Coprocessor, the workstation into which the coprocessor is installed.

I

Interactive Code Analysis Tool (ICAT). A remote debugger used to debug applications running within the IBM 4767 PCIe Cryptographic Coprocessor.

intrusion latch. A software-monitored bit that can be triggered by an external switch connected to a jumper on the IBM 4767 PCIe Cryptographic Coprocessor. This latch can be used, for example, to detect when the cover of the coprocessor host workstation has been opened. The intrusion latch does not trigger the destruction of data stored within the coprocessor.

J

jumper. A wire that joins two unconnected circuits.

K

key. In computer security, a sequence of symbols used with an algorithm to encipher or decipher data.

key-encrypting key. (1) A key used to encipher, decipher, or authenticate another key. (2) Contrast with data-encrypting key.

M

master key. In computer security, the top-level key in a hierarchy of key-encrypting keys (KEKs).

message authentication code (MAC). In computer security, (1) a number of value derived by processing data with an authentication algorithm. (2) The cryptographic result of block cipher operations, on text or data, using the cipher block chain (CBC) mode of operation.

miniboot. Software within the IBM 4767 PCIe Cryptographic Coprocessor designed to initialize the operating system and to control updates to flash memory.

P

passphrase. In computer security, a string of characters known to the computer system and to a user; the user must specify it to gain full or limited access to the system and to the data stored therein.

Peripheral Component Interconnect (PCI). A 32-bit parallel computer expansion card standard.

Peripheral Component Interconnect Express (PCIe). A high-speed serial connection computer expansion card standard that replaces the PCI and PCI-X standards, utilized in the IBM 4765 and 4767 Cryptographic Adapter.

Peripheral Component Interconnect eXtended (PCI-x). A 64-bit version of the PCI, utilized in the IBM 4764 Cryptographic Adapter.

private key. (1) In computer security, a key that is known only to the owner and used with a public key algorithm to decipher data. Data is enciphered using the related public key. (2) Contrast with public key.

public key. (1) In computer security, a key that is widely known and used with a public key algorithm to encipher data. The enciphered data can be deciphered only with the related private key. (2) Contrast with private key.

public key algorithm (PKA). (1) In computer security, an asymmetric cryptographic process that uses a public key to encipher data and a related private key to decipher data. (2) See also *RSA algorithm*.

R

RSA algorithm. A public key encryption algorithm developed by R. Rivest, A. Shamir, and L. Adleman.

S

Security Relevant Data Item (SRDI). Data that is securely stored by the IBM 4767 Cryptographic Adapter.

V

verb. A function possessing an entry_point_name and a fixed-length parameter list. The procedure call for a verb uses the syntax standard to programming languages.

vital product data (VPD). A structured description of a device or program that is recorded at the manufacturing site.

W

workstation. A terminal or microcomputer, usually one that is connected to a mainframe or a network, and from which a user can perform applications.

Numerics

4764. IBM 4764 PCI-X Cryptographic Coprocessor.

4765. IBM 4765 PCIe Cryptographic Coprocessor.

4767. IBM 4767 PCIe Cryptographic Coprocessor.

10 Index