IBM 4765 PCIe Cryptographic Coprocessor
Custom Software Interface Reference
Sixth Edition (December, 2015)
This and other publications related to the IBM 4765 PCIe Cryptographic Coprocessor can be obtained in PDF format from the product Web site. Click on the PCIe Cryptographic Coprocessor link at http://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.

© Copyright International Business Machines Corporation 2011, 2015.
Note to U. S. Government Users—Documentation related to restricted rights Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.
Contents

About this document............................................................................................................................... vii
Prerequisite knowledge.............................................................................................................................. vii
Organization of this document............................................................................................................... vii
Typographic conventions...................................................................................................................... vii
Syntax diagrams.................................................................................................................................... viii
Related publications............................................................................................................................ viii
Summary of changes............................................................................................................................. viii

1 Overview............................................................................................................................................. 1
  1.1 System architecture....................................................................................................................... 1
  1.2 Host and coprocessor interaction................................................................................................. 2
  1.3 Virtual packets.............................................................................................................................. 3
  1.4 Byte order..................................................................................................................................... 4
  1.5 Request priority............................................................................................................................. 4
  1.6 Software attacks and defensive coding....................................................................................... 4
  1.7 Sample applications..................................................................................................................... 5
    1.7.1 How to compile and link the sample programs...................................................................... 5

2 Host-side API...................................................................................................................................... 6
  2.1 General information....................................................................................................................... 6
    2.1.1 Host-side API functions........................................................................................................ 6
    2.1.2 Header files........................................................................................................................... 6
    2.1.3 Sample code.......................................................................................................................... 6
    2.1.4 Error codes............................................................................................................................ 7
    2.1.5 xcAdapterCount - count installed coprocessors................................................................. 8
    2.1.6 xcOpenAdapter - open channel to coprocessor................................................................. 9
    2.1.7 xcRequest - send request to coprocessor application..................................................... 11
    2.1.8 xcCloseOperation - close channel to coprocessor............................................................ 14
    2.1.9 xcGetAdapterData - retrieve identification data from a coprocessor............................ 15
    2.1.10 xcGetHardwareVersion - retrieve a coprocessor's hardware version............................ 17
    2.1.11 xcResetAdapter - reset a coprocessor............................................................................. 18

3 Coprocessor-side API..................................................................................................................... 19
  3.1 General information..................................................................................................................... 19
    3.1.1 Coprocessor-side API functions.......................................................................................... 19
    3.1.2 Host communication functions............................................................................................. 19
    3.1.3 Hash functions..................................................................................................................... 20
    3.1.4 Symmetric key functions....................................................................................................... 20
    3.1.5 Public key algorithm functions............................................................................................. 21
    3.1.6 Large integer modular math functions................................................................................ 21
    3.1.7 Random number generator functions.................................................................................. 21
    3.1.8 Coprocessor configuration functions................................................................................... 21
    3.1.9 Outbound authentication functions....................................................................................... 22
    3.1.10 Header files.......................................................................................................................... 22
    3.1.11 Sample code........................................................................................................................ 22
    3.1.12 Device names and directories............................................................................................ 23
    3.1.13 Data format.......................................................................................................................... 23
    3.1.14 Mapped kernel buffers and DMA-eligible buffers.......................................................... 23

3.2 Host communication functions.................................................................................................. 25
  3.2.1 xcAttachWithCDUoOption - register to receive requests................................................. 25
  3.2.2 xcInitMappings - prepare memory mappings for driver buffers....................................... 26
  3.2.3 xcGetRequest - get request from the host.......................................................................... 27
  3.2.4 xcPutReply - send a reply to the host.................................................................................. 29
  3.2.5 xcKillMappings - release memory mappings for driver buffers...................................... 32
  3.2.6 xcDetach - deregister a coprocessor application............................................................... 33
  3.2.7 xcQueryMRBstatus - count active and available host requests....................................... 34
3.3 Hash functions .......................................................... 36
  3.3.1 xcSha1 - SHA-1 hash ........................................... 36
  3.3.2 xcSha2 - SHA-2 hash ........................................... 39
  3.3.3 Chained operations ............................................. 41
3.4 AES functions .......................................................... 42
  3.4.1 xcAES - AES encryption / decryption / MAC ........... 42
  3.4.2 xcAESKeyWrap* and xcAESKeyUnwrap* - AES key wrapping .............................................. 46
  3.4.3 xcAESKeyWrapX9102 and xcAESKeyWrapX9102Hash ............................................................ 47
  3.4.4 xcAESKeyUnwrapX9102 and xcAESKeyUnwrapX9102Hash .................................................... 50
  3.4.5 xcAESKeyWrapNIST ............................................ 53
  3.4.6 xcAESKeyUnwrapNIST ........................................ 55
3.5 DES functions .......................................................... 57
  3.5.1 xcDES - triple DES encryption/decryption/MAC ................................................................. 57
  3.5.2 xcDES - DES encryption/decryption/MAC ............................................................. 61
  3.5.3 xcDES8Bytes - eight-byte DES ........................................ .............................................. 64
  3.5.4 xcDES3Key - eight-byte triple DES ........................................ ........................................... 66
3.6 Public key algorithm functions ....................................... 68
  3.6.1 RSA key tokens ................................................. 68
  3.6.2 xcRSAKeyGenerate - generate an RSA keypair ................................................................. 72
  3.6.3 xcRSA - encrypt/decrypt data or wrap/unwrap X9.31 encapsulated hash .................................. 77
  3.6.5 xcComputeBlindingValues - compute blinding values for RSA key ....................................... 82
  3.6.6 DSA key tokens ................................................... 84
  3.6.7 DSA signature tokens ......................................... 85
  3.6.8 xcDSAGenerate - generate DSA keypair ................................................................. 86
  3.6.9 xcDSA - sign data or verify signature for data ................................................................. 89
3.7 Large integer modular math functions ................................ 91
  3.7.1 Large integers ..................................................... 91
  3.7.2 xcModMath - perform modular computations ................................................................. 92
3.8 Random number generator functions ................................ 94
  3.8.1 xcDRBGenRandom - instantiate a DRBG random number generator .................................... 94
  3.8.2 xcDRBGGenerate - generate random num ................. .......................................................... 96
  3.8.3 xcDRBGenRandom - generate random num ................................................................. 99
  3.8.4 xcDRBGtest - test the RNG functions ................................................................. 101
  3.8.5 xcDRBGUninstantiate - uninstantiate DRBG ................................................................. 103
  3.8.6 xcRandomNumberGenerate - generate random number ..................................................... 104
  3.8.7 xcTestRandomNumber - test random number generator .................................................. 106
3.9 Configuration functions ................................................ 107
  3.9.1 Privileged operations ........................................... 107
  3.9.2 Date/time ......................................................... 107
  3.9.3 xcGetOption - get coprocessor configuration ................................................................. 107
  3.9.4 xcGetOptions - get coprocessor configuration ................................................................. 107
  3.9.5 xcGetLowBattery - clear coprocessor low battery warning latch ...................................... 111
  3.9.6 xcGetLowBattery - clear coprocessor low battery warning latch ...................................... 112
3.10 Outbound authentication functions ................................... 113
  3.10.1 Coprocessor architecture ....................................... 113
  3.10.2 Overview of the authentication scheme ................................................................. 114
  3.10.3 OA certificates .................................................. 123
  3.10.4 xcOAGetDir - count and list OA certificates ................................................................. 133
  3.10.5 xcOAGetCert - retrieve an OA certificate ................................................................. 135
  3.10.6 xcOAGenerate - generate application keypair and OA certificate ...................................... 136
  3.10.7 xcOAGenerate - generate application keypair and OA certificate ...................................... 139
  3.10.8 xcOAPrivOp - perform cryptographic operation with an application key ................................ 140
  3.10.9 xcOASStatus - get coprocessor status ................................................................. 142
4 Error code formatting ................................................... 145
5 DES weak, semi-weak, and possibly weak keys ...................... 147
6 IBM root public keys ..................................................... 149
7 Notices ................................................................. 152
7.1 Copying and distributing softcopy files.................................................................152
7.2 Trademarks...........................................................................................................152
8 List of abbreviations and acronyms......................................................................154
9 Glossary..................................................................................................................156
10 Index.....................................................................................................................162
Figures

Figure 1 System architecture.................................................................2
Figure 2 Initial certificate chain.................................................................117
Figure 3 Application generates configuration key........................................118
Figure 4 Operating system updated.............................................................119
Figure 5 Application generates new configuration key.................................120
Figure 6 Miniboot updated........................................................................121
Figure 7 Configuration keypair and epoch keypair created..........................122
Figure 8 Foreign application loaded.............................................................123
Figure 9 Structure of an OA certificate.......................................................124
About this document

The IBM 4765 PCIe Cryptographic Coprocessor Custom Software Interface Reference describes the 4765 application programming interface (API) function calls that applications running on the 4765 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 4765 PCIe Cryptographic Coprocessor Custom Software Developer's Toolkit Guide) may also be helpful.

Organization of this document

“Overview” discusses the separation of the 4765 API into host-side 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 4765 API in detail.

“Coprocessor-side API” describes the coprocessor-side portion of the 4765 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 keys” lists the modulus (in hex) of 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.
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:

Publications specific to the IBM 4765 PCIe Cryptographic Coprocessor and to CCA are available at:

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/pciecc/library.shtml.


Summary of changes
This edition of the IBM 4765 PCIe Cryptographic Coprocessor Custom Software Interface Reference contains product information that is current with the IBM 4765 PCIe Cryptographic Coprocessor announcements.
1 Overview

The IBM 4765 API allows applications running on the host to interact with applications running on the cryptographic coprocessor. The 4765 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. *

Other drivers on the coprocessor control sensitive pieces of the hardware (for example, the DES encryption circuitry). Nearly all of these drivers use ioctls to handle requests for service from an application running on the coprocessor. These ioctls execute on the application's thread and block the application until the service is complete.

---

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

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. Exactly one thread within each coprocessor application must call `xcAttachWithCDUOption` on behalf of all the threads owned by the application in order for any API calls (except `xcPutEvent`) made from that application to be honored.

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 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 when the host request arrives. The coprocessor application processes the request, possibly requesting services (for example, DES or RSA operations) from coprocessor drivers.

7. 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 4 through 7 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`).

8. The host application calls `xcCloseAdapter` to close the communication channel between the host application and the coprocessor.

9. The coprocessor application calls `xcDetach` to refuse the receipt of host requests.

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 (`data_start`) 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 virtual packet (which determines how much padding is added at the end) depends on the number of buffers present in the packet. If a single buffer is passed, the length is given by the formula:
Presently, `COMM_MAGIC_1_BUFFER` is 18.

If two buffers are passed, the length is given by the formula:

\[
\left(\left(\left(COMM\_MAGIC\_1\_BUFFER + \text{sizeof}(\text{xcVirtualPacket\_t}) - 1 + (\text{number of bytes in request control block}) + 7\right) / 8\right) \times 8\right) - COMM\_MAGIC\_1\_BUFFER
\]

Presently, `COMM_MAGIC_2_BUFFERS` is 30.

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.

### 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.

In particular, the host application must know the proper byte order to use for the fields in the coprocessor application's `agentID\_t` in order to direct requests to the coprocessor application.

### 1.5 Request priority

The coprocessor maintains four 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.

IBM's Common Cryptographic Architecture (CCA) application uses three queues in its operation and adopts the convention that queue 0 is High Priority and queue 2 is Low Priority. The host portion of CCA sends requests that are generally executed very quickly (for example, requests for random numbers) to queue 0 and those that are generally executed very slowly (for example, requests to generate an RSA keypair) to queue 2. When the coprocessor portion of CCA searches the queues for a request, it first examines queue 0, then queue 1, and finally queue 2.

### 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 \textit{exit()} or by returning from \textit{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 with the stack dump will
appear in the host /var/log/messages file.

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:

- \textit{OEM_hdr.h} defines the protocol used between the host and coprocessor applications.
- \textit{OEM_card.c} is the coprocessor application source code.
- \textit{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
\textit{OEM_host.c} should be compiled and linked just like any other application on the host (link with
\textit{libcsulcca.so}).

Refer to the \textit{IBM 4765 PCIe Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide} for
information on how to compile and link \textit{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 \textit{OEM_hdr.h} and the various IBM 4765 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 4765 API (host API) allows an application running on the host to exchange information with an application running on a cryptographic coprocessor.

Host 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 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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xcAdapterCount</td>
<td>Determine the number of cryptographic coprocessors installed on the host.</td>
</tr>
<tr>
<td>xcOpenAdapter</td>
<td>Establish a communication channel to a specific coprocessor.</td>
</tr>
<tr>
<td>xcRequest</td>
<td>Send a request across an open communication channel to a specific application and receive the reply.</td>
</tr>
<tr>
<td>xcCloseAdapter</td>
<td>Close a communication channel that was previously opened via a call to xcOpenAdapter or xcMBOpenAdapter.</td>
</tr>
<tr>
<td>xcGetAdapterData</td>
<td>Retrieve identification data from a coprocessor.</td>
</tr>
<tr>
<td>xcGetHardwareVersion</td>
<td>Retrieve a coprocessor's hardware version.</td>
</tr>
<tr>
<td>xcResetAdapter</td>
<td>Reset a coprocessor.</td>
</tr>
</tbody>
</table>

All host 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 *xc_types.h* and *xc_err.h*. The code that implements the host API functions is in *libcsulcca.so*. The library is included in the Toolkit; refer to the *IBM 4765 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 API functions can be found in the following files shipped with the IBM 4765 PCIe Cryptographic Coprocessor Custom Software Developer's Toolkit:
2.1.4 Error codes

“Error code formatting” on page 146 describes the format of a return code. Note that although the host 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 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 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 API call invokes the host operating system for service and the invocation fails, the host 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 \texttt{xcAdapterCount} - count installed coprocessors

\texttt{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:

\texttt{pAdapterCount} contains the address of a buffer large enough to hold an item of type \texttt{xcAdapterNumber_t}.

Output

On successful exit from this routine:

\texttt{*pAdapterCount} contains the number of coprocessors installed on the host.

Notes

An \texttt{xcOpenAdapter} call is not required prior to this request.

\textbf{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.

\textbf{\texttt{xcAdapterNumber_t} is arithmetic}

An item of type \texttt{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:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDDGood (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>HDDError</td>
<td>The operation was unsuccessful.</td>
</tr>
<tr>
<td>HOST_OS_ERROR</td>
<td>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).</td>
</tr>
</tbody>
</table>

Refer to \texttt{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 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:

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDDGood (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>HDDInvalidParm</td>
<td>One or more inputs were not valid.</td>
</tr>
<tr>
<td>HDDTooManyOpens</td>
<td>The device driver or host operating system cannot create a new communication channel due to lack of resources.</td>
</tr>
<tr>
<td><strong>HDDDeviceBusy</strong></td>
<td>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.</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>HDDError</strong></td>
<td>The operation was unsuccessful.</td>
</tr>
<tr>
<td><strong>HOST_OS_ERROR</strong></td>
<td>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).</td>
</tr>
</tbody>
</table>

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

```c
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).

    - **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. Call the CALM_MAX_TRANSFER_DATA macro, defined in cmncryt2.h, to calculate the maximum transfer size. This is not a constant but instead it depends on what is in the request block. Both **RequestControlBlkLength** and the sum of **RequestControlBlkLength + RequestDataLength** are affected by this limit.

        - **RequestDataAddress** is the address of a buffer containing the request data.

    - **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 reply 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.

**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 29 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 29.
  
  - 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.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>HDDGood (that is, 0)</th>
<th>The operation was successful.</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDDError</td>
<td>The operation was unsuccessful.</td>
</tr>
<tr>
<td>HOST_OS_ERROR</td>
<td>An error occurred on a call to the host operating system.</td>
</tr>
</tbody>
</table>

Common return codes placed in the **Status** field by the host device driver or the coprocessor application are:
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDDInvalidLength</td>
<td>The length of a buffer associated with the request is not a multiple of 8 or is out of range.</td>
</tr>
<tr>
<td>HDDInvalidParm</td>
<td>One or more inputs were not valid.</td>
</tr>
<tr>
<td>HDDDeviceBusy</td>
<td>Due to the lack of resources, a new request cannot be initiated until a pending request has completed. Try again later.</td>
</tr>
<tr>
<td>HDDRequestAborted</td>
<td>The request was aborted (for example, because an application on the coprocessor faulted).</td>
</tr>
<tr>
<td>HDDError</td>
<td>The operation was unsuccessful.</td>
</tr>
</tbody>
</table>

Common status codes returned from the Communication Driver are:

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCCM_UNDELIVERABLE</td>
<td>The identifier in pRequestBlock-&gt;AgentID does not match the identifier of any registered agent on the coprocessor.</td>
</tr>
<tr>
<td>XCCM_PROCESS_DIED</td>
<td>The coprocessor application process died or is dying.</td>
</tr>
</tbody>
</table>

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**

```c
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 API function.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDDGood (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>HDDError</td>
<td>The operation was unsuccessful.</td>
</tr>
<tr>
<td>HOST_OS_ERROR</td>
<td>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).</td>
</tr>
</tbody>
</table>

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,
                               uint32_t          length);
```

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.
- `length` is the length in bytes of the buffer referenced by `pVpd`. `length` must be >= sizeof(xcVpd_t).

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 4765-001 PCIe 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.
  - `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*.
• *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:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDDGood (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>HDDError</td>
<td>The operation was unsuccessful.</td>
</tr>
<tr>
<td>HDDInvalidParm</td>
<td>The length was too short.</td>
</tr>
<tr>
<td>HOST_OS_ERROR</td>
<td>An error occurred on a call to the host operating system (the low-</td>
</tr>
<tr>
<td></td>
<td>order 16 bits contain the return code from the call that failed).</td>
</tr>
</tbody>
</table>

Refer to *xc_err.h* for a comprehensive list of return codes.
2.1.10 xcGetHardwareVersion - retrieve a coprocessor's hardware version

xcGetHardwareVersion retrieves a coprocessor's hardware version.

**Function prototype**

```c
unsigned int xcGetHardwareVersion(xcAdapterHandle_t AdapterHandle, 
                                   xcHdwVer_t *pHdwVersion);
```

**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**.

- **pHdwVersion** contains the address of a buffer large enough to hold an item of type **xcHdwVer_t**.

**Output**

On successful exit from this routine:

- **pHdwVersion** contains the coprocessor's hardware version. The fields of **pHdwVersion** are set as follows:
  
  - **length** is the length in bytes of the structure referenced by **pHdwVersion**.
  
  - **regHCSR** indicates the hardware version of the coprocessor. The version information is split – part is in the four high order bits of **regHCSR** (bits 0-3) and part is in the high order byte of the low order word (bits 16-23).

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDDGood</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>HDDError</td>
<td>The operation was unsuccessful.</td>
</tr>
<tr>
<td>HOST_OS_ERROR</td>
<td>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).</td>
</tr>
</tbody>
</table>

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**

```c
unsigned int xcResetAdapter(xcAdapterHandle_t hAdapterHandle);
```

**Input**

On entry to this routine:

*hAdapterHandle* identifies a communication channel to one of the cryptographic coprocessors installed on the host. *hAdapterHandle* 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:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDDGood</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>HDDError</td>
<td>The operation was unsuccessful.</td>
</tr>
<tr>
<td>HOST_OS_ERROR</td>
<td>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).</td>
</tr>
</tbody>
</table>

Refer to *xc_err.h* for a comprehensive list of return codes.
3 Coprocessor-side API

The coprocessor-side portion of the IBM 4765 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 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 crypto@us.ibm.com.

This chapter describes each of the functions supplied by the coprocessor 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 API includes functions in the following categories:

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

3.1.2 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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xcAttachWithCDUOption</td>
<td>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.</td>
</tr>
<tr>
<td>xcInitMappings</td>
<td>Prepare memory mappings to allow buffers owned by the Communication driver to be manipulated by a coprocessor application.</td>
</tr>
</tbody>
</table>
### 3.1.3 Hash 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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xcSHA1</td>
<td>Compute the hash of a block of data using the Secure Hash Algorithm (SHA-1) as defined in FIPS Publication 180-1.</td>
</tr>
<tr>
<td>xcSHA2</td>
<td>Compute the hash of a block of data using the Secure Hash Algorithm (SHA-2) as defined in FIPS Publication 180-2.</td>
</tr>
</tbody>
</table>

### 3.1.4 Symmetric key functions

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xcAES</td>
<td>Encipher or decipher an arbitrary amount of data using the AES algorithm.</td>
</tr>
<tr>
<td>xcAESKeyWrapX9102</td>
<td>Wrap or unwrap a cryptographic key using the ANSI X9.102 or the NIST algorithm¹. 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.</td>
</tr>
<tr>
<td>xcAESKeyUnwrapX9102</td>
<td></td>
</tr>
<tr>
<td>xcAESKeyWrapX9102Hash</td>
<td></td>
</tr>
<tr>
<td>xcAESKeyUnwrapX9102Hash</td>
<td></td>
</tr>
<tr>
<td>xcAESKeyWrapNIST</td>
<td></td>
</tr>
<tr>
<td>xcAESKeyUnwrapNIST</td>
<td></td>
</tr>
<tr>
<td>xcTDES</td>
<td>Encipher or decipher an arbitrary amount of data or generate a message authentication code using the triple-DES algorithm.</td>
</tr>
<tr>
<td>xcDES</td>
<td>Encipher or decipher an arbitrary amount of data or generate a message authentication code (MAC) using the DES algorithm.</td>
</tr>
<tr>
<td>xcDES8bytes</td>
<td>Encipher or decipher eight bytes of data using the DES algorithm.</td>
</tr>
<tr>
<td>xcDES3Key</td>
<td>Triple-encipher (wrap) or triple-decipher (unwrap) a</td>
</tr>
</tbody>
</table>

cryptographic key using the DES algorithm.

3.1.5 Public key algorithm 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.
- **xcComputeBlindingValues**: Compute blinding values used to defeat timing-based attacks against an RSA key.
- **xcDS AKKeyGenerate**: Generate a DSA keypair.
- **xcDSA**: Sign or verify the signature for an arbitrary amount of data using the DSA algorithm.

3.1.6 Large integer modular math functions
These functions allow a coprocessor application to ask the PKA Driver to perform specific operations on large integers:

- **xcModMath**: Perform a modular multiplication \( C = A \times B \mod N \), modular exponentiation \( C = A^B \mod N \), or modular reduction \( C = A \mod N \).

3.1.7 Random number generator 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:

- **xcRandomNumberGenerate**: Generate a 64-bit random number.
- **xcTestRandomNumber**: Ensure the random number generator meets the applicable FIPS standards.

3.1.8 Coprocessor configuration functions
These functions configure certain processor features or return information about the coprocessor:
### 3.1.9 Outbound authentication functions

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xcGetConfig</td>
<td>Get information about the coprocessor.</td>
</tr>
<tr>
<td>xcClearILatch</td>
<td>Clear the coprocessor intrusion latch.</td>
</tr>
<tr>
<td>xcClearLowBatt</td>
<td>Clear the coprocessor low battery warning latch.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xcOAGetDir</td>
<td>Count or list all certificates.</td>
</tr>
<tr>
<td>xcOAGetCert</td>
<td>Retrieve a certificate.</td>
</tr>
<tr>
<td>xcOAGenerate</td>
<td>Generate a keypair and a certificate that contains the public half of the keypair.</td>
</tr>
<tr>
<td>xcOADelete</td>
<td>Delete a certificate and the corresponding keypair.</td>
</tr>
<tr>
<td>xcOAPrivOp</td>
<td>Perform an operation using one of the keys from a keypair generated by <code>xcOAGenerate</code>.</td>
</tr>
<tr>
<td>xcOAVerify</td>
<td>Verify the signature in one certificate using the public key from another certificate.</td>
</tr>
<tr>
<td>xcOAStatus</td>
<td>Obtain information about the status of and the software installed on the coprocessor.</td>
</tr>
</tbody>
</table>

### 3.1.10 Header files

The prototypes for most coprocessor 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 4765 PCIe Application Program Development Toolkit. Refer to the *IBM 4765 PCIe Cryptographic Coprocessor Custom Software Developer's Toolkit Guide* for details.

### 3.1.11 Sample code

Examples of the use of many of the coprocessor API functions can be found in the following subdirectories shipped with the IBM 4765 PCIe Cryptographic Coprocessor Custom Software Developer’s Toolkit:

- **Host Communication Functions**
  - `y4tk/samples/toolkit/oem/host`
  - `y4tk/samples/toolkit/rte/host`
  - `y4tk/samples/toolkit/skeleton/host`

- **Hash Functions**
  - `y4tk/samples/toolkit/skeleton/hshserv`

- **Symmetric Key Functions**
3.1.12 Device names and directories

The embedded system includes the devices listed below. A coprocessor application must open a device (using `open(2)`) in order to request the corresponding service. For example, an application must open `/dev/skch` in order to issue requests to hash data.

- `/dev/crypto` IBM 4765 adapter (general configuration)
- `/dev/pka` PKA device (RSA/DSA/large integer operations)
- `/dev/skch` Symmetric Key Cipher Hash device (AES/DES/SHA)
- `/dev/hwrng` Hardware random number generator (RNG)
- `/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.

3.1.13 Data format

Unless otherwise noted, all integers are in the native format for the processor. For the IBM 4765 coprocessor, the native format is big-endian.

3.1.14 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. If a buffer does not have those properties, the coprocessor Communication driver must copy data to or from...
A coprocessor application cannot in general ensure that a buffer it creates has the necessary properties, so the Communication driver endeavors to provide suitable buffers whenever possible. In particular, when a coprocessor application asks the Communication driver to return the next request received from the host,

1. The Communication driver will attempt to 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-only, and only the thread that receives the request is guaranteed to be able to see it. Other threads may or may not be able to see the buffer. Attempts to write to the buffer, or actions that cause the kernel to attempt to write to the buffer, may fail silently.

2. The Communication driver may map one or 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, and only the thread that receives the request can see them.

See “xcGetRequest - get request from the host” on page 27 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 4765 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 4765 device drivers. Each embedded application must issue an attach with a distinct agent ID before calling any other function.

Function prototype

```c
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 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  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 23 for details.

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

Function prototype

```c
int xcInitMappings(int fd);
```

Input

On entry to this routine:

`fd` is a file descriptor returned by `xcAttachWithCDUOption`.

Output

None.

Notes

Each application thread that wishes to take advantage of the performance improvement made possible by `xcInitMappings` must call `xcInitMappings` once before calling `xcGetRequest`.

The Communication driver maintains state information about each application thread for which a map has been established (and not yet freed). This limits the number of such threads to 75.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITMAP_SUCCESS</td>
<td>The initialization was successful.</td>
</tr>
<tr>
<td>INITMAP_FAILED</td>
<td>The initialization was unsuccessful.</td>
</tr>
</tbody>
</table>

Refer to `xc_err.h` for a comprehensive list of return codes.
3.2.3 xcGetRequest - get request from the host

xcGetRequest waits for the host to send a request to the coprocessor application and returns the first available request.

Function prototype

```c
int xcGetRequest(int fd, getReq_t *pGetReq);
```

Input

On entry to this routine:

- `fd` is a file descriptor returned by `xcAttachWithCDUOption`.
- `pGetReq` contains the address of a request block whose fields are initialized as follows:
  - `startMRB` is the index of the first request queue to examine to find a pending request (see “Request priority” on page 4).
  - `endMRB` is the index of the last request queue to examine to find a pending request (see “Request priority” on page 4).
  - If `startMRB` is less than `endMRB`, `xcGetRequest` searches the request queues in increasing order of index (for example, 0, 1, 2). If `startMRB` is greater than `endMRB`, `xcGetRequest` searches the request queues in decreasing order of index (for example, 2, 1, 0). If the two fields are equal, `xcGetRequest` examines a single queue.
  - `pVPacket` points to a buffer into which the coprocessor Communication driver may place the virtual packet received from the host (see “Virtual packets” on page 3). The buffer must be large enough to accommodate the virtual packet.

The driver may elect to map the virtual packet into the caller’s address space rather than copying it to the buffer referenced by `pVPacket` (see “Output” below).

Output

On successful exit from this routine:

The following fields of `pGetReq` are changed as noted:

- `pVPacket` either is NULL or points to the first byte of the virtual packet received from the host.
  - If `pVPacket` is NULL, the virtual packet was copied to the buffer supplied by the caller (i.e., the buffer referenced by `pVPacket` when `xcGetRequest` was called).
  - If `pVPacket` is not NULL, the Communication driver mapped the virtual packet into the caller’s address space beginning at the address specified by `pVPacket`. In this case, the virtual packet is mapped read-only. See “Mapped kernel buffers and DMA-eligible buffers” on page 23.
- `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.
- `offsHRB` is state information from the Communication driver.
- If `pVbuff1` is not NULL, the Communication driver mapped a buffer into the caller’s address space beginning at the address specified by `pVbuff1`. The caller must put the contents of the Reply...
Control Block portion of the reply to the request received from the host in this buffer (see “xcPutReply - send a reply to the host” on page 29). The buffer is 64K bytes long and is mapped read-write. See “Mapped kernel buffers and DMA-eligible buffers” on page 23.

If `pVbuff1` is NULL, the Communication driver has not provided a buffer to the caller. The caller must make its own arrangements for the Reply Control Block portion of the reply.

`pVbuff1` will be NULL if the caller did not call `xcInitMappings` prior to calling `xcGetRequest`.

- If `pVbuff2` is not NULL, the Communication driver mapped a buffer into the caller's address space beginning at the address specified by `pVbuff2`. The caller must put the contents of the Reply Data portion of the reply to the request received from the host in this buffer (see “xcPutReply - send a reply to the host” on page 29). The buffer is 64K bytes long and is mapped read-write. See “Mapped kernel buffers and DMA-eligible buffers” on page 23.

If `pVbuff2` is NULL, the Communication driver has not provided a buffer to the caller. The caller must make its own arrangements for the Reply Data portion of the reply.

`pVbuff2` will be NULL if the caller did not call `xcInitMappings` prior to calling `xcGetRequest`.

**Notes**

A thread that has called `xcGetRequest` must call `xcPutReply` to end the request before the thread calls `xcGetRequest` again. And only the thread that receives a request via `xcGetRequest` can call `xcPutReply` to end the request.

Neither the request block referenced by `pGetReq` nor the buffer referenced on entry to the routine by `pGetReq->pVPacket` may reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

The following fields of `*pGetReq` returned by a successful call to `xcGetRequest` must be passed unchanged in the corresponding fields of the `putRep_t` structure passed as the second argument to the call to `xcPutReply` that returns the results of the request: `pVPacket`, `srcMRB`, `sizeHRB`, and `offsHRB`.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GETREQUEST_SUCCESS</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>GETREQUEST_FAILED</td>
<td>The operation was unsuccessful.</td>
</tr>
</tbody>
</table>

Refer to `xc_err.h` for a comprehensive list of return codes.
3.2.4 xcPutReply - send a reply to the host

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

Function prototype

```c
int xcPutReply(int fd, putRep_t *pPutRep);
```

Input

On entry to this routine:

- *fd* is a file descriptor returned by *xcAttachWithCDUOption*.
- *pPutRep* contains the address of a reply block whose fields are initialized as follows:

  - **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,
    - `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.
  - The buffer must be no longer than the corresponding buffer allocated by the host application when it calls *xcRequest*. `tagLen.dataLen` may not be zero.
  - **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.
  - The following fields of the reply block must be set to the values of the corresponding fields in the request block returned by *xcGetRequest* when the request for which *xcPutReply* is issuing the reply was received: srcMRB, pVPacket, sizeHRB, and offsHRB.
  - **reqID** must be set to the value of the RequestID field in the virtual packet header returned by *xcGetRequest* when the request for which *xcPutReply* is issuing the reply was received.

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:
Success.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_OCPRB_NOTFOUND</td>
<td>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.</td>
</tr>
<tr>
<td>T_REPDAT_NOTFOUND</td>
<td>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.</td>
</tr>
<tr>
<td>T_OCPRB_DATA_TRUNC</td>
<td>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.</td>
</tr>
<tr>
<td>T_REPDAT_DATA_TRUNC</td>
<td>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.</td>
</tr>
</tbody>
</table>

- \( \text{tankMRB0} \) is the number of host requests pending on queue 0. See “Request priority” on page 4.
- \( \text{tankMRB1} \) is the number of host requests pending on queue 1.
- \( \text{tankMRB2} \) is the number of host requests pending on queue 2.

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

- \( \text{pGetReq->pVbuff1} \neq \text{NULL} \), the vptr field in the pTLV entry whose tag is TAG_OCPRB must be set to the value xcrequest returned in pGetReq->pVbuff1.
- \( \text{pGetReq->pVbuff2} \neq \text{NULL} \), the vptr field in the pTLV entry whose tag is TAG_REPDAT must be set to the value xcrequest returned in pGetReq->pVbuff2.

Failure to follow these rules will cause a resource leak.

The reply block referenced by pPutRep must not reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUTREPLY_SUCCESS</td>
<td>The operation was successful.</td>
</tr>
</tbody>
</table>
The operation was unsuccessful.

Refer to \textit{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**

```plaintext
int xcKillMappings(int fd);
```

**Input**

On entry to this routine:

- `fd` is a file descriptor returned by `xcAttachWithCDUOption`.

**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:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KILLMAP_SUCCESS</td>
<td>The function was successful.</td>
</tr>
<tr>
<td>KILLMAP_FAILED</td>
<td>The function was unsuccessful.</td>
</tr>
</tbody>
</table>

Refer to `xc_err.h` for a comprehensive list of return codes.
3.2.6 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

\[
\text{int xcDetach(int fd);}\]

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 application calls \(xcDetach\), those requests are canceled. The host application receives a reply whose \textit{Status} field is \texttt{XCCM\_UNDELIVERABLE}.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DETACH_SUCCESS</td>
<td>The detach was successful.</td>
</tr>
<tr>
<td>DETACH_FAILED</td>
<td>The detach was unsuccessful.</td>
</tr>
</tbody>
</table>

Refer to \texttt{xc\_err.h} for a comprehensive list of return codes.
3.2.7 xcQueryMRBstatus - count active and available host requests

xcQueryMRBstatus examines three of the four request queues and returns the number of active and available host requests for each queue.

Function prototype

```c
int xcQueryMRBstatus(int fd, queryStat_t *pQStat);
```

Input

On entry to this routine:

- `fd` is a file descriptor returned by `xcAttachWithCDUOption`.
- `pQStat` contains the address of a buffer large enough to hold an item of type `queryStat_t`.

Output

On successful exit from this routine:

The following fields of `pQStat` are changed as noted:

- `tankMRB0` contains the number of active and available host requests in queue 0. See “Request priority” on page 4.
- `tankMRB1` contains the number of active and available host requests in queue 1.
- `tankMRB2` contains the number of active and available host requests in queue 2.

Notes

The term “active and available host requests” means

- a request that has been sent by the host and is waiting to be delivered to the coprocessor application (i.e., a request that might be returned when the coprocessor application calls `xcGetRequest`) (an “available request”) or
- a request that has been received by the coprocessor application but that has not yet been completed (i.e., a request for which the coprocessor application has not yet called `xcPutReply`) (an “active request”).

The values returned in `pQStat` are volatile. A new request may arrive after the call has counted available requests but before the caller has received the result of the call. Also, if the coprocessor application is multithreaded, the values returned to one thread may be stale because another thread may have called `xcPutReply` before the values were seen by the first thread.

The buffer referenced by `pQStat` must not reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUERYSTAT_SUCCESS</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>QUERYSTAT_FAILED</td>
<td>The operation was unsuccessful.</td>
</tr>
</tbody>
</table>
Refer to xc_err.h for a comprehensive list of return codes.
3.3 **Hash 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 must open `/dev/skch` with the `O_RDWR` flag 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**

```c
unsigned int xcSha1(int          fd,
                     xcSHA1_RB_t *pSHA1_rb);
```

**Input**

On entry to this routine:

- `fd` is the file descriptor returned when `/dev/skch` was opened.
- `pSHA1_rb` contains the address of a SHA-1 request block whose fields are initialized as follows:
  - `options` controls the operation of the function and must be set to the logical OR of constants from the following category:
    - **Operating mode**
      - `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 41.
      - `SHA1_MSGPART_MIDDLE`: The input data constitutes an additional portion of a block of data to be hashed. See “Chained operations” on page 41.
      - `SHA1_MSGPART_FINAL`: The input data constitutes the final portion of a block of data to be hashed. See “Chained operations” on page 41.
    - `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.
    - `source.data_ptr` points to a buffer containing the input data.
    - `final_data` may hold the last few bytes of the input data.
      - If `options` specifies `SHA1_MSGPART_ONLY` or `SHA1_MSGPART_FINAL` and `source_length` is not a multiple of 4, the last `source_length` mod 4 bytes of input are not read from the buffer referenced by `source.data_ptr` but are instead taken from `final_data` (starting with `final_data[0]`).
• hash_value contains a partial hash returned by a prior call to xcSha1. hash_value is used only if options specifies SHA1_MSGPART_MIDDLE or SHA1_MSGPART_FINAL. See “Chained operations” on page 41.

• running_length contains the number of bytes of input that have been processed by prior calls to xcSHA1. See “Chained operations” on page 41.

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:

• hash_value contains the SHA-1 hash of the input data.

If, when xcSha1 was called, options specified SHA1_MSGPART_MIDDLE or SHA1_MSGPART_FINAL, hash_value also incorporates the value of hash_value 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 source.data_ptr points to a DMA-eligible buffer the hash operation may complete more quickly than would otherwise be the case. The SHA-1 request block referenced by pSHA1_rb must not reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMGood (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>DMBadParm</td>
<td>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).</td>
</tr>
<tr>
<td>DMBadAddr</td>
<td>Part of the buffer defined by source.data_ptr and source_length is not readable (i.e., lies in unmapped memory).</td>
</tr>
</tbody>
</table>
Refer to \textit{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**

```c
unsigned int xcSha2(int          fd,
                     xcSHA2_RB_t *pSHA2_rb);
```

**Input**

On entry to this routine:

- `fd` is the file descriptor returned when `/dev/skch` was opened.
- `pSHA2_rb` contains the address of a SHA-2 request block whose fields are initialized as follows:
  - `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:

<table>
<thead>
<tr>
<th>SHA2_MSPART ONLY</th>
<th>SHA2_MSPART_FIRST</th>
<th>SHA2_MSPART_MIDDLE</th>
<th>SHA2_MSPART_FINAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>The input data constitutes the entire block of data to be hashed. The hash value is computed and returned.</td>
<td>The input data constitutes the first portion of a block of data to be hashed. See “Chained operations” on page 41.</td>
<td>The input data constitutes an additional portion of a block of data to be hashed. See “Chained operations” on page 41.</td>
<td>The input data constitutes the final portion of a block of data to be hashed. See “Chained operations” on page 41.</td>
</tr>
</tbody>
</table>

**Hash method**

`options` must include exactly one of the following constants:

<table>
<thead>
<tr>
<th>SHA224_METHOD</th>
<th>SHA256_METHOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute a SHA 224 hash of the input.</td>
<td>Compute a SHA 256 hash of the input.</td>
</tr>
</tbody>
</table>

- `source_length` contains the length in bytes of the input data.
  If `options` specifies `SHA2_MSPART_FIRST` or `SHA2_MSPART_MIDDLE`, `source_length` must be a multiple of 64.
- `source.data_ptr` points to a buffer containing the input data.
- `hash_value` contains a partial hash returned by a prior call to `xcSha2`. `hash_value` is used only if `options` specifies `SHA1_MSPART_MIDDLE` or `SHA1_MSPART_FINAL`. See “Chained operations” on page 41.
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 41.

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:

• hash_value 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, 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 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 source.data_ptr points to a DMA-eligible buffer the hash operation may complete more quickly than would otherwise be the case. The SHA-2 request block referenced by pSHA2_rb must not reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMGood</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>DMNotAuth</td>
<td>The coprocessor application is not authorized to perform SHA operations (for example, because it has not called xcAttachWithCDUOption).</td>
</tr>
<tr>
<td>DMBadFlags</td>
<td>The options argument is not valid.</td>
</tr>
<tr>
<td>DMBadParm</td>
<td>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).</td>
</tr>
<tr>
<td>DMBadAddr</td>
<td>Part of the buffer defined by source.data_ptr and source_length is</td>
</tr>
</tbody>
</table>
Refer to `xc_err.h` for a comprehensive list of return codes.

### 3.3.3 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 or 2) 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 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 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` and must open `/dev/skch` with the `O_RDWR` flag 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**

```c
unsigned int xCAES(int         fd,
                    xCAES_RB_t *pAES_rb)
```

**Input**

On entry to this routine:

`fd` is the file descriptor returned when `/dev/skch` was opened.

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

- `options` controls the operation of the function and must be set to the logical OR of constants from the following categories:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES_ENCRYPT</td>
<td>Encrypt the input.</td>
</tr>
<tr>
<td>AES_DECRYPT</td>
<td>Decrypt the input.</td>
</tr>
<tr>
<td>AES_MAC</td>
<td>Generate a message authentication code (MAC) for the input.</td>
</tr>
</tbody>
</table>

   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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES_128BIT_KEY</td>
<td>The key length is 128 bits (16 bytes).</td>
</tr>
<tr>
<td>AES_192BIT_KEY</td>
<td>The key length is 192 bits (24 bytes).</td>
</tr>
<tr>
<td>AES_256BIT_KEY</td>
<td>The key length is 256 bits (32 bytes).</td>
</tr>
</tbody>
</table>

**Chaining mode**

`options` must include exactly one of the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES_CBC_MODE</td>
<td>Use outer Cipher Block Chaining (CBC) mode.</td>
</tr>
</tbody>
</table>
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.

Padding options

options may include a subset of the following constants:

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES PREPAD</td>
<td>Prepand the input with 16 bytes of data.</td>
</tr>
<tr>
<td>AES PAD WITH 16</td>
<td>Pad the input with 16 bytes of data.</td>
</tr>
<tr>
<td>AES PAD WITH 32</td>
<td>Pad the input with 32 bytes of data.</td>
</tr>
</tbody>
</table>

Only one of AES PAD WITH 16 and AES PAD WITH 32 may be specified.

- 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, options must specify AES PAD WITH 16 or AES PAD WITH 32 or both AES MAC and AES PREPAD.
- source.data_ptr points to a buffer containing the data to be processed by xcAES. The “input data” to xcAES 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 destination.data_ptr. destination_length must be at least as large as the length of the input data.
- destination.data_ptr points to a writeable buffer.
- prePadding contains 16 bytes of data that are prepended to the data to be processed by xcAES before the operation is performed if options specifies AES PREPAD.

If options does not specify AES PREPAD, prePadding is not used.

- postPadding contains up to 32 bytes of data that are appended to the data to be processed by xcAES before the operation is performed.

If options specifies AES PAD WITH 16, postPadding[0] through postPadding[15] are appended to the input. If options specifies AES PAD WITH 32, the entire postPadding buffer is appended to the input.

If options specifies neither AES PAD WITH 16 nor AES PAD WITH 32, postPadding is not used.
Output
On successful exit from this routine:
The following fields of \( *pAES\_rb \) are changed as noted:

- \( \text{term\_v} \) contains
  - the message authentication code (MAC) generated from the input data if \( \text{options} \) specifies \text{AES\_MAC}.
  - the initialization vector for the next AES operation if \( \text{options} \) specifies \text{AES\_CBC\_MODE}\(^2\).

\( \text{term\_v} \) is undefined otherwise.

- The buffer referenced by \( \text{destination\_data\_ptr} \) contains
  - The input data encrypted using key (and \( \text{init\_v} \) if \( \text{options} \) specifies \text{AES\_CBC\_MODE}) if \( \text{options} \) specifies \text{AES\_ENCRYPT}.
  - The input data decrypted using key (and \( \text{init\_v} \) if \( \text{options} \) specifies \text{AES\_CBC\_MODE}) if \( \text{options} \) specifies \text{AES\_DECRYPT}.

The buffer referenced by \( \text{destination\_data\_ptr} \) is undefined otherwise.

Notes
The length of the input data may be less than \( \text{destination\_length} \). In this case, any excess bytes at the end of the output buffer are not affected by \( \text{xcAES} \).

If \( \text{source\_data\_ptr} \) and/or \( \text{destination\_data\_ptr} \) point to a DMA-eligible buffer the AES operation may complete more quickly than would otherwise be the case. Neither the AES request block referenced by \( pAES\_rb \) nor the buffers referenced by \( pAES\_rb->\text{term\_v} \) and \( pAES\_rb->\text{destination\_data\_ptr} \) may reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

The buffers defined by \( \text{source\_data\_ptr}/\text{source\_length} \) and \( \text{destination\_data\_ptr}/\text{destination\_length} \) should not overlap.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>DMGood (that is, 0)</th>
<th>The operation was successful.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMNotAuth</td>
<td>The coprocessor application is not authorized to perform AES operations (for example, because it has not called \text{xcAttachWithCDUOption}).</td>
</tr>
<tr>
<td>DMBadFlags</td>
<td>The options argument is not valid.</td>
</tr>
<tr>
<td>DMBadParm</td>
<td>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 ( \text{source_length} ) is zero and no padding option was specified.</td>
</tr>
<tr>
<td>DMBadAddr</td>
<td>Part of the buffer defined by ( \text{source_data_ptr} ) and ( \text{source_length} )</td>
</tr>
</tbody>
</table>

\(^2\) If \( \text{options} \) specifies both \text{AES\_MAC} and \text{AES\_CBC\_MODE}, the MAC and the initialization vector are the same value.
is not readable, part of the buffer defined by destination.data_ptr 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.
3.4.2 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)

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:

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

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:

<table>
<thead>
<tr>
<th>ICV (8 bytes)</th>
<th>Key to be wrapped</th>
</tr>
</thead>
</table>

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 and xcAESKeyWrapX9102Hash

xcAESKeyWrapX9102 and xcAESKeyWrapX9102Hash wrap an AES key according to the X9.102 standard. The two functions differ in how they handle the associated data supplied by the user (see “X9.102 formatted buffer” on page 47):

- **xcAESKeyWrapX9102** allows the caller to supply up to 255 bytes of associated data. This data is incorporated into the formatted buffer without change.
- **xcAESKeyWrapX9102Hash** 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.

### Function prototype

```c
int xcAESKeyWrapX9102(int fd, xcaESKW_X9102_t *pKW);
int xcAESKeyWrapX9102Hash(int fd, xcaESKW_X9102hash_t *pKW);
```

### Input

On entry to this routine:

- **fd** is the file descriptor returned when `/dev/skch` was opened
- **pKW** contains the address of an AES key wrap request block whose fields are initialized as follows:
  - **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. This 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 47 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** (xcAESKeyWrapX9102Hash only) 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** (xcAESKeyWrapX9102Hash only) points to a writeable buffer large enough to hold the result of the hash operation.
• \textit{pHashAsDataLen} (\textit{xcAESKeyWrapX9102Hash} only) points to a writeable buffer that contains the length in bytes of the buffer referenced by \textit{pHashAsData}.

• \textit{pKeyData} points to a buffer containing the AES key to be wrapped.

• \textit{keyDataLen} is the length in bits of the key referenced by \textit{pKeyData}.

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

• \textit{pWrapData} points to a writeable buffer large enough to hold the wrapped key. See “X9.102 formatted buffer” on page 47 for details.

• \textit{pWrapDataLen} points to a writeable buffer that contains the length in bytes of the buffer referenced by \textit{pWrapData}.

\textbf{Output}

On successful exit from this routine:

The following fields of \textit{*pKW} are changed as noted:

• (\textit{xcAESKeyWrapX9102Hash} only) \textit{*pHashAsDataLen} contains the actual length in bytes of the hash that was incorporated into the X9.102 formatted buffer.

• \textit{*pWrapDataLen} contains the actual length in bytes of the wrapped key.

For \textit{xcAESKeyWrapX9102Hash} only, the buffer defined by \textit{pHashAsData} contains the hash that was incorporated into the X9.102 formatted buffer.

The buffer defined by \textit{pWrapData} contains the wrapped key.

\textbf{Notes}

(\textit{xcAESKeyWrapX9102Hash} only) The value of \textit{pKW->hashOption} is placed into the H1 field of the X9.102 formatted buffer in big-endian order.

Neither the AES key wrap request block referenced by \textit{pKW} nor the buffers referenced by \textit{pKW->pHashAsData}, \textit{pKW->pHashAsDataLen}, \textit{pKW->pWrapData}, and \textit{pKW->pWrapDataLen} may reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

\textbf{Return codes}

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>\textbf{DMGood} (that is, 0)</th>
<th>The operation was successful.</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{DMNotAuth}</td>
<td>The coprocessor application is not authorized to perform AES key wrapping operations (for example, because it has not called \textit{xcAttachWithCDUOption}).</td>
</tr>
<tr>
<td>\textbf{DMBadFlags}</td>
<td>An input argument was invalid (for example, \textit{pKW} is NULL or \textit{pKW-&gt;hashOption} specifies more than one algorithm or a buffer is too small).</td>
</tr>
</tbody>
</table>
Refer to `xc_err.h` for a comprehensive list of return codes.
3.4.4 xcAESKeyUnwrapX9102 and xcAESKeyUnwrapX9102Hash

xcAESKeyUnwrapX9102 and xcAESKeyUnwrapX9102Hash unwrap an AES key wrapped according to the X9.102 standard. The two functions differ in how they handle the associated data supplied by the user (see “X9.102 formatted buffer” on page 47):

- `xcAESKeyUnwrapX9102` 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.
- `xcAESKeyUnwrapX9102Hash` 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.

**Function prototype**

```c
int xcAESKeyUnwrapX9102(int fd, xcaESKUW_X9102_t *pKUW);

int xcAESKeyUnwrapX9102Hash(int fd, xcaESKUW_X9102hash_t *pKUW);
```

**Input**

On entry to this routine:

- `fd` is the file descriptor returned when `/dev/skch` was opened.
- `pKUW` contains the address of an AES key unwrap request block whose fields are initialized as follows:
  - `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 47 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` (only for `xcAESKeyUnwrapX9102Hash`) 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 *pKUW* 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 *pKUW-*pAsData/*pKUW-*asDataLen*.

For *xcAESKeyUnwrapX9102Hash*, the associated data in the buffer must match the value of *pKUW-*hashOption* (in big-endian order) concatenated with the hash value (computed using the algorithm dictated by *pKUW-*hashOption*) of the buffer defined by *pKUW-*pAsData/*pKUW-*asDataLen*.

5. All of the pad bits must be zero.

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

Neither the AES key unwrap request block referenced by *pKUW* nor the buffers referenced by *pKUW-*pClearKey* and *pKUW-*pClearKeyLen* may reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMGood</td>
<td>(that is, 0)  The operation was successful.</td>
</tr>
<tr>
<td>DMNotAuth</td>
<td>The coprocessor application is not authorized to perform AES key wrapping operations (for example, because it has not called</td>
</tr>
<tr>
<td>DMBadFlags</td>
<td>An input argument was invalid (for example, pKUW is NULL or pKUW-&gt;hashOption specifies more than one algorithm or a buffer is too small).</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DMBadX9102KUW</td>
<td>Unwrap operation did not produce a valid X9.102 formatted buffer.</td>
</tr>
</tbody>
</table>

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

xcAESKeyWrapNIST wraps an AES key according to the NIST standard (see “NIST formatted buffer” on page 47 for details):

**Function prototype**

```c
int xcAESKeyWrapNIST(int             fd,
                      xcAESKW_NIST_t *pKW);
```

**Input**

On entry to this routine:

- `fd` is the file descriptor returned when `/dev/skch` was opened.
- `pKW` contains the address of an AES key wrap request block whose fields are initialized as follows:
  - `pWrapKey` points to a buffer containing the wrapping key (an item of type `xAES_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 47 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 `*pKW` are changed as noted:

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

The buffer defined by `pWrapData` contains the wrapped key.

**Notes**

Neither the AES key wrap request block referenced by `pKW` nor the buffers referenced by `pKW->pWrapData`, and `pKW->pWrapDataLen` may reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

**Return codes**

Common return codes generated by this routine are:

<p>| DMGood (that is, 0) | The operation was successful. |</p>
<table>
<thead>
<tr>
<th>DMNotAuth</th>
<th>The coprocessor application is not authorized to perform DES operations (for example, because it has not called xcAttachWithCDUOption).</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMBadFlags</td>
<td>An input argument was invalid (for example, pKW is NULL or a buffer has an invalid length).</td>
</tr>
</tbody>
</table>

Refer to *xc_err.h* for a comprehensive list of return codes.
3.4.6 `xcAESKeyUnwrapNIST`  
`xcAESKeyUnwrapNIST` unwraps an AES key wrapped according to the NIST standard (see “NIST formatted buffer” on page 47 for details):

**Function prototype**

```c
int xcaESKeyUnwrapNIST(int fd, xcaESKUW_NIST_t *pKUW);
```

**Input**

On entry to this routine:

- `fd` is the file descriptor returned when `/dev/skch` was opened.

- `pKUW` contains the address of an AES key unwrap request block whose fields are initialized as follows:
  
  - `pUnwrapKey` points to a buffer containing the unwrapping key (an item of type `xAES_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 `pKUW` 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`.

Neither the AES key unwrap request block referenced by `pKUW` nor the buffers referenced by `pKUW->pClearKey` and `pKUW->pClearKeyLen` may reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.
**Return codes**
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DMGood</strong></td>
<td>(that is, 0) The operation was successful.</td>
</tr>
<tr>
<td><strong>DMNotAuth</strong></td>
<td>The coprocessor application is not authorized to perform DES operations (for example, because it has not called xcAttachWithCDUOption).</td>
</tr>
<tr>
<td><strong>DMBadFlags</strong></td>
<td>An input argument was invalid (for example, pKUW is NULL or a buffer has an invalid length).</td>
</tr>
<tr>
<td><strong>DMBadNISTKUW</strong></td>
<td>Unwrap operation did not produce a valid NIST formatted buffer.</td>
</tr>
</tbody>
</table>

Refer to *xc_err.h* for a comprehensive list of return codes.
3.5 DES 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 and must open /dev/skch with the O_RDWR flag 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

\[
\text{unsigned int} \; \text{xcTDES}(\text{int} \; \text{fd}, \; \text{xctDES_RB_t} \; *\text{pTDES_rb});
\]

Input

On entry to this routine:

- \text{fd} is the file descriptor returned when /dev/skch was opened.
- \text{pTDES_rb} contains the address of a TDES request block whose fields are initialized as follows:
- \text{options} controls the operation of the function and must be set to the logical OR of constants from the following categories:

<table>
<thead>
<tr>
<th>\text{options}</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES_ENCRYPT</td>
<td>Encrypt the input.</td>
</tr>
<tr>
<td>DES_DECRYPT</td>
<td>Decrypt the input.</td>
</tr>
<tr>
<td>DES_MAC</td>
<td>Generate a message authentication code (MAC) for the input.</td>
</tr>
</tbody>
</table>

If \text{DES_MAC} is specified, \text{DES_ECB_MODE} must not be specified. \text{DES_CBC_MODE} will be assumed.

Chaining mode

\text{options} must include exactly one of the following constants:

<table>
<thead>
<tr>
<th>\text{options}</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES_CBC_MODE</td>
<td>Use outer Cipher Block Chaining (CBC) mode.</td>
</tr>
</tbody>
</table>
**DES_ECB_MODE** must not be specified if **DES_MAC** is specified. **DES_CBC_MODE** will be assumed.

**Padding options**

*options* may include a subset of the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DES_PREPAD</strong></td>
<td>Prepadd the input with 8 bytes of data.</td>
</tr>
<tr>
<td><strong>DES_PAD_WITH_8</strong></td>
<td>Pad the input with 8 bytes of data.</td>
</tr>
<tr>
<td><strong>DES_PAD_WITH_16</strong></td>
<td>Pad the input with 16 bytes of data.</td>
</tr>
</tbody>
</table>

Only one of **DES_PAD_WITH_8** and **DES_PAD_WITH_16** may be specified.

- `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, `options` must specify **DES_PAD_WITH_8** or **DES_PAD_WITH_16** or both **DES_MAC** and **DES_PREPAD**.
- `source.data_ptr` 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 `destination.data_ptr`. `destination_length` must be at least as large as the length of the input data.
- `destination.data_ptr` points to a writeable buffer.
- `prePadding` contains 8 bytes of data that are prepended to the data to be processed by `xcTDES` before the operation is performed if `options` specifies **DES_PREPAD**.
  
  If `options` does not specify **DES_PREPAD**, `prePadding` is not used.
- `postPadding` contains up to 16 bytes of data that may be appended to the data to be processed by `xcTDES` before the operation is performed.
  
  If `options` specifies **DES_PAD_WITH_8**, `postPadding[0]` through `postPadding[7]` are appended to the input. If `options` specifies **DES_PAD_WITH_16**, the entire `postPadding` buffer is appended to the input.
  
  If `options` specifies neither **DES_PAD_WITH_8** nor **DES_PAD_WITH_16**, `postPadding` is not used.
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`.
  - the initialization vector for the next DES operation if options specifies `DES_CBC_MODE`.
  `term_v` is undefined otherwise.

- The buffer referenced by `destination.data_ptr` contains
  - The input data processed using `key1`, `key2`, and `key3` and `init_v` if options specifies `DES_CBC_MODE` if options specifies `DES_ENCRYPT`.
  - The input data processed using `key1`, `key2`, and `key3` and `init_v` if options specifies `DES_CBC_MODE` if options specifies `DES_DECRYPT`.

The buffer referenced by `destination.data_ptr` 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`.

If `source.data_ptr` and/or `destination.data_ptr` point to a DMA-eligible buffer the DES operation may complete more quickly than would otherwise be the case. Neither the TDES request block referenced by `pTDES_rb` nor the buffers referenced by `pTDES_rb->term_v` and `pTDES_rb->destination_ptr` may reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

The buffers defined by `source.data_ptr/source_length` and `destination.data_ptr/destination_length` should not overlap.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMGood (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>DMNotAuth</td>
<td>The coprocessor application is not authorized to perform DES operations (e.g., because it has not called <code>xcAttachWithCDUOption</code>).</td>
</tr>
<tr>
<td>DMBadFlags</td>
<td>The options argument is not valid.</td>
</tr>
<tr>
<td>DMBadParm</td>
<td>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 <code>source_length</code> is zero and no padding option was specified.</td>
</tr>
<tr>
<td>DMBadAddr</td>
<td>Part of the buffer defined by <code>source.data_ptr</code> and <code>source_length</code></td>
</tr>
</tbody>
</table>

3 If `options` specifies both `DES_MAC` and `DES_CBC_MODE`, the MAC and the initialization vector are the same value.
is not readable, part of the buffer defined by `destination.data_ptr` 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

```c
unsigned int xcDES(int fd,
                   xcDES_RB_t *pDES_rb);
```

Input

On entry to this routine:

- `fd` is the file descriptor returned when `/dev/skch` was opened.
- `pDES_rb` contains the address of a DES request block whose fields are initialized as follows:
  - `options` controls the operation of the function and must be set to the logical OR of constants from the following categories:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES_ENCRYPT</td>
<td>Encrypt the input.</td>
</tr>
<tr>
<td>DES_DECRYPT</td>
<td>Decrypt the input.</td>
</tr>
<tr>
<td>DES_MAC</td>
<td>Generate a message authentication code (MAC) for the input.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES_CBC_MODE</td>
<td>Use outer Cipher Block Chaining (CBC) mode.</td>
</tr>
<tr>
<td>DES_ECB_MODE</td>
<td>Use Electronic Code Book (ECB) mode.</td>
</tr>
</tbody>
</table>

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

Padding options

- `options` may include a subset of the following constants:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES_PREPAD</td>
<td>Prepad the input with 8 bytes of data.</td>
</tr>
<tr>
<td>DES_PAD_WITH_8</td>
<td>Pad the input with 8 bytes of data.</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>DES_PAD_WITH_16</td>
<td>Pad the input with 16 bytes of data.</td>
</tr>
</tbody>
</table>

Only one of DES_PAD_WITH_8 and DES_PAD_WITH_16 may be specified.

- `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 xcDES. `source_length` must be a multiple of 8. If `source_length` is 0, `options` must specify DES_PAD_WITH_8 or DES_PAD_WITH_16 or both DES_MAC and DES_PREPAD.
- `source.data_ptr` points to a buffer containing the data to be processed by xcDES. The “input data” to xcDES 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 `destination.data_ptr`. `destination_length` must be at least as large as the length of the input data.
- `destination.data_ptr` points to a writeable buffer.
- `prePadding` contains 8 bytes of data that are prepended to the data to be processed by xcDES before the operation is performed if `options` specifies DES_PREPAD.
  
  If `options` does not specify DES_PREPAD, `prePadding` is not used.
- `postPadding` contains up to 16 bytes of data that may be appended to the data to be processed by xcDES before the operation is performed.
  
  If `options` specifies DES_PAD_WITH_8, `postPadding[0]` through `postPadding[7]` are appended to the input. If `options` specifies DES_PAD_WITH_16, the entire `postPadding` buffer is appended to the input.
  
  If `options` specifies neither DES_PAD_WITH_8 nor DES_PAD_WITH_16, `postPadding` is not used.

**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.

  `term_v` is undefined otherwise.
- The buffer referenced by `destination.data_ptr` contains
  
  - The input data encrypted using `key` (and `init_v` if `options` specifies DES_CBC_MODE) if `options` specifies DES_CBC_MODE.

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

---

If `options` specifies both DES_MAC and DES_CBC_MODE, the MAC and the initialization vector are the same value.
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 destination.data_ptr 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 source.data_ptr and/or destination.data_ptr point to a DMA-eligible buffer the DES operation may complete more quickly than would otherwise be the case. Neither the DES request block referenced by pDES_rb nor the buffers referenced by pDES_rb->term_v and pDES_rb->destination_ptr may reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

The buffers defined by source.data_ptr/source_length and destination.data_ptr/destination_length should not overlap.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMGood</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>DMNotAuth</td>
<td>The coprocessor application is not authorized to perform DES operations (e.g., because it has not called xcAttachWithCDUOption).</td>
</tr>
<tr>
<td>DMBadFlags</td>
<td>The options argument is not valid.</td>
</tr>
<tr>
<td>DMBadParm</td>
<td>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.</td>
</tr>
<tr>
<td>DMBadAddr</td>
<td>Part of the buffer defined by source.data_ptr and source_length is not readable, part of the buffer defined by destination.data_ptr and destination_length is not writeable, or a field in the request block cannot be accessed.</td>
</tr>
</tbody>
</table>

Refer to xc_err.h for a comprehensive list of return codes.
3.5.3  xcDES8bytes - eight-byte DES

xcDES8bytes enciphers or deciphers eight bytes of data using the DES (Data Encryption Standard) algorithm in Electronic Code Book (ECB) mode and a single-length DES key.

Function prototype

```c
unsigned int xcDES8bytes(int fd,
                         xcdES8bytes_RB_t *pDES8bytes_rb);
```

Input

On entry to this routine:

- `fd` is the file descriptor returned when `/dev/skch` was opened.
- `pDES8bytes_rb` contains the address of a DES request block whose fields are initialized as follows:
  - `options` controls the operation of the function and must be set to DES_ENCRYPT (encrypt the input) or DES_DECRYPT (decrypt the input).
  - `key` contains the key to use for the operation.
  - `input_data` contains the data to be processed by xcDES8bytes.

Output

On successful exit from this routine:

The following fields of `*pDES8bytes_rb` are changed as noted:

- `output_data` contains
  - `input_data` encrypted using `key` if `options` specifies DES_ENCRYPT.
  - `input_data` decrypted using `key` if `options` specifies DES_DECRYPT.

Notes

The DES request block referenced by `pDES8bytes_rb` must not reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMGood</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>DMNotAuth</td>
<td>The coprocessor application is not authorized to perform DES operations (e.g., because it has not called xcAttachWithCDUOption).</td>
</tr>
<tr>
<td>DMBadFlags</td>
<td>The options argument is not valid.</td>
</tr>
<tr>
<td>DMBadAddr</td>
<td>A field in the request block cannot be accessed.</td>
</tr>
</tbody>
</table>

Refer to `xc_err.h` for a comprehensive list of return codes.
3.5.4 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**

```c
unsigned int xcDES3Key(int             fd,
                        xcDES3Key_RB_t  *pDES3Key_rb);
```

**Input**

On entry to this routine:

- *fd* is the file descriptor returned when `/dev/skch` was opened.
- *pDES3Key_rb* contains the address of a DES request block whose fields are initialized as follows:
  - *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* if *options* specifies `DES_ENCRYPT`.
  - *key_in* processed using *key1*, *key2*, and *key3* if *options* specifies `DES_DECRYPT`.

**Notes**

The DES request block referenced by *pDES3Key_rb* must not reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DMGood</code> (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><code>DMNotAuth</code></td>
<td>The coprocessor application is not authorized to perform DES operations (e.g., because it has not called <code>xcAttachWithCDUOption</code>).</td>
</tr>
<tr>
<td><strong>DMBadFlags</strong></td>
<td>The options argument is not valid.</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td><strong>DMBadAddr</strong></td>
<td>A field in the request block cannot be accessed.</td>
</tr>
</tbody>
</table>

Refer to `xc_err.h` for a comprehensive list of return codes.
3.6 Public key algorithm 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 Digital Signature Algorithm (DSA).

A coprocessor application must call `xcAttachWithCDUOption` and must open `/dev/pka` with the `O_RDWR` flag before calling any of the functions in this section.

3.6.1 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:

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 2048.
  
  If `type` is `RSA_X931_PRIVATE_MODULUS_EXPONENT`, `n_BitLength` must be 1024, 1280, 1536, 1792, or 2048.
- `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.
- \( x.d\_Length \) is the length in bytes of the private exponent \( d \).
- \( y.d\_Ptr \) points to the first (most significant) byte of the private exponent \( d \).
- \( r\_Length \) is the length in bytes of the blinding value \( r \).
- \( r\_Ptr \) points to the first (most significant) byte of the blinding value \( r \).
- \( r1\_Length \) is the length in bytes of the blinding value \( r^{-1} \), which is the inverse of \( r \) modulo \( n \).
- \( r1\_Ptr \) 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+256\( k \) 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 \( p \).
- \( dp\_Length \) is the length in bytes of \( dp = d \mod (p-1) \), where \( d \) is the private exponent.
- \( dp\_Ptr \) points to the first (most significant) byte of \( dp \).
- \( dq\_Length \) is the length in bytes of \( dq = d \mod (q-1) \), where \( d \) is the private exponent.
- \( dq\_Ptr \) points to the first (most significant) byte of \( dq \).
- \( ap\_Length \) is the length in bytes of \( ap = q^{p^{-1}} \mod n \).
- \( ap\_Ptr \) points to the first (most significant) byte of \( ap \).

---

5 \( d \) is the inverse of the public exponent \( e \) modulo \( (p-1)(q-1) \)
6 \( r = R^p \mod n \) and \( r^{-1} \) is the inverse of \( r \) modulo \( n \) where \( R \) is a random number less than the modulus \( n \). 
7 \( n = pq \)
• $a_{Length}$ is the length in bytes of $a_{q} = n + 1 - ap$.
• $a_{q}Ptr$ points to the first (most significant) byte of $a_{q}$.
• $r\_Length$ is the length in bytes of the blinding value $r^8$.
• $r\_Ptr$ points to the first (most significant) byte of the blinding value $r$.
• $r1\_Length$ is the length in bytes of the blinding value $r^{-1}$, which is the inverse of $r$ modulo $n$.
• $r1\_Ptr$ 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).
• $token\_Length$ is the length in bytes of the key token.
• $n\_Bit\_Length$ is the length in bits of the modulus $n$. $n\_Bit\_Length$ cannot exceed 4096.
  If $type$ is $RSA\_X931\_PRIVATE\_CHINESE\_REMAINDER$, $n\_Bit\_Length$ 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.
• $dp\_Ptr$ points to the first (most significant) byte of $dp$.
• $dq\_Length$ is the length in bytes of $dq = d \mod (q-1)$, where $d$ is the private exponent.
• $dq\_Ptr$ points to the first (most significant) byte of $dq$.
• $qInv\_Length$ is the length in bytes of $q^{-1} \mod p$.
• $qInv\_Ptr$ points to the first (most significant) byte of $q^{-1} \mod p$.
• $not\_Defined1$ and $not\_Defined2$ are reserved and should be set to zero.

---

8 $r = R^e \mod n$ and $r^{-1}$ is the inverse of $r$ modulo $n$ where $R$ is a random number less than the modulus $n$.
9 $n = pq$
• $r_{\text{Length}}$ is the length in bytes of the blinding value $r^{10}$.
• $r_{\text{Ptr}}$ points to the first (most significant) byte of the blinding value $r$.
• $r1\text{Length}$ is the length in bytes of the blinding value $r^{-1}$, which is the inverse of $r$ modulo $n$.
• $r1\text{Ptr}$ points to the first (most significant) byte of the blinding value $r^{-1}$.

Use of a private key of type \texttt{RSA\_PRIVATE\_CHINESE\_REMAINDER}, \texttt{RSA\_X931\_PRIVATE\_CHINESE\_REMAINDER}, \texttt{RSA\_PKCS\_PRIVATE\_CHINESE\_REMAINDER}, or \texttt{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 $n_{\text{BitLength}}$ 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 $n_{\text{BitLength}}$ is 4096, $e_{\text{Length}}$ should not be greater than 3.

\footnote{10 \( r = R^e \mod n \) and $r^{-1}$ is the inverse of $r$ modulo $n$ where $R$ is a random number less than the modulus $n$.}
3.6.2 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**

```c
unsigned int xcRSAKeyGenerate(unsigned int      fd,
                               xcRSAKeyGen_RB_t *pRSAKeyGen_rb);
```

**Input**

On entry to this routine:

- `fd` is the file descriptor returned when `/dev/pka` was opened.
- `pRSAKeyGen_rb` contains the address of a RSA key generation request block whose fields are initialized as follows:
  - `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`
    - `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 2048 if `key_type` is `RSA_PRIVATE_MODULUS_EXPONENT` or `RSA_X931_PRIVATE_MODULUS_EXPONENT`, and must be less than or equal to 4096 otherwise.
    - 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:

<table>
<thead>
<tr>
<th><code>RSA_EXPONENT_ randomness</code></th>
<th>Choose a pseudo-random number containing <code>mod_size</code> bits that meets the standards described in ANSI X9.31</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RSA_EXPONENT_FIXED</code></td>
<td>Use the value of <code>e</code> in the RSA key token referenced by <code>key_token</code></td>
</tr>
<tr>
<td><code>RSA_EXPONENT_2</code></td>
<td><code>Set e = 2</code></td>
</tr>
<tr>
<td><code>RSA_EXPONENT_3</code></td>
<td><code>Set e = 3</code></td>
</tr>
<tr>
<td><code>RSA_EXPONENT_65537</code></td>
<td><code>Set e = 65537 (0x10001)</code></td>
</tr>
</tbody>
</table>


**RSA_EXPONENT_RANDOM** should not be used with 4096-bit keys, because the key generation will generally take longer than the device driver timeout allows for a coprocessor call.

**RSA_EXPONENT_3** and **RSA_EXPONENT_65537** provide support for certain standards that require specific public exponents (for example, SET). 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 \( RSA\_PKCS\_\ast \), \( apLength \) must be greater than or equal to \( n\_Length \). If \( pRSAKeyGen_rb->key_type \) is \( RSA\_PKCS\_\ast \), \( apLength \) 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 \( RSA\_PKCS\_\ast \), \( aqLength \) must be greater than or equal to \( n\_Length \). If \( pRSAKeyGen_rb->key_type \) is \( RSA\_PKCS\_\ast \), \( aqLength \) 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 \^{}\ast{}^{}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. \^{}\ast{}^{}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 101-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.

Output

On successful exit from this routine:

The following fields of \^{}\ast{}^{}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 \( dp = d \mod (p-1) \).
- The buffer defined by dqPtr/dqLength contains \( dq = d \mod (q-1) \).
- The buffer defined by apPtr/apLength contains
  - \( ap = q^{p-1} \mod n \) if key_token->type is not RSA_PKCS_ or
  - \( q^{-1} \mod p \) 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.
- 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.

Neither the RSA key generation request block referenced by pRSAKeyGen_rb nor the key token referenced by pRSAKeyGen_rb nor any buffer referenced by a pointer in pRSAKeyGen_rb->key_token (for example, pRSAKeyGen_rb->keytoken->n_ptr) may reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

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. |

76 IBM 4765 Custom Software Interface Reference
| 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.

3.6.3
### 3.6.4 xcrSA - encipher/decipher data or wrap/unwrap X9.31 encapsulated hash

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

**Function prototype**

```c
unsigned int xcrRSA(unsigned int fd,
                    xcrRSA_RB_t  *pRSA_rb);
```

**Input**

On entry to this routine:

- `fd` is the file descriptor returned when `/dev/pka` was opened.
- `pRSA_rb` contains the address of a RSA request block whose fields are initialized as follows:
  - `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:

<table>
<thead>
<tr>
<th>Constants</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA_PUBLIC</td>
<td>Perform the operation using the public key from the key token (that is, <code>output = input^e mod n</code>).</td>
</tr>
<tr>
<td>RSA_PRIVATE</td>
<td>Perform the operation using the private key from the key token (for example, <code>output = input^d mod n</code>).</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA_DONT_BLIND</td>
<td>Perform the operation without using the blinding values.</td>
</tr>
<tr>
<td>RSA_BLIND_NO_UPDATE</td>
<td>Perform the operation using the blinding values and replace the blinding values in the key token.</td>
</tr>
<tr>
<td>RSA_BLIND_UPDATE</td>
<td>Perform the operation using the blinding values but do not replace the blinding values in the key token.</td>
</tr>
</tbody>
</table>

The names RSA_BLIND_NO_UPDATE and RSA_BLIND_UPDATE are somewhat confusing. RSA_BLIND_NO_UPDATE means the caller is not going to replace the blinding values (so the PKA driver does so). RSA_BLIND_UPDATE means the caller is going to replace the blinding values (so the PKA driver refrains from doing so).

RSA_DONT_BLIND yields the best performance. RSA_BLIND_NO_UPDATE requires the most time. RSA_BLIND_UPDATE uses a fast blinding scheme but assumes the coprocessor application replaces the blinding values in the key token (for example, by calling xcComputeBlindingValues) before calling xcRSA again.

RSA_BLIND_NO_UPDATE is the default.

**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,

```c
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.

**Mapped kernel buffers**

Neither the RSA request block referenced by `pRSA_rb` nor the buffer referenced by `pRSA_rb->data_out` may reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKAGood</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>PKABadAddr</td>
<td>A pointer in the request block or the key token is invalid.</td>
</tr>
<tr>
<td>PKABadParm</td>
<td>An argument is not valid. Many structural deficiencies in the request block or key token can generate this error.</td>
</tr>
<tr>
<td>PKANoSpace</td>
<td>The operation failed due to lack of space (for example, there is no free memory available to the PKA driver).</td>
</tr>
<tr>
<td>PKARangeOverflow</td>
<td>The last <code>pRSA_rb-&gt;key_token-&gt;n_Length</code> bytes of the buffer described by <code>pRSA_rb-&gt;data_in</code>, when interpreted as a big-</td>
</tr>
</tbody>
</table>
endian integer, exceed the value of the modulus $n$.

Refer to $xc_{-err}.h$ for a comprehensive list of return codes.
3.6.5 xcComputeBlindingValues - compute blinding values for RSA key

In some implementations of the RSA algorithm, an adversary can use differences in the amount of time it takes to process various messages with a particular private key to defeat the cryptographic security provided by the key. This is called a timing attack. The PCIe Cryptographic Coprocessor’s RSA implementation is not vulnerable to timing attacks. However, in order to allow the keys generated by this system to be used by other systems, an application can create “blinding values”, two large integers that can be used by other systems to defeat timing attacks.

xcComputeBlindingValues generates blinding values for an RSA key.

Function prototype

```c
unsigned int xcComputeBlindingValues(int         fd,
                                      xcCBV_RB_t *pcvb_rb);
```

Input

On entry to this routine:

`fd` is the file descriptor returned when `/dev/pka` was opened.

`pcvb_rb` contains the address of a blinding value request block whose fields are initialized as follows:

- `n` points to a buffer containing the modulus \( n \).
- `nsize` is the length in bits of the modulus \( n \).
  
  If `nsize` is not a multiple of eight, any excess high-order bits in the modulus \( n \) are treated as zeros (that is, \( n \) is essentially padded on the left with zeros, regardless of the actual bits that appear in the buffer referenced by \( n \)).

- `e` points to a buffer containing the public exponent \( e \).
- `esize` is the length in bytes of the public exponent \( e \).
  
  The public exponent \( e \) must have the same length in bits as the modulus \( n \), so `esize` must be \((nsize + 7)/8\).

  If `nsize` is not a multiple of eight, any excess high-order bits in the public exponent \( e \) are treated as zeros (that is, \( e \) is essentially padded on the left with zeros, regardless of the actual bits that appear in the buffer referenced by \( e \)).

  If the length of \( e \) in bits is not a multiple of eight, any excess high-order bits in the public exponent are treated as zeros (that is, \( e \) is essentially padded on the left with zeros, regardless of the actual bits that appear in the buffer).

- `r_e` and `rinv` point to writeable buffers, each of which is at least `esize` bytes in length.

Output

On successful exit from this routine:

The buffer defined by `pcvb_rb->r_e/pcvb_rb->esize` contains the blinding value \( r \) generated by the call.

The buffer defined by `pcvb_rb->rinv/pcvb_rb->esize` contains inverse of the blinding value \( r^{-1} \) generated by the call.
Notes
The modulus, public exponent, and blinding values are stored in big-endian order: the byte at the lowest address contains the most significant byte of the value.

Neither the buffer referenced by `pcvb_rb->r_e` nor the buffer referenced by `pcvb_rb->rinv` may reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKAGood (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>PKABadParm</td>
<td>An argument is not valid.</td>
</tr>
<tr>
<td>PKANoSpace</td>
<td>The buffer referenced by <code>pcvb_rb-&gt;r_e</code> or <code>pcvb_rb-&gt;rinv</code> is not large enough to hold the updated blinding values generated by the call.</td>
</tr>
</tbody>
</table>

Refer to `xc_err.h` for a comprehensive list of return codes.
3.6.6 DSA key tokens

The PKA interface defines the `xcDSAKeyToken_t` type to hold information about DSA public and private keys. An item of type `xcDSAKeyToken_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 a DSA public key includes the large prime \( p \), the small prime \( q \), the generator \( g \), and the public exponent \( y \).) 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 a DSA public key are set as follows:

- `type` is `DSA_PUBLIC_KEY_TYPE`.
- `key_token_length` is the length in bytes of the key token.
- `prime_p_bit_length` is the length in bits of the large prime \( p \). `prime_p_bit_length` must be a multiple of 64 between 512 and 1024, inclusive.
- `p_Length` is the length in bytes of the large prime \( p \).
- `p_Ptr` points to the first (most significant) byte of the large prime \( p \).
- `q_Length` is the length in bytes of the small prime \( q \). `q_Length` must be 20.
- `q_Ptr` points to the first (most significant) byte of the small prime \( q \).
- `g_Length` is the length in bytes of the generator \( g \).
- `g_Ptr` points to the first (most significant) byte of the generator \( g \).
- `y_Length` is the length in bytes of the public exponent \( y \).
- `y_Ptr` points to the first (most significant) byte of the public exponent \( y \).

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

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

- `type` is `DSA_PRIVATE_KEY_TYPE`.
- `key_token_length` is the length in bytes of the key token.
- `prime_p_bit_length`, `p_Length`, `p_Ptr`, `q_Length`, `q_Ptr`, `g_Length`, `g_Ptr`, `y_Length`, and `y_Ptr` are set in the same manner as for a DSA public key.
- `x_Length` is the length in bytes of the private exponent \( x \).
- `x_Ptr` points to the first (most significant) byte of the private exponent \( x \).

Note that a DSA private key token includes information about the corresponding DSA public exponent. The public exponent need not be present when the token is used as a private key.

\[ g = h^{p/q} \mod p \] where \( h \) is a number less than \( p-1 \) chosen at random so that \( g \) is greater than 1.

\[ y = g^x \mod p, \] where \( x \) is the private exponent corresponding to the DSA public key defined by \( p, q, g, \) and \( y \).

\[ x \] is a random integer between 1 and \( q-1 \), inclusive.
3.6.7 DSA signature tokens

The PKA interface defines the `xcDSASignatureToken_t` type to hold a digital signature generated by the DSA algorithm. An item of type `xcDSASignatureToken_t` consists of a descriptive header followed by a buffer containing information about the values of the various elements of the signature. The header 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 a DSA signature token are set as follows:

- `signature_token_length` is the length in bytes of the signature token.
- `r_length` is the length in bytes of the first half of the signature r. `r_length` must be less than or equal to 20.
- `r_Ptr` points to the first (most significant) byte of the first half of the signature r.
- `s_length` is the length in bytes of the second half of the signature s. `s_length` must be less than or equal to 20.
- `s_Ptr` points to the first (most significant) byte of the second half of the signature s.

\[ r = (g^k \mod p) \mod q \text{ where } k \text{ is a number less than } q \text{ chosen at random.} \]

\[ s = (H(m) + xr) / k \mod q \text{ where } H(m) \text{ is the value generated by the Secure Hash Algorithm when applied to the data to be signed.} \]
3.6.8 xcDSAKeyGenerate - generate DSA keypair

xcDSAKeyGenerate generates a key token for a DSA private key. The token includes information that defines the corresponding DSA public key. The user may specify values for certain portions of the DSA private key.

**Function prototype**

```c
unsigned int xcDSAKeyGenerate(unsigned int fd,
                            xcDSAKeyGen_RB_t *pDSAKeyGen_rb);
```

**Input**

On entry to this routine:

- `fd` is the file descriptor returned when `/dev/pka` was opened.
- `pDSAKeyGen_rb` contains the address of a DSA key generation request block whose fields are initialized as follows:

  - `prime_p_size` specifies the desired length in bits of the large prime `p`. `prime_p_size` must be a multiple of 64 between 512 and 1024, inclusive.
  - `key_token_size` contains the length in bytes of the buffer referenced by `key_token`. `key_token_size` must be greater than or equal to `sizeof(xcDSAKeyToken_t)`.
  - `key_token` points to a writeable buffer in which an item of type `xcDSAKeyToken_t` can be stored. The fields of `key_token` must be initialized as follows:

    - `xcDSAKeyGenerate` normally generates `p`, `q`, and `g` randomly. If `options` specifies `DSA_PARTIAL_KEYGEN`, the values of `p`, `q`, and `g` are instead taken from the appropriate fields of `key_token`.
    - `p_length` contains the length in bytes of the buffer referenced by `p_Ptr`. `p_length` must be greater than or equal to `(prime_p_size + 7)/8`. If `options` specifies `DSA_PARTIAL_KEYGEN`, `p_length` must be equal to `(prime_p_size + 7)/8`.
    - `q_length` contains the length in bytes of the buffer referenced by `q_Ptr`. `q_length` must be greater than or equal to 20. If `options` specifies `DSA_PARTIAL_KEYGEN`, `q_length` must be 20.
    - `g_length` contains the length in bytes of the buffer referenced by `g_Ptr`. `g_length` must be greater than or equal to `(prime_p_size + 7)/8`. If `options` specifies `DSA_PARTIAL_KEYGEN`, `g_length` must be equal to `(prime_p_size + 7)/8`.
    - `y_length` contains the length in bytes of the buffer referenced by `y_Ptr`. `y_length` must be greater than or equal to `(prime_p_size + 7)/8`.
    - `x_length` must contain the length in bytes of the buffer referenced by `x_Ptr`. `x_length` must be greater than or equal to 20.
• p_Ptr, q_Ptr, g_Ptr, y_Ptr, and x_Ptr must point to writeable buffers whose lengths are given by the corresponding *_length fields.

If options specifies **DSA_PARTIAL_KEYGEN**,
• the buffer defined by p_Ptr/p_Length contains the value to be used for the large prime p,
• the buffer defined by q_Ptr/q_Length contains the value to be used for the small prime q, and
• the buffer defined by g_Ptr/g_Length contains the value to be used for the generator g.
• random_seed_size is the length in bytes of the buffer referenced by random_seed. If random_seed is NULL, random_seed_size must be 0.

If options specifies **DSA_PARTIAL_KEYGEN**, random_seed_size is not used.
• random_seed may be NULL.

If random_seed is not NULL, it points to a buffer containing a string of bits used to seed the PKA driver pseudo-random number generator, which in turn generates p, q, and g\(^{16}\). The strings of bits should contain at least 160 bits of entropy.

If options specifies **DSA_PARTIAL_KEYGEN**, random_seed is not used.

**Output**
On successful exit from this routine, the following fields of *(pDSAKeyGen_rb->key_token) are changed as noted:
• key_type is set to **DSA_PRIVATE_KEY_TYPE**
• key_token_length is set to sizeof(xcDSAKeyToken_t)
• prime_p_bit_length is set to pDSAKeyGen_rb->prime_p_size
• p_Length is set to the length in bytes of the large prime p and the buffer defined by p_Ptr/p_Length contains the large prime p.

If pDSAKeyGen_rb->options specifies **DSA_PARTIAL_KEYGEN**, this represents no change from the values of p_Length and the buffer defined by p_Ptr/p_Length on entry to the routine.
• q_Length is set to the length in bytes of the small prime q and the buffer defined by q_Ptr/q_Length contains the small prime q.

If pDSAKeyGen_rb->options specifies **DSA_PARTIAL_KEYGEN**, this represents no change from the values of q_Length and the buffer defined by q_Ptr/q_Length on entry to the routine.
• g_Length is set to the length in bytes of the generator g and the buffer defined by g_Ptr/g_Length contains the generator g.

If pDSAKeyGen_rb->options specifies **DSA_PARTIAL_KEYGEN**, this represents no change from the values of g_Length and the buffer defined by g_Ptr/g_Length on entry to the routine.
• y_Length is set to the length in bytes of the public exponent y and the buffer defined by y_Ptr/y_Length contains the public exponent y.
• x_Length is set to the length in bytes of the private exponent x and the buffer defined by

16 x is always generated randomly, so use of random_seed does not ensure reproducible results...
$x_{\text{Ptr}}/x_{\text{Length}}$ contains the private exponent $x$.

Notes
A key token for a DSA public key can be generated from the key token for the corresponding DSA private key by copying all the buffers in the DSA private key except the buffer defined by $x_{\text{Ptr}}/x_{\text{Length}}$, copying the $\text{prime}_p_{\text{bit}}_{\text{length}}$ field, and setting the public key token's type field to $\text{DSA}_{\text{PUBLIC KEY}}_{\text{TYPE}}$.

Neither the DSA key generation request block referenced by $p\text{DSAKeyGen}_{\text{rb}}$ nor the key token referenced by $p\text{DSAKeyGen}_{\text{rb}}->key_token$ nor any buffer referenced by a pointer in $p\text{DSAKeyGen}_{\text{rb}}->key_token$ (for example, $p\text{DSAKeyGen}_{\text{rb}}->keytoken->p_{\text{ptr}}$) may reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

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:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKAGood (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>PKADSAKeyGenFailed</td>
<td>$p\text{DSAKeyGen}<em>{\text{rb}}-&gt;prime_p</em>{size}$ is not a multiple of 64</td>
</tr>
<tr>
<td></td>
<td>between 512 and 1024, inclusive.</td>
</tr>
<tr>
<td>PKABadParm</td>
<td>An argument is not valid.</td>
</tr>
<tr>
<td>PKANoSpace</td>
<td>The buffer referenced by $p\text{DSAKeyGen}_{\text{rb}}-&gt;key_token$ is not</td>
</tr>
<tr>
<td></td>
<td>large enough to hold the token generated by the call.</td>
</tr>
</tbody>
</table>

Refer to $xc_{\text{err}.h}$ for a comprehensive list of return codes.
3.6.9 xcdSA - sign data or verify signature for data

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

Function prototype

```c
unsigned int xcdSA(unsigned int fd,
                    xcdSA_RB_t *pDSA_rb);
```

Input

On entry to this routine:

- `fd` is the file descriptor returned when /dev/pka was opened.
- `pDSA_rb` contains the address of a DSA request block whose fields are initialized as follows:
  
  - `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:

    | Constant                          | Description                                                      |
    |-----------------------------------|------------------------------------------------------------------|
    | DSA_SIGNATURE_SIGN                | Compute the DSA signature for the input data.                    |
    | DSA_SIGNATURE_VERIFY              | Verify that the signature for the input data is correct.        |

    If `DSA_SIGNATURE_SIGN` is specified, `pDSA_rb->key_token->key_type` must be `DSA_PRIVATE_KEY_TYPE`. If `DSA_SIGNATURE_VERIFY` is specified, `pDSA_rb->key_token->key_type` must be `DSA_PUBLIC_KEY_TYPE`.

  - `key_token` points to a buffer containing the DSA key token for the key to be used in the operation.
    
    If `options` specifies `DSA_SIGNATURE_SIGN`, `key_token` must be the token for a DSA private key (e.g., `key_token->key_type` must be `DSA_PRIVATE_KEY_TYPE`).
    
    If `options` specifies `DSA_SIGNATURE_VERIFY`, `key_token` must be the token for a DSA public key (e.g., `key_token->key_type` must be `DSA_PUBLIC_KEY_TYPE`).
    
    - `key_token_size` is the length in bytes of the buffer referenced by `key_token` (i.e., `key_token_size` equals `key_token->key_token_length`).
    
    - `sig_token` points to a buffer in which an item of type `xcDSASignatureToken_t` can be stored.
    
    If `options` specifies `DSA_SIGNATURE_SIGN`, the buffer must be writeable. In this case, the buffers defined by `sig_token->s_Ptr/sig_token->s_length` and `sig_token->r_Ptr/sig_token->r_length` must be writeable and at least 20 bytes long.
    
    If `options` specifies `DSA_SIGNATURE_VERIFY`, the buffer defines the signature that is to be verified (and
the buffers defined by \texttt{sig\_token->s\_Ptr/sig\_token->s\_length} and \texttt{sig\_token->r\_Ptr/sig\_token->r\_length} contain the signature itself).

- \textit{sig\_token\_size} is the length in bytes of the buffer referenced by \texttt{sig\_token}. \textit{sig\_token\_size} must be greater than or equal to \texttt{sizeof(xcDSASignatureToken\_t)}.
- \textit{data} points to a buffer that contains the input data (i.e., a SHA-1 hash if options specifies \texttt{DSA\_PRE\_DIGESTED\_DATA} and an arbitrary block of data otherwise).
- \textit{data\_size} is the length in bytes of the buffer referenced by data.

**Output**

On successful exit from this routine,

- If \textit{options} specifies \texttt{DSA\_SIGNATURE\_VERIFY}, \texttt{xcDSA} returns \texttt{PKAGood} if the signature verifies and \texttt{PKADSASigIncorrect} if the signature does not verify.
- If \textit{options} specifies \texttt{DSA\_SIGNATURE\_SIGN}, \texttt{xcDSA} returns \texttt{PKAGood} and the buffer defined by \texttt{sig\_token/sig\_token\_size} contains the requested signature. In this case, \texttt{sig\_token->s\_length} and \texttt{sig\_token->r\_length} are changed to reflect the actual lengths in bytes of the two halves of the signature.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{PKAGood} (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>\textbf{PKADSASigIncorrect}</td>
<td>\texttt{pDSA_rb-&gt;options} specifies \texttt{DSA_SIGNATURE_VERIFY} but the signature does not verify.</td>
</tr>
<tr>
<td>\textbf{PKABadParm}</td>
<td>An argument is not valid.</td>
</tr>
<tr>
<td>\textbf{PKANoSpace}</td>
<td>The operation failed due to lack of space.</td>
</tr>
</tbody>
</table>

Refer to \texttt{xc\_err\_h} for a comprehensive list of return codes.
3.7 Large integer modular math 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 \( C = A \times B \mod N \)
- Modular exponentiation \( C = A^B \mod N \)
- Modular reduction \( C = A \mod N \)

A coprocessor application must call `xcAttachWithCDUOption` and must open `/dev/pka` with the `O_RDWR` flag before calling any of the functions in this section.

3.7.1 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 \((\text{bitsize} + 7)/8\) 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.
3.7.2 xcModMath - perform modular computations

xcModMath performs one of the following operations on large integers:

- Modular multiplication \( (C = A \times B \mod N) \)
- Modular exponentiation \( (C = A^B \mod N) \)
- Modular reduction \( (C = A \mod N) \)

**Function prototype**

```c
unsigned int xcModMath(int              fd,  
                        int              cmd,  
                        int              numints,  
                        xcModMath_Int_t  *intbufs);
```

**Input**

On entry to this routine:

- `fd` is the file descriptor returned when `/dev/pka` was opened.
- `cmd` controls the operation of the function and must be set to the logical OR of constants from the following categories:

<table>
<thead>
<tr>
<th>Operation</th>
<th>cmd</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODM_MULT</td>
<td>MODM_MULT</td>
<td>Compute ( C = A \times B \mod N )</td>
</tr>
<tr>
<td>MODM_EXP</td>
<td>MODM_EXP</td>
<td>Compute ( C = A^B \mod N )</td>
</tr>
<tr>
<td>MODM_MOD</td>
<td>MODM_MOD</td>
<td>( C = A \mod N )</td>
</tr>
</tbody>
</table>

**Operation**

`cmd` must include exactly one of the following constants:

**Large integer byte order**

`cmd` must include MODM_BIG.

`numInts` is the number of elements in the array referenced by `intbufs`. If `cmd` specifies MODM_MULT or MODM_EXP, `numInts` must be at least 4. If `cmd` specifies MODM_MOD, `numInts` must be at least 3.

`intbufs` points to an array of large integer descriptors. Its elements are as follows:

- `intbufs[MODM_C]` is the descriptor for \( C \), the result of the operation. The buffer defined by `intbufs[MODM_C].bytesize` and `intbufs[MODM_C].buffer` must be large enough to hold the result of the operation, i.e., must be as large as the modulus \( N \).
- `intbufs[MODM_C].bitsize` is not used.
- `intbufs[MODM_N]` is the descriptor for \( N \), the modulus.
- `intbufs[MODM_A]` is the descriptor for \( A \), the first (or only) operand.

If `cmd` specifies MODM_MULT or MODM_EXP, 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.
• `intbufs[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`, `intbufs[MODM_B]` is not used.

**Output**

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

**Notes**

The buffer referenced by `intbufs[MODM_C].buffer` must not reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

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:

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

Refer to `xc_err.h` for a comprehensive list of return codes.
3.8 Random number generator functions

The functions described in this section allow a coprocessor application to request services from the Random Number Generator (RNG) 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` and must open `/dev/hwrng` with the `O_RDONLY` flag before calling any of the functions in this section.

3.8.1 xcDRBGinstantiate – instantiate a DRBG random number generator

This function is available in version 4.3 and up.

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 the SHA256 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

```c
int xcDRBGinstantiate(int fd,           // (Input)
                       xcDRBGinstan_t *pDRBGinst);  // (Input/Output)
```

Input

On entry to this routine:

- `fd` is the file descriptor of the `/dev/hwrng` device.
- `pDRBGinst` is a pointer to a structure with fields set as follows:
  
  - `mechanism` is set to the constant `DRBG_HASH_MECH_SHA256`.
  - `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.

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:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RN_Success</strong> (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><strong>RN_Invalid</strong></td>
<td>One of the parameters in the structure pointed to by the <code>pDRBGInst</code> parameter is invalid.</td>
</tr>
<tr>
<td><strong>RN_NotWorking</strong></td>
<td>DRBG not working: DRBG was working when the request was received, but the operation failed.</td>
</tr>
<tr>
<td><strong>RN_None</strong></td>
<td>DRBG was offline when the request was received.</td>
</tr>
</tbody>
</table>

Refer to `xc_err.h` for a comprehensive list of return codes.
3.8.2 xCDBGgenerate – generate random number

This function is available in version 4.3 and up.

The xCDBGgenerate 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;
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);
9. Performs KATs on demand from the xCDBGtest API.

Function prototype

```c
int xCDBGgenerate(int fd,            // (Input)
                   xCDBGgenerate_t *pDRBGgen);     // (Input/Output)
```

Input

On entry to this routine:

- `fd` is the file descriptor of the /dev/hwrng device.
- `pDRBGgen` is a pointer to a structure with fields set as follows:
  - `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 xCDBGInstantiate 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 `DBRG_SSTR_256`, for 256-bit security strength.
  - `predRRq` must be set to one of the following constants:

<table>
<thead>
<tr>
<th>DRBG_PRR_ON</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRBG_PRR_1</td>
<td>1</td>
</tr>
</tbody>
</table>

Reseeding is requested. If the prediction resistance byte in the internal state was set during the xCDBGInstantiate 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 xCDBGInstantiate call, then setting `DRBG_PRR_ON` to 1 will
cause the RN_Invalid return code to be returned

<table>
<thead>
<tr>
<th>DRBG_PRR_OFF</th>
<th>0</th>
<th>Reseeding not requested. The bit generator will not be reseeded UNLESS the previous reseed has reached the end of its life cycle.</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>RANDOM_RANDOM</th>
<th>Generate the requested number of random bytes. Disregard parity.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANDOM_ODD_PARITY</td>
<td>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.</td>
</tr>
<tr>
<td>RANDOM_EVEN_PARITY</td>
<td>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.</td>
</tr>
</tbody>
</table>

- ODD and EVEN parity cannot both be set at once. Doing so will result in a return code of RN_Invalid.

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

<table>
<thead>
<tr>
<th>RANDOM_SW</th>
<th>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 result in a return code of RN_Invalid.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANDOM_NOT_WEAK</td>
<td>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.</td>
</tr>
</tbody>
</table>

- **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:
<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RN_Success</td>
<td>(that is, 0) The operation was successful.</td>
</tr>
<tr>
<td>RN_Invalid</td>
<td>One of the parameters in the structure pointed to by the pDRBGgen parameter is invalid.</td>
</tr>
<tr>
<td>RN_NotWorking</td>
<td>DRBG not working: DRBG was working when the request was received, but the operation failed</td>
</tr>
<tr>
<td>RN_None</td>
<td>DRBG was offline when the request was received.</td>
</tr>
</tbody>
</table>

Refer to xc_err.h for a comprehensive list of return codes.
3.8.3  xcDRBGrseede – reseed an instance of the RNG

This function is available in version 4.3 and up.

The xcDRBGrseede 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 xcDRBGrgenerate 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

```c
int xcDRBGrseede(int             fd,
                  xcDBRGrsede_t *pDBGrsd);
```

Input

On entry to this routine:
- `fd` is the file descriptor of the /dev/hwrng device.
- `pDBGrsd` is a pointer to a structure with fields set as follows:
  - `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 “xcDBRGrseede: 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 xcDBRGr instantiate call in the pDBRGrinstan->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:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RN_Success (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>RN_Invalid</td>
<td>One of the parameters in the structure pointed to by the pDBRGrsd parameter is invalid.</td>
</tr>
<tr>
<td>RN_NotWorking</td>
<td>DRBG not working: DRBG was working when the request was received, but the operation failed</td>
</tr>
<tr>
<td>RN_None</td>
<td>DRBG was offline when the request was received.</td>
</tr>
</tbody>
</table>

Refer to `xc_err.h` for a comprehensive list of return codes.
### 3.8.4 xcDRBGtest – test the RNG functions

This function is available in version 4.3 and up.

The *xcDRBGtest* function tests the RNG instantiate, generate, reseed, and uninstantiate APIs by invoking statically declared `internal_DRBGinstantiate`, `internal_DRBGgenerate`, `internal_DRBGreseed`, and `internal_DRBGuninstantiate`. All other threads that may be running to obtain DRBG output will be blocked during these tests so that only the *xcDRBGtest* operates to obtain pseudo random bits in single threaded mode. A failure of any portion of this test is treated as follows:

1. DRBG enters an error state, and outputs an error message,
2. no further DRBG operations are permitted and no further DRBG output is allowed,
3. user intervention such as card reset or power cycle shall be required to exit the error state, and
4. DRBG re-instantiation must be performed before the DRBG can be used to produce further pseudo random bits.

Some examples of catastrophic errors include: errors deliberately inserted but not detected; incorrect result from the instantiate test, reseed test, or generate test.

#### Function prototype

```c
int xcDRBGtest(int fd);
```

#### Input

On entry to this routine:

- *fd* is the file descriptor of the `/dev/hwrng` device.

#### Output

This function has no outputs. If KATs pass and errors are handled correctly, the tested values of security strength and prediction resistance flag may be used during normal operation. `reqStrength` must be set to the constant `DRBG_SSTR_256`, for 256-bit security strength.

#### Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RN_Success (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>RN_Invalid</td>
<td>One of the parameters in the structure pointed to by the <code>pDRBGinst</code> parameter is invalid.</td>
</tr>
<tr>
<td>RN_NotWorking</td>
<td>DRBG not working: DRBG was working when the request was received, but the operation failed</td>
</tr>
<tr>
<td>RN_None</td>
<td>DRBG was offline when the request was received.</td>
</tr>
<tr>
<td>RN_TestInstantiateFail</td>
<td>DRBG Instatiation test failed.</td>
</tr>
<tr>
<td>RN_TestGenerateFail</td>
<td>DRBG Generate test failed.</td>
</tr>
<tr>
<td>RN_TestReseedFail</td>
<td>DRBG Reseed test failed.</td>
</tr>
<tr>
<td>RN_TestUninstanFail</td>
<td>DRBG Uninstantiate test failed.</td>
</tr>
</tbody>
</table>

102 IBM 4765 Custom Software Interface Reference
Refer to `xc_err.h` for a comprehensive list of return codes.
3.8.5  xcDRBGuninstantiate – uninstantiate DRBG
This function is available in version 4.3 and up.

The xcDRBGuninstantiate 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

```c
int xcDRBGuninstantiate(int fd, xcDRBGuninstantiate_t *pDRBGuninst);
```

Input
On entry to this routine:

- `fd` is the file descriptor of the /dev/hwrng device.
- `pDRBGuninst` is a pointer to a structure whose fields are initialized as follows:
  - `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 exit, 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:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RN_Success</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>RN_Invalid</td>
<td>One of the parameters in the structure pointed to by the <code>pDRBGuninst</code> parameter is invalid.</td>
</tr>
<tr>
<td>RN_NotWorking</td>
<td>DRBG not working; DRBG was working when the request was received, but the operation failed</td>
</tr>
</tbody>
</table>

Refer to xc_err.h for a comprehensive list of return codes.
3.8.6 `xcRandomNumberGenerate` - generate random number

`xcRandomNumberGenerate` generates a specific number of random bytes based on a hardware noise source. For version 4.3 and up, the random bits meet the standards described in Deterministic Random Bit Generator (DRBG) NIST Special Publication 800-90A. This function will call `xcDRBGinstantiate` (if required) and `xcDRBGgenerate` to generate the random numbers.

Note: This API is deprecated and will degrade performance of verbs that use random numbers. Use of this API should be replaced with use of the `xcDRBG` APIs.

**Function prototype**

```c
unsigned int xcRandomNumberGenerate(int         fd,
                                    xCRNG_RB_t *pRNG_rb);
```

**Input**

On entry to this routine:

- `fd` is the file descriptor returned when `/dev/hwrng` was opened.
- `pRNG_rb` contains the address of a RNG request block whose fields are initialized as follows:
  - `optionsRng` controls the operation of the function and must include exactly one of the following parity bit constants:

<table>
<thead>
<tr>
<th>RANDOM_RANDOM</th>
<th>Generate the requested number of random bytes. Disregard parity.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANDOM_ODD_PARITY</td>
<td>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.</td>
</tr>
<tr>
<td>RANDOM_EVEN_PARITY</td>
<td>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.</td>
</tr>
</tbody>
</table>

- In addition, `optionsRng` may include exactly one of the following source and filter constants:

<table>
<thead>
<tr>
<th>RANDOM_SW</th>
<th>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 <code>RANDOM_HW</code> is no longer supported since retrieving random bits directly from the hardware is a violation of the NIST SP 800-90 standard. Using <code>RANDOM_HW</code> will result in a return code of <code>RN_Invalid</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANDOM_NOT_WEAK</td>
<td>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.</td>
</tr>
</tbody>
</table>

- `lenRng` specifies the number of random bytes to generate.
• \textit{pBufferRng} points to a writeable buffer. The buffer must be at least \textit{lenRng} bytes long.

\textbf{Output}

On successful exit from this routine, \((pRNG\_rb->pBufferRng)\) contains \textit{pRNG\_rb->lenRng} random bytes, the low order bit of each set as requested by the “Parity Bit” setting of \textit{pRNG\_rb->optionsRng}.

\textbf{Notes}

The buffer referenced by \textit{pRNG\_rb->pBufferRng} must not reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

\textbf{Return codes}

Common return codes generated by this routine are:

\begin{center}
\begin{tabular}{|l|l|}
\hline
\textbf{RN\_Success} (that is, 0) & The operation was successful. \\
\hline
\textbf{RN\_Invalid} & One of the parameters in the structure pointed to by the \textit{pDRBGgen} parameter is invalid. \\
\hline
\textbf{RN\_NotWorking} & DRBG not working: DRBG was working when the request was received, but the operation failed. \\
\hline
\textbf{RN\_None} & DRBG was offline when the request was received. \\
\hline
\end{tabular}
\end{center}

Refer to \textit{xc\_err.h} for a comprehensive list of return codes.
3.8.7 xcTestRandomNumber - test random number generator

For version 4.3 and up, xcTestRandomNumber calls xcDRBGtest, which verifies that the xcDRBGInstantiate, xcDRBGgenerate, xcDRBGreseed, and xcDRBGuninstantiate functions work correctly.

Note: This API is deprecated and will degrade performance of verbs that use random numbers. Use of this API should be replaced with use of the xcDRBG APIs.

Function prototype

```c
unsigned int xcTestRandomNumber(int fd, xcrNG_test_RB_t *pRNG_test_rb);
```

Input

On entry to this routine:

- `fd` is the file descriptor returned when `/dev/hwrng` was opened.
- `pRNG_test_rb` contains the address of a RNG test request block whose fields are initialized as follows:
  - `options` must be set to `RNG_TEST_HRNG` or to `RNG_TEST_PRNG`.
    - This field is a carryover from a previous version of the coprocessor and has no effect on the operation of the function – the FIPS standards have changed over time and now only require a known-answer test on the pseudo-random stream.
    - Note: even if you specify `RNG_TEST_PRNG`, you must provide the file descriptor for the hardware RNG device.

Output

None.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RN_Success (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>RN_Invalid</td>
<td>One of the parameters in the structure pointed to by the pDRBGinst parameter is invalid.</td>
</tr>
<tr>
<td>RN_NotWorking</td>
<td>DRBG not working: DRBG was working when the request was received, but the operation failed</td>
</tr>
<tr>
<td>RN_None</td>
<td>DRBG was offline when the request was received.</td>
</tr>
<tr>
<td>RN_TestInstantiateFail</td>
<td>DRBG INSTANtiation test failed.</td>
</tr>
<tr>
<td>RN_TestGenerateFail</td>
<td>DRBG Generate test failed.</td>
</tr>
<tr>
<td>RN_TestReseedFail</td>
<td>DRBG Reseed test failed.</td>
</tr>
<tr>
<td>RN_TestUninstantiateFail</td>
<td>DRBG Uninstantiatie test failed.</td>
</tr>
</tbody>
</table>

Refer to `xc_err.h` for a comprehensive list of return codes.
3.9 Configuration functions

The functions described in this section allow a coprocessor application to interact with the 4765 driver to
configure certain processor features or obtain information about the coprocessor.

A coprocessor application must call xcAttachWithCDUOption and must open /dev/crypto with the
O_RDWR flag before calling any of the functions in this section.

3.9.1 Privileged operations

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

3.9.2 Date/time

Support for querying and modifying the time-of-day is provided through the Linux real-time clock (RTC)
driver and device (/dev/rtc). This support is provided within the generic Linux kernel that is provided on
the PCIe Cryptographic Coprocessor.

3.9.3 xcGetConfig - get coprocessor configuration

xcGetConfig obtains information about the coprocessor.

Function prototype

\[
\text{unsigned int } \text{xcGetConfig}(\text{int } \text{fd}, \text{xcAdapterInfo_t *pConfigData, unsigned long *info_length});
\]

Input

On entry to this routine:

\text{fd} is the file descriptor returned when /dev/crypto was opened.
\text{pConfigData} points to a writeable buffer.
\text{info_length} points to a writeable buffer containing the length in bytes of the buffer referenced by
\text{pConfigData}.

Output

On successful exit from this routine:

\text{*pConfigData} 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 \text{xcAdapterInfo_t} structure whose fields are set as
indicated below. If the buffer was too small, the structure is truncated.

\text{*info_length} contains the length of the full \text{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 \text{xcAdapterInfo_t} structure are set as follows:

\begin{itemize}
  \item \text{sid} identifies the structure and contains its length in bytes. \text{sid.ID} is \text{STRUCT_xcAdapterInfo}.
  \item \text{AMCC_EEPROM} is undefined.
\end{itemize}
• **HdwRigolettoID** is the firmware version for the FPGA in the host-to-coprocessor bridge.
• **HdwOtelloECID** is the version for the coprocessor's crypto chip.
• **EthernetMAC** is the MAC address of the coprocessor's onboard Ethernet adapter.
• **VPD** contains the coprocessor Vital Product Data. Its fields are set as follows:
  
  - **ds_tag** is 0x82.
  - **ds_length** is 44. *This field is in little-endian byte order as required by PCI specification.*
  - **ds** contains the unquoted ASCII characters “IBM 4765-001 PCI-e Cryptographic Coprocessor”.
  - **vpdr_tag** is 0x90.
  - **vpdr_length** is 205. *This field is in little-endian byte order as required by PCI specification.*
  - **ec_tag** contains the unquoted ASCII characters “EC”.
  - **ec_length** is 7.
  - **ec** contains ASCII characters that specify the engineering change level for the cryptographic coprocessor (e.g., 0G36842).
  - **pn_tag** contains the unquoted ASCII characters “PN”.
  - **pn_length** is 7.
  - **pn** contains ASCII characters that specify the part number for the cryptographic coprocessor (e.g., 45D2365).
  - **fn_tag** contains the unquoted ASCII characters “FN”.
  - **fn_length** is 7.
  - **fn** contains ASCII characters that specify the field replaceable unit number for the cryptographic coprocessor (e.g., 44P1035).
  - **mf_tag** contains the unquoted ASCII characters “MF,”
  - **mf_length** is 2.
  - **mf** contains ASCII characters that specify the location where the cryptographic coprocessor was manufactured (e.g., 91).
  - **sn_tag** contains the unquoted ASCII characters “SN”.
  - **sn_length** is 8.
  - **sn** contains ASCII characters that specify the serial number for the cryptographic coprocessor (e.g., 99000061).
  - **rv_tag** contains the unquoted ASCII characters “RV”.
  - **rv_length** is 155.
  - **checksum** contains a checksum covering that portion of the *xcAdapterInfo_t* structure that appears prior to the checksum.
  - **reserved** is undefined.
• **end_tag** is 0x78.

• **rsvd** is undefined.

• **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.*

• **HardwareOptions** provides information about the coprocessor hardware. **HardwareOptions.DES_level** is the level of the SKCH engine and **HardwareOptions.RSA_level** is the level of the modular math engine.

• **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.

**Notes**

Neither the buffer referenced by **pConfigData** nor the buffer referenced by **info_length** may reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th><strong>SCCGood</strong> (that is, 0)</th>
<th>The operation was successful.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SCCBadFun</strong></td>
<td>Invalid function.</td>
</tr>
<tr>
<td><strong>SCCBadFlags</strong></td>
<td>Invalid options.</td>
</tr>
<tr>
<td>SCCBadParm</td>
<td>Invalid parameter.</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>SCCNotFound</td>
<td>Device driver access error.</td>
</tr>
<tr>
<td>SCCNotAuthorized</td>
<td>Invalid requestor authority.</td>
</tr>
<tr>
<td>SCCGenErr</td>
<td>General error.</td>
</tr>
</tbody>
</table>

Refer to *xc_err.h* for a comprehensive list of return codes.
3.9.4 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 \texttt{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. \texttt{xcClearILatch} resets the coprocessor intrusion latch.

**Function prototype**

\[
\text{unsigned int } \text{xcClearILatch}(\text{int } \text{fd});
\]

**Input**

On entry to this routine:

\texttt{fd} is the file descriptor returned when \texttt{/dev/crypto} was opened.

**Output**

None.

**Notes**

An application must run as root to call \texttt{xcClearILatch}.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCCGood (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>SCCBadFu</td>
<td>Invalid function.</td>
</tr>
<tr>
<td>SCCBadFlags</td>
<td>Invalid options.</td>
</tr>
<tr>
<td>SCCBadParm</td>
<td>Invalid parameter.</td>
</tr>
<tr>
<td>SCCNotFound</td>
<td>Device driver access error.</td>
</tr>
<tr>
<td>SCCNotAuthorized</td>
<td>Invalid requestor authority.</td>
</tr>
<tr>
<td>SCCGenErr</td>
<td>General error.</td>
</tr>
</tbody>
</table>

Refer to \texttt{xc_err.h} for a comprehensive list of return codes.
3.9.5  

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 disable 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 17.

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. `xcClearLowBatt` resets the coprocessor low battery warning latch.

**Function prototype**

```
unsigned int xcClearLowBatt(int fd);
```

**Input**

On entry to this routine:

`fd` is the file descriptor returned when `/dev/crypto` was opened.

**Output**

None.

**Notes**

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

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCCGood</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>SCCBadFun</td>
<td>Invalid function.</td>
</tr>
<tr>
<td>SCCBadFlags</td>
<td>Invalid options.</td>
</tr>
<tr>
<td>SCCBadParm</td>
<td>Invalid parameter.</td>
</tr>
<tr>
<td>SCCNotFound</td>
<td>Device driver access error.</td>
</tr>
<tr>
<td>SCCNotAuthorized</td>
<td>Invalid requestor authority.</td>
</tr>
<tr>
<td>SCCGenErr</td>
<td>General error.</td>
</tr>
</tbody>
</table>

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

---

17 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.
3.10 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.

3.10.1 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 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

18 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.

3.10.2 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") 19.

Updates to segment 1

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

- Generates a new random RSA keypair (a "Transition Keypair").
- 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.
- 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

19 The value of the IBM Root Public Key appears in “IBM root public keys” on page 150.
includes the program with the suspicion it warrants 20.

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 21.

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 22.

An application can ask the OA daemon 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:

20 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.
21 Refer to “Using Signer and Packager” in the IBM 4765 PCIe Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide for a discussion of which operations clear segment 3 BBRAM.
22 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 half of any configuration keypairs the application has caused to be created
2. The private half 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 halves 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.
Since the existing Configuration Keypair no longer has a private key, the application must ask the OA daemon 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.
3.10.3 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.\textsuperscript{23}
- **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_NEWCERT** 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.\textsuperscript{24}
- **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 `SHA1_ISO_RSA`, an RSA private key is used to generate the signature. The body of the certificate is hashed using the SHA-1 algorithm. The hash is extended according to the ISO-9796-1 standard to the full length of the modulus of the key. The extended hash is then decrypted using the RSA private key to give the signature.

  If `tSig` is `SHA1_ISO_COMPLEMENT_RSA`, the cryptographic signature is computed in the same manner as in the `SHA1_ISO_RSA` case. If the signature is congruent to 6 modulo 16, it is then complemented with respect to the modulus of the key; otherwise, the signature is used as-is.

  If `tSig` is `SHA256_X931_RSA`, an RSA private key is used to generate the signature. The body of the certificate is hashed using the SHA-256 algorithm. The hash is encapsulated according to the ANSI X9.31 standard. The encapsulated hash is then decrypted using the RSA private key to give the signature.

\textsuperscript{23} 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.

\textsuperscript{24} 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 tSig is **DSS_COMPLIANT**, a DSA private key is used to generate the signature. The body of the certificate is processed as dictated by the Digital Signature Standard.

A signature generated using an RSA 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.

A signature generated using a DSA private key is stored in a DSA signature token. The block of data whose offset and length are specified in vSig begins with a structure of type `sccDSASignatureToken_t`. This structure defines the elements of the signature (as described in “DSA signature tokens” on page 86), which appear following the `sccDSASignatureToken_t` structure.

- **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 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 Error: Reference source not found” 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 69), which appear following the structure.
  - If **tPublic** is **OA_DSS**, the public key is a DSA public key. The block of data whose offset and length are specified in vPublic begins with a structure of type `sccDSAKeyToken_t`. This structure defines the elements of the public key (as described in “DSA key tokens” on page 85), 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 133 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 131 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 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 miniboott 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 \texttt{xcOA\_CKO\_Head\_t} and \texttt{xcOA\_CKO\_Body\_t} headers for a Transition Certificate are set as follows:

- \textit{cko\_type} is \texttt{OA\_CKO\_MB}.
- \textit{cko\_name} names a coprocessor-generated keypair. See “Keypair names” on page 130 for details.
- \textit{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.
- \textit{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 131 for details.
- \textit{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 131 for details. The fields of \textit{vDescA} are used in the same manner as the fields of \texttt{xcOA\_CKO\_Head\_t.vData}.
- \textit{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 \textit{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 \textit{vDescB} are used in the same manner as the fields of \texttt{xcOA\_CKO\_Head\_t.vData}.

Operating system key certificates

The type-dependent fields in the \texttt{xcOA\_CKO\_Head\_t} and \texttt{xcOA\_CKO\_Body\_t} headers for an Operating System Key Certificate are set as follows:

- \textit{cko\_type} is \texttt{OA\_CKO\_SEG2\_SEG3}.
- \textit{cko\_name} names a coprocessor-generated keypair. See “Keypair names” on page 130 for details.
- \textit{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.
- \textit{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 131 for details.
- \textit{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 \textit{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 \textit{vDescA} are used in the same manner as the fields of \texttt{xcOA\_CKO\_Head\_t.vData}.
- \textit{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 \textit{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 \textit{vDescB} are used in the same manner as the fields of \texttt{xcOA\_CKO\_Head\_t.vData}.

Application key certificates

The type-dependent fields in the \texttt{xcOA\_CKO\_Head\_t} and \texttt{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 131 for details.

- **vDescA** is reserved.

- **vDescB** specifies the offset and length of a block of data supplied by the application to be associated with the certificate when the keypair was created. See “xcOAGenerate - generate application keypair and OA certificate” on page 137 and “xcOAGetCert - retrieve an OA certificate” on page 136 for details. 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 \textit{creation\_boot}.

- \textit{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.

\textbf{Device names and device descriptors}

The interface to the OA daemon defines the \textit{xcOADeviceName\_t} type to hold the name of a particular coprocessor. The fields in a \textit{xcOADeviceName\_t} structure are set as follows:

- \textit{struct\_id.name} is \textit{XCOADEVICENAME\_T}.
- \textit{struct\_id.version} is the value to which \textit{XCOADEVICENAME\_VER} is defined in the header file that defines the version of \textit{xcOADeviceName\_t} that maps the header. See “footnote Error: Reference source not found” on page 126 for the description of \textit{struct\_id.version} for \textit{xcOA\_CKO\_Head\_t}.
- \textit{padbytes} is two bytes of zeros.
- \textit{adapterID} is a serial number that uniquely identifies the coprocessor. It matches the value of the AdapterID field returned by \textit{xcGetConfig}. See “\textit{xcGetConfig} - get coprocessor configuration” on page 108 for details.

\textit{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.

\textbf{Layer names and layer descriptors}

The interface to the OA daemon defines the \textit{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:

- \textit{struct\_id.name} is \textit{XCOALAYERNAME\_T}.
- \textit{struct\_id.version} is the value to which \textit{XCOALAYERNAME\_VER} is defined in the header file that defines the version of \textit{xcOALayerName\_t} that maps the timestamp. See “footnote Error: Reference source not found” on page 126 for the description of \textit{struct\_id.version} for \textit{xcOA\_CKO\_Head\_t}.
- \textit{padbytes} is two bytes of zeros.
- \textit{epoch\_start} marks the beginning of the epoch in which the software that the structure names was loaded. In particular, \textit{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.
- \textit{config\_start} marks the start of the configuration that includes the software that the structure names. In particular, \textit{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.
- \textit{config\_count} specifies how many configurations there have been during the epoch whose
beginning \textit{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 $\text{xcOALayerDesc\_t}$ 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:

- \textit{struct\_id.name} is $\text{XCOALAYERDESC\_T}$.
- \textit{struct\_id.version} is the value to which $\text{XCOALAYERDESC\_VER}$ is defined in the header file that defines the version of $\text{xcOALayerDesc\_t}$ that maps the layer description. See “footnote Error: Reference source not found” on page 126 for the description of \textit{struct\_id.version} for $\text{xcOA\_CKO\_Head\_t}$.
- \textit{padbyte} is one byte of zeros.
- \textit{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.
- \textit{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.
- \textit{image\_name} is the name associated with the software. See “Overview of the authentication scheme” on page 115 for details.
- \textit{image\_revision} is the revision number associated with the software. See “Overview of the authentication scheme” on page 115 for details.
- \textit{image\_hash} is the SHA-256 hash of the software. See “Overview of the authentication scheme” on page 115 for details.
- \textit{layer\_name} uniquely identifies the software.

\textbf{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:
• \( \text{epoch\_start} = 0x60c \)
• \( \text{config\_start} = 0x612 \)
• \( \text{config\_count} = 2 \)

and a layer descriptor for segment 3 whose fields have the following values:

• \( \text{epoch\_start} = 0x60d \)
• \( \text{config\_start} = 0x620 \)
• \( \text{config\_count} = 3 \)

**Timestamps**

The interface to the OA daemon defines the \( \text{xcOATime\_t} \) type to hold a timestamp. The fields of a timestamp are set as follows:

- \( \text{struct\_id.name} \) is \( \text{xcOATIME\_T} \).
- \( \text{struct\_id.version} \) is the value to which \( \text{XCOATIME\_VER} \) is defined in the header file that defines the version of \( \text{xcOATime\_t} \) that maps the timestamp. See “footnote Error: Reference source not found” on page 126 for the description of \( \text{struct\_id.version} \) for \( \text{xcOA\_CKO\_Head\_t} \).
- \( \text{year} \) is a BCD representation of the year (for example, 0x2000 represents the year 2000).
- \( \text{month} \) is a BCD representation of the month (for example, 0x12 represents December).
- \( \text{day} \) is a BCD representation of the day of the month (for example, 0x10 represents the 10th).
- \( \text{hour} \) is a BCD representation of the hour using a 24-hour clock (for example, 0x17 represents 5 p.m.).
- \( \text{minute} \) 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 \( \text{xcOA\_CKO\_Descr\_t} \) type to hold the description of an IBM Class Root Keypair. The fields in the \( \text{xcOA\_CKO\_Descr\_t} \) structure are set as follows:

- \( \text{struct\_id.name} \) is \( \text{XCOA\_CKO\_DESCR\_T} \).
- \( \text{struct\_id.version} \) is the value to which \( \text{XCOA\_CKO\_DESCR\_VER} \) is defined in the header file that defines the version of \( \text{xcOA\_CKO\_Descr\_t} \) that maps the description. See “footnote Error: Reference source not found” on page 126 for the description of \( \text{struct\_id.version} \) for \( \text{xcOA\_CKO\_Head\_t} \).
- \( \text{cert\_qualifier} \) is an integer that identifies the keypair. See the \( \text{OA\_CLASS\_ROOT\_*} \) constants in \( \text{xc\_oa\_mb\_h} \).
- \( \text{descr} \) is a text description of the keypair.
3.10.4   xcOAGetDir - count and list OA certificates

xcOAGetDir 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.

Function prototype

```c
unsigned int xcGetOADir(int            fd,  
unsigned long  *pCount,     
void           *pBuffer,     
unsigned long  *pLen);
```

Input

On entry to this routine:

* `fd` is not used.

* `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`.

Output

On successful exit from this routine:

* `*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 Error: Reference source not found” 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.
- **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_DSS` if the keypair is a DSA 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...

---

25 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.

---

134 IBM 4765 Custom Software Interface Reference
• *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 in the certificate list.

**Notes**
None of the buffers referenced by *pCount*, *pBuffer*, and *pLen* may reside in a mapped kernel buffer. See "Mapped kernel buffers and DMA-eligible buffers" on page 23 for details.

**Return codes**
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OAGood (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>OABadParm</td>
<td>An argument is not valid.</td>
</tr>
<tr>
<td>OANoSpace</td>
<td>The operation failed due to lack of space (for example, <em>pLen</em> is too small).</td>
</tr>
</tbody>
</table>

Refer to *xc_err.h* for a comprehensive list of return codes.
3.10.5  xcOAGetCert - retrieve an OA certificate

xcOAGetCert either returns the length of an OA certificate in the certificate list or retrieves the certificate itself.

Function prototype

\[
\text{unsigned int } \text{xcOAGetCert}(\text{int } \text{fd},
\text{xcOA_CKO_Name_t *} \text{pCKO_name}, \text{void *} \text{pBuffer}, \text{unsigned long *} \text{pLen});
\]

Input

On entry to this routine:

\text{fd} is not used.

\text{*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.

\text{pBuffer} either must be NULL or must point to a writeable buffer.

\text{pLen} points to a writeable buffer. If \text{pBuffer} is not NULL, \text{*pLen} is the length in bytes of the buffer referenced by \text{pBuffer}.

Output

On successful exit from this routine:

\text{*pLen} is the length in bytes of the OA certificate of interest.

If \text{pBuffer} is not NULL, the buffer it references contains a copy of the desired OA certificate. See “OA certificates” on page 124 for details.

Notes

Neither the buffer referenced by \text{pBuffer} nor the buffer referenced by \text{pLen} may reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OAGood (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>OABadParm</td>
<td>An argument is not valid.</td>
</tr>
<tr>
<td>OANoSpace</td>
<td>The operation failed due to lack of space (for example, \text{*pLen} is too small).</td>
</tr>
<tr>
<td>OANotFound</td>
<td>\text{*pCKO_name} does not identify an OA certificate in the certificate list.</td>
</tr>
</tbody>
</table>

Refer to \text{xc_err.h} for a comprehensive list of return codes.
3.10.6  xCOAGenerate - generate application keypair and OA certificate

xCOAGenerate 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.

Function prototype

```c
unsigned int xCOAGenerate(int fd,
xCOAGen_RB_t *pOAGenRB,
unsigned long lOAGenRB,
void *pPKGenRB,
unsigned long lPKGenRB);
```

Input

On entry to this routine:

- `fd` is not used.
- `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 Error: Reference source not found” 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 either `OA_RSA` (to generate an RSA keypair) or `OA_DSS` (to generate a DSA 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.
    - `vSeg3Field.length` is the length in bytes 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).

- `pCKO_name` points to a writeable buffer in which an item of type `xcOA_CKO_Name_t` can be stored.

---

26 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.
IOAGenRB is the length in bytes of the request block referenced by pOAGenRB. IOAGenRB includes the length of the block of data whose offset and length are specified in pOAGenRB->vSeg3Field.

pPKGenRB points to a public key algorithm key generate request block:

- If pOAGenRB->algorithm is OA_RSA, *pPKGenRB must be an RSA key generate request block (an item of type xcRSAKeyGen_RB_t) whose key_type, mod_size, and public_exp fields are initialized as required by xcRSAKeyGenerate (see “xcRSAKeyGenerate - generate an RSA keypair” on page 73 for details). public_exp must not be RSA_EXPONENT_FIXED. The remaining fields in *pPKGenRB are ignored.

- If pOAGenRB->algorithm is OA_DSS, *pPKGenRB must be a DSA key generate request block (an item of type xcdSAKeyGen_RB_t) whose prime_p_size field is initialized as required by xcdSAKeyGenerate (see “xcDSAKeyGenerate - generate DSA keypair” on page 87 for details). The remaining fields in *pPKGenRB are ignored.

IPKGenRB is the length in bytes of the request block referenced by pPKGenRB.

Output
On successful exit from this routine:

*(pOAGenRB->pCKO_name) identifies the newly generated Application Keypair. See “Keypair names” on page 130 for details.

Notes

Signature on new OA certificate

The cryptographic signature for the OA certificate generated by xcOAGenerate 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).

Use separate keys to encrypt and to sign

It is recommended that an RSA private key be used solely to encrypt or solely to create cryptographic signatures, and not for both purposes. If an application needs to perform both operations, the application should create separate Application Keypairs and set a flag in the block of data whose offset and location are specified in pOAGenRB->vSeg3Field be used to indicate in which operation a particular Application Keypair is to be used. The application should check the flag prior to calling xcOAPrivOp to ensure the private key is being used in the intended manner.

Mapped kernel buffers

The buffer referenced by pOAGenRB->pCSO_name must not reside in a mapped kernel buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OAGood (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>OABadParm</td>
<td>An argument is not valid.</td>
</tr>
<tr>
<td>OANotAllowed</td>
<td>$pOAGenRB-&gt;cko_type$ is neither $\text{OA_CKO_SEG3_CONFIG}$ nor $\text{OA_CKO_SEG3_EPOCH}$.</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OANoSpace</td>
<td>The operation failed due to lack of space.</td>
</tr>
</tbody>
</table>

Refer to $xc\_err.h$ for a comprehensive list of return codes.
3.10.7     **xcOADelete - delete application keypair and OA certificate**

*xcOADelete* deletes an Application Keypair and the OA certificate that contains the keypair's public key.

**Function prototype**

```c
unsigned int xOADelete(int              fd,
                        xCOA_CKO_Name_t *pCKO_name);
```

**Input**

On entry to this routine:

- `fd` is not used.
- `*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.

**Output**

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.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OAGood</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>OABadParm</td>
<td>An argument is not valid.</td>
</tr>
<tr>
<td>OANotAllowed</td>
<td><code>*pCKO_name</code> does not identify an Application Keypair.</td>
</tr>
<tr>
<td>OANotFound</td>
<td><code>*pCKO_name</code> does not identify an OA certificate in the certificate list.</td>
</tr>
</tbody>
</table>

Refer to *xc_err.h* for a comprehensive list of return codes.
3.10.8 xCOAPrivOp - perform cryptographic operation with an application key

xCOAPrivOp 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.

Function prototype

unsigned int xCOAPrivOP(int fd,
                        xCOA_CKO_Name_t *pCKOName,
                        void *pPKPrivRB,
                        unsigned long lPKPrivPB);

Input

On entry to this routine:

fd is not used.

*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:

- If the keypair identified by *pCKOName is an RSA keypair, *pPKPrivRB must be an RSA operation request block (an item of type xcRSA_RB_t) whose options, data_in, data_out, and data_size fields are initialized as required by xcRSA (see “xcRSA - encipher/decipher data or wrap/unwrap X9.31 encapsulated hash” on page 78 for details). options must include RSA_DONT_BLIND. The remaining fields in *pPKPrivRB are ignored.

- If the keypair identified by *pCKOName is a DSA keypair and *pPKPrivRB is a DSA operation request block (an item of type xcDSA_RB_t) whose options, sig_token, sig_token_size, data, and data_size fields are initialized as required by xcDSA (see “xcDSA - sign data or verify signature for data” on page 90 for details). The remaining fields in *pPKPrivRB are ignored.

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 xcRSA or xcDSA, as appropriate.

lPKPrivRB is the length in bytes of the request block referenced by pPKPrivRB.

Output

On successful exit from this routine:

- If the keypair identified by *pCKOName is an RSA keypair, (((xcRSA_RB_t *)pPKPrivRB)->data_out) contains
  - "(((xcRSA_RB_t *)pPKPrivRB)->data_in) transformed using the public half of the keypair identified by *pCKOName if (((xcRSA_RB_t *)pPKPrivRB)->options specifies RSA_PUBLIC and
  - "(((xcRSA_RB_t *)pPKPrivRB)->data_in) transformed using the private half of the keypair identified by *pCKOName if (((xcRSA_RB_t *)pPKPrivRB)->options specifies RSA_PRIVATE.

- If the keypair identified by *pCKOName is a DSA keypair and (((xcDSA_RB_t *)pPKPrivRB)-
>options specifies DSA_SIGNATURE_SIGN, (((xcDSA_RB_t *)pPKPrivRB)->sig_token) contains
the digital signature produced by signing (((xcDSA_RB_t *)pPKPrivRB)->data) with the private
half of the keypair identified by "pCKOName".

- If the keypair identified by "pCKOName" is a DSA keypair and (((xcDSA_RB_t *)pPKPrivRB)-
>options specifies DSA_SIGNATURE_VERIFY, a return code of zero implies that the signature
in (((xcDSA_RB_t *)pPKPrivRB)->sig_token) was produced by signing (((xcDSA_RB_t
*)pPKPrivRB)->data) with the private half of the keypair identified by "pCKOName".

**Notes**
The output buffer referenced by the appropriate field of pPKPrivRB must not reside in a mapped kernel
buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

**Return codes**
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OAGood (that is, 0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>OABadParm</td>
<td>An argument is not valid.</td>
</tr>
<tr>
<td>OANotAllowed</td>
<td>&quot;pCKOName&quot; does not identify an Application Keypair.</td>
</tr>
<tr>
<td>OANotFound</td>
<td>&quot;pCKOName&quot; does not identify any keypair.</td>
</tr>
<tr>
<td>OANoSpace</td>
<td>The operation failed due to lack of space.</td>
</tr>
<tr>
<td>PKADSA SigIncorrect</td>
<td>The keypair identified by &quot;pCKOName&quot; is a DSA keypair and a DSA verify operation was requested but the signature failed to verify.</td>
</tr>
</tbody>
</table>

Refer to xc_err.h for a comprehensive list of return codes.
3.10.9  xcOAStatus - get coprocessor status

xcOAStatus 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.

Function prototype

    unsigned int xcOAStatus(int            fd,
                            void          *pBuffer,
                            unsigned long *pLen);

Input

On entry to this routine:

    fd is unused.
    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.

Output

On successful exit from this routine:

    *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 Error: Reference source not found” 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 108 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 108 for 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 

• 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 

• 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 
  131 for details.

Notes
Neither the buffer referenced by pBuffer nor the buffer referenced by pLen may reside in a mapped kernel 
buffer. See “Mapped kernel buffers and DMA-eligible buffers” on page 23 for details.

Return codes
Common return codes generated by this routine are:
<table>
<thead>
<tr>
<th>OA Good (that is, 0)</th>
<th>The operation was successful.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OABadParm</td>
<td>An argument is not valid.</td>
</tr>
<tr>
<td>OANoSpace</td>
<td>The operation failed due to lack of space (for example, *pLen is too small to hold the entire status structure).</td>
</tr>
</tbody>
</table>

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:

\textit{axWXYYzzzz}

where:

- \textit{W}: Eight indicates a negative number; an error has occurred
- \textit{X}: Used by the error-generating module; usually zero
- \textit{YY}: Code number of the error-generating module
- \textit{Zzzz}: Actual error code determined by the entity detecting the error

Common code combinations for WXYY:

Error Codes

\begin{itemize}
  \item 8001 SVC Handler
  \item 8002 Memory Manager
  \item 8003 Resource Manager
  \item 8004 Session Manager
  \item 8006 Loader
  \item 8007 File API Stubs
  \item 8040 Host Device Driver
  \item 8107 File Router
  \item 8140 Host Operating System
  \item 8207 File System
  \item 8240 POST Error
  \item 8307 Device Router
  \item 8340 MiniBoot 0
  \item 8407 Device Driver
  \item 8440 MiniBoot 1
  \item 8507 Server File Router
  \item 8607 Client File Router
\end{itemize}

xC Error Codes

\begin{itemize}
  \item 8041 xC Support Manager
  \item 8042 Comm Driver
\end{itemize}
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8044</td>
<td>DES Driver</td>
</tr>
<tr>
<td>8045</td>
<td>PKA Driver</td>
</tr>
<tr>
<td>8046</td>
<td>RNG Driver</td>
</tr>
<tr>
<td>8048</td>
<td>OA Driver</td>
</tr>
</tbody>
</table>

**Reserved for IBM Use**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>806x</td>
<td>CCA modules</td>
</tr>
</tbody>
</table>

**Programmer Defined**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8X8x</td>
<td>Used by applications</td>
</tr>
</tbody>
</table>

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
011fe0f0e 010ef1fe
011ffee0 010efee1
01e001e0 01f101f1
01e01ffe 01f10eef
01e0e001 01f1f101
01e0f01f 01f1f0e0
01f001fe 01f0e0fe
01f1fe0f 01f0f0ef
01fefe0f 01f1f1ef
1f01011f 0e01010e
1f011f01 0e010e01
1f01e0fe 0e01f1fe
1f01fee0 0e01feef
1f1f0010 0e0e0101
1f1f010f 0e0e0f0e
1f1f0e0e 0e0e0f01
1f1ffe0f 0e0e0eef
1f00e1fe 0ef01fe
1fe01fe0 0ef0eef1
1f0e01f0 0ef1f0e0
1fe0fe01 0ef1fee0
1fe0e01f 0eef01f1
1ffef00f 0eefef0e
1ffe1ffe 0efe0eef
1ffefe0f 0eefef0e
1f00101e 0f101f1
DES weak, semi-weak, and possibly weak keys
6 IBM root public keys

As of the date of this document, the key IBM uses to sign the certificates for the class keys used with the IBM 4765 PCIe Cryptographic Coprocessor is a 4096-bit RSA key whose public exponent is 65537 (decimal) and whose modulus in hex is as follows:

C686E350 E09D6B08
64914CED CSA50827
9D9C9ADA 6A84F01A
239D9ADB 7B0CCD07
B350EA55 A10142BD
DB8DCCFD CBF54BE3
1E784362 B4734E76
ED9583E5 98BF868A
B168464C C118099D
EBA19A51 963FD3F1
8B39D9D1 E371D5E9
52361D20 D7F6EA9F
1C31A527 B7D94D0A
57DB1DF0 41B7C25B
B77ECDB4 16D9BEA0
D012C320 DD1E94D1
0E7C36C3 5B7D333A
FC168A86 7FDBEA30
C82BF1FF 75C65391
3EAF7D63 C022E074
F8C34C5B 734D09FA
0C24583C 8440F167
6C5CB5E2 FA8D676B
37151A5D DC47AA1E
9B5FF846 B68EC29C
9226F1B5 B19E3FD5
C134CF36 34C77FD8
AE01DA56 1ECE58B9
B415A143 4E176477
16C42EFB AA3A2E19
93639C01 CA841A91
2B580065 DC73C676
The most significant byte of the modulus is 0xC6 and the least significant byte is 0xB9.

The key IBM uses to sign the certificates for the class keys used with the IBM 4758 Cryptographic Coprocessor and the IBM 4764 PCI-X Cryptographic Coprocessor is a 1024-bit RSA key whose public exponent is 65537 (decimal) and whose modulus in hex is as follows:
The most significant byte of the modulus is 0x80 and the least significant byte is 0x17.
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

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

- AIX
- IBM
- eServer
- z/OS
- zSeries
Linux is a trademark of Linus Torvalds 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 and acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
<td>PCI</td>
<td>peripheral component interconnect</td>
</tr>
<tr>
<td>API</td>
<td>Application program interface</td>
<td>PCle</td>
<td>Peripheral component interconnect express</td>
</tr>
<tr>
<td>ASCII</td>
<td>American National Standard Code for Information Exchange</td>
<td>PCi-X</td>
<td>peripheral component interconnect extended</td>
</tr>
<tr>
<td>BBRAM</td>
<td>battery-backed random access memory</td>
<td>PDF</td>
<td>portable document format</td>
</tr>
<tr>
<td>CCA</td>
<td>Common Cryptographic Architecture</td>
<td>PIN</td>
<td>personal identification number</td>
</tr>
<tr>
<td>CMK</td>
<td>current master key</td>
<td>PKA</td>
<td>public key algorithm</td>
</tr>
<tr>
<td>CPRB</td>
<td>Cooperative Processing Request Block</td>
<td>RAM</td>
<td>random access memory</td>
</tr>
<tr>
<td>CLU</td>
<td>Coprocessor Load Utility</td>
<td>RNG</td>
<td>random number generator</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard</td>
<td>RSA</td>
<td>Rivest-Shamir-Adleman (algorithm)</td>
</tr>
<tr>
<td>EPROM</td>
<td>Erasable Programmable Read-Only Memory</td>
<td>SET</td>
<td>secure electronic transaction</td>
</tr>
<tr>
<td>FIPS</td>
<td>Federal Information Processing Standard</td>
<td>SHA</td>
<td>Secure Hash Algorithm</td>
</tr>
<tr>
<td>KEK</td>
<td>key encrypting key</td>
<td>SLES</td>
<td>SUSE Linux Enterprise Server</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines Corporation</td>
<td>SRDI</td>
<td>Security Relevant Data Item</td>
</tr>
<tr>
<td>IPL</td>
<td>Initial program load</td>
<td>TOD</td>
<td>time-of-day (clock)</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
<td>TVV</td>
<td>token validation value</td>
</tr>
<tr>
<td>MAC</td>
<td>message authentication code</td>
<td>UDX</td>
<td>user-defined extension</td>
</tr>
<tr>
<td>MD5</td>
<td>Message digest 5 (hashing algorithm)</td>
<td>VPD</td>
<td>vital product data</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MKVP</td>
<td>master key verification pattern</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>xC,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>xCrypto</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NMK</td>
<td>new master key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OMK</td>
<td>old master key</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*IBM 4764 PCI-X Cryptographic Coprocessor*
9 Glossary

This glossary includes terms and definitions from the IBM Dictionary of Computing, New York: McGraw Hill, 1994. This glossary also includes terms and definitions taken from:

- The American National Standard Dictionary for Information Systems, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42 Street, New York, New York 10036. Definitions are identified by the symbol (A) following the definition.

- The Information Technology Vocabulary, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) following the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) following the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

A

access. In computer security, a specific type of interaction between a subject and an object that results in the flow of information from one to the other.

access control. Ensuring that the resources of a computer system can be accessed only by authorized users and in authorized ways.

access method. A technique for moving data between main storage and input/output devices.

adapter. Synonym for expansion card.

agent. (1) An application that runs within the IBM 4765. (2) Synonym for Cryptographic coprocessor application.

American National Standard Code for Information Interchange (ASCII). The standard code, using a coded character set consisting of seven-bit characters (eight bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. (A)

American National Standards Institute (ANSI). An organization consisting of producers, consumers, and general interest groups that establishes the procedures by which accredited organizations create and maintain voluntary industry standards for the United States. (A)

ANSI. American National Standards Institute.

API. Application program interface.

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. (1) A process used to verify the integrity of transmitted data, especially a message. (T) (2) In computer security, a process used to verify the user of an information system or protected resource.

authorization. (1) In computer security, the right granted to a user to communicate with or make use of a computer system. (T) (2) The process of granting a user either complete or restricted access to an object, resource, or function.
authorize. To permit or give authority to 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 4765 PCIe Cryptographic Coprocessor uses BBRAM to store persistent data for xC applications, as well as the coprocessor device key.

BBRAM. Battery-backed random access memory.

bus. In a processor, a physical facility along which data is transferred.

C

call. The action of bringing a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and jumping to an entry point. (1) (A)

card. (1) An electronic circuit board that is plugged into an expansion slot of a system unit. (2) A plug-in circuit assembly. (3) See also expansion card.

CBC. Cipher block chain.

CCA. Common Cryptographic Architecture.

ciphertext. (1) Data that has been altered by any cryptographic process. (2) See also plaintext.

cipher block chain (CBC). A mode of operation that cryptographically connects one block of ciphertext to the next plaintext block.

cleartext. (1) Data that has not been altered by any cryptographic process. (2) Synonym for plaintext. (3) See also ciphertext.

CLU. Coprocessor Load Utility.

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 Common Cryptographic Architecture functions; it is described in the IBM 4765 CCA Basic Services Reference and Guide.

coprocessor. (1) A supplementary processor that performs operations in conjunction with another processor. (2) A microprocessor on an expansion card 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 4765 PCIe Cryptographic Coprocessor.

Cryptographic Coprocessor (IBM 4765). An expansion card that provides a comprehensive set of cryptographic functions to a workstation.

cryptographic node. A node 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.

Data Encryption Standard Manager (DES_Mgr). A Linux extension that manages the IBM 4765 PCIe Cryptographic Coprocessor DES processing hardware.

decipher. (1) To convert enciphered data into clear data. (2) Contrast with encipher.

DES. Data Encryption Standard.

DES_Mgr. Data Encryption Standard Manager.

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.

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.

EPROM. Erasable programmable read-only memory.

erasable programmable read-only memory (EPROM). Programmable read-only memory that can be erased by a special process and reused.

expansion board. Synonym for expansion card.

expansion card. A circuit board that a user can plug into an expansion slot to add memory or special features to a computer.

expansion slot. One of several receptacles in a computer into which a user can install an expansion card.

feature. A part of an IBM product that can be ordered separately from the essential components of the product.


FIPS. Federal Information Processing Standard.

flash memory. A specialized version of erasable programmable read-only memory (EPROM) commonly used to store code in small computers.

hertz (Hz). A unit of frequency equal to one cycle per second. Note: In the United States, line frequency is 60 Hz, a change in voltage polarity 120 times per second; in Europe, line frequency is 50 Hz, a change in voltage polarity 100 times per second.

host. As regards to the IBM 4765 PCIe Cryptographic Coprocessor, the workstation into which the coprocessor is installed.

ICAT. Interactive Code Analysis Tool.

initial program load (IPL). (1) The initialization procedure that causes an operating system to commence operation. (2) The process by which a configuration image is loaded into storage. (3) The process of loading system programs and preparing a system to run jobs.

input/output (I/O). (1) Pertaining to input, output, or both. (A) (2) Pertaining to a device, process, or channel involved in data input, data output, or both.

Interactive Code Analysis Tool (ICAT). A remote debugger used to debug applications running within the IBM 4765 PCIe Cryptographic Coprocessor.

interface. (1) A boundary shared by two functional units, as defined by functional
characteristics, signal characteristics, or other characteristics as appropriate. The concept includes specification of the connection between two devices having different functions. (T) (2) Hardware, software, or both that links systems, programs, and devices.

**International Organization for Standardization (ISO).** An organization of national standards bodies established to promote the development of standards that facilitate the international exchange of goods and services; also, to foster cooperation in intellectual, scientific, technological, and economic activity.

**intrusion latch.** A software-monitored bit that can be triggered by an external switch connected to a jumper on the IBM 4765 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.

**I/O.** Input/output.

**IPL.** Initial program load.

**ISO.** International Organization for Standardization.

**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.

**M**

**MAC.** Message authentication code.

**master key.** In computer security, the top-level key in a hierarchy of 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 4765 PCIe Cryptographic Coprocessor designed to initialize the operating system and to control updates to flash memory.

**multi-user environment.** A computer system that supports terminals and keyboards for more than one user at the same time.

**N**

**National Institute of Science and Technology (NIST).** Current name for the US National Bureau of Standards.

**NIST.** National Institute of Science and Technology.

**node.** (1) In a network, a point at which one or more functional units connects channels or data circuits. (I) (2) The endpoint of a link or junction common to two or more links in a network. Nodes can be processors, communication controllers, cluster controllers, or terminals. Nodes can vary in routing and other functional capabilities.

**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.

**PCI.** Peripheral Component Interconnect.

**PCle.** Peripheral Component Interconnect Express.

**PCI-X.** Peripheral Component Interconnect eXtended.

Peripheral Component Interconnect Express (PCIe). A high-speed serial connection computer expansion card standard that replaces the PCI and PCI-X standards.

Peripheral Component Interconnect eXtended (PCI-x). A 64-bit version of the PCI, utilized in the IBM 4764 Cryptographic Adapter.

PKA. Public key algorithm.

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. (3) See also public key algorithm.

procedure call. In programming languages, a language construct for invoking execution of a procedure. (1) A procedure call usually includes an entry name and the applicable parameters.

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. (3) See also public key algorithm.

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.

Public Key Algorithm Manager (PKA_Mgr). A Linux extension that manages the IBM 4765 PCIe Cryptographic Coprocessor PKA processing hardware.

R

RAM. Random access memory.

random access memory (RAM). A storage device into which data is entered and from which data is retrieved in a non-sequential manner.

random number generator (RNG). A system designed to output values that cannot be predicted. Since software-based systems generate predictable, pseudo-random values, the IBM 4765 uses a hardware-based system to generate true random values for cryptographic use.

reduced instruction set computer (RISC). A computer that processes data quickly by using only a small, simplified instruction set.

return code. (1) A code used to influence the execution of succeeding instructions. (2) A value returned to a program to indicate the results of an operation requested by that program.

RNG. Random Number Generator.

RSA algorithm. A public key encryption algorithm developed by R. Rivest, A. Shamir, and L. Adleman.

S

security. The protection of data, system operations, and devices from accidental or intentional ruin, damage, or exposure.

system administrator. The person at a computer installation who designs, controls, and manages the use of the computer system.

Security Relevant Data Item (SRDI). Data that is securely stored by the IBM 4765 Cryptographic Adapter.

T

time-of-day (TOD) clock. A hardware feature that is incremented once every microsecond, and provides a consistent measure of elapsed time suitable for indicating date and time. The TOD clock runs regardless of whether the processing unit is in a running, wait, or stopped state.

throughput. (1) A measure of the amount of work performed by a computer system over a given period of time; for example, number of jobs-per-day. (2) A measure of the amount of information transmitted over a
network in a given period of time; for example, a network data-transfer-rate is usually measured in bits-per-second.

**TOD clock.** Time-of-day clock.

**U**

**utility program.** A computer program in general support of computer processes. (T)

**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.

**VPD.** Vital product data.

**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.

**X**

**xC, xCrypto.** IBM 4764 PCI-X Cryptographic Coprocessor.

**Numerics**

**4764.** IBM 4764 PCI-X Cryptographic Coprocessor.

**4765.** IBM 4765 PCIe Cryptographic Coprocessor.
## 10 Index

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbreviations</td>
<td>154</td>
</tr>
<tr>
<td>3.4 AES functions</td>
<td>42</td>
</tr>
<tr>
<td>xcAESKeyWrapX9102</td>
<td>47</td>
</tr>
<tr>
<td>API function categories, coprocessor-side</td>
<td>19</td>
</tr>
<tr>
<td>application key certificates</td>
<td>128</td>
</tr>
<tr>
<td>application sign-on function</td>
<td>25</td>
</tr>
<tr>
<td>3.10.2 authentication scheme</td>
<td>114</td>
</tr>
<tr>
<td>changes to segments 2 and 3</td>
<td>115</td>
</tr>
<tr>
<td>configuration end</td>
<td>115</td>
</tr>
<tr>
<td>configuration start</td>
<td>115</td>
</tr>
<tr>
<td>epoch end</td>
<td>116</td>
</tr>
<tr>
<td>examples</td>
<td>116</td>
</tr>
<tr>
<td>initialization</td>
<td>114</td>
</tr>
<tr>
<td>updates to segment 1</td>
<td>114</td>
</tr>
<tr>
<td>blinding values</td>
<td>82</td>
</tr>
<tr>
<td>buffer data, get</td>
<td>27</td>
</tr>
<tr>
<td>buffer data, put</td>
<td>29</td>
</tr>
<tr>
<td>byte order</td>
<td>4</td>
</tr>
<tr>
<td>categories, coprocessor-side API functions</td>
<td>19</td>
</tr>
<tr>
<td>common fields</td>
<td>124</td>
</tr>
<tr>
<td>application key</td>
<td>128</td>
</tr>
<tr>
<td>device key</td>
<td>127</td>
</tr>
<tr>
<td>IBM class root</td>
<td>127</td>
</tr>
<tr>
<td>operating system key</td>
<td>128</td>
</tr>
<tr>
<td>transition</td>
<td>128</td>
</tr>
<tr>
<td>chained operations</td>
<td>41</td>
</tr>
<tr>
<td>class root certificates, IBM</td>
<td>127</td>
</tr>
<tr>
<td>class root descriptions</td>
<td>132</td>
</tr>
<tr>
<td>class root keypairs, IBM</td>
<td>129</td>
</tr>
<tr>
<td>clear coprocessor intrusion latch</td>
<td>111</td>
</tr>
<tr>
<td>clear low battery warning latch</td>
<td>112</td>
</tr>
<tr>
<td>clock, set</td>
<td>107</td>
</tr>
<tr>
<td>communication functions</td>
<td>25</td>
</tr>
<tr>
<td>communication-related</td>
<td></td>
</tr>
<tr>
<td>get request</td>
<td>27</td>
</tr>
<tr>
<td>put buffer data</td>
<td>29</td>
</tr>
<tr>
<td>query buffer status</td>
<td>34</td>
</tr>
<tr>
<td>3.9 configuration functions</td>
<td>107</td>
</tr>
<tr>
<td>date/time</td>
<td>107</td>
</tr>
<tr>
<td>privileged operations</td>
<td>107</td>
</tr>
<tr>
<td>xcGetConfig</td>
<td>107</td>
</tr>
<tr>
<td>3 co processor API</td>
<td>19</td>
</tr>
<tr>
<td>function categories</td>
<td>19</td>
</tr>
<tr>
<td>coprocessor architecture</td>
<td>113</td>
</tr>
<tr>
<td>coprocessor configuration functions</td>
<td>21</td>
</tr>
<tr>
<td>coprocessor functions</td>
<td>19</td>
</tr>
<tr>
<td>coprocessor-generated keypairs</td>
<td>129</td>
</tr>
<tr>
<td>data format</td>
<td>23</td>
</tr>
<tr>
<td>decipher functions</td>
<td></td>
</tr>
<tr>
<td>AES</td>
<td>42</td>
</tr>
<tr>
<td>DES</td>
<td>61</td>
</tr>
<tr>
<td>DES eight-byte</td>
<td>64</td>
</tr>
<tr>
<td>RSA</td>
<td>77</td>
</tr>
<tr>
<td>TDES</td>
<td>57</td>
</tr>
<tr>
<td>TDES eight-byte</td>
<td>66</td>
</tr>
<tr>
<td>DES encryption/decryption/MAC</td>
<td>61</td>
</tr>
<tr>
<td>DES functions</td>
<td>20, 57</td>
</tr>
<tr>
<td>DES weak keys</td>
<td>147</td>
</tr>
<tr>
<td>descriptions, class root</td>
<td>132</td>
</tr>
<tr>
<td>device names</td>
<td>23</td>
</tr>
<tr>
<td>devices names</td>
<td>130</td>
</tr>
<tr>
<td>DSA key tokens</td>
<td>84</td>
</tr>
<tr>
<td>DSA signature tokens</td>
<td>85</td>
</tr>
<tr>
<td>epochs and configurations</td>
<td>131</td>
</tr>
<tr>
<td>error code formatting</td>
<td>145</td>
</tr>
<tr>
<td>examples, authentication scheme</td>
<td></td>
</tr>
<tr>
<td>initial certificate chain</td>
<td>117</td>
</tr>
<tr>
<td>application generates configuration key</td>
<td>118</td>
</tr>
<tr>
<td>application generates new configuration key</td>
<td>120</td>
</tr>
<tr>
<td>configuration keypair and epoch keypair created</td>
<td>122</td>
</tr>
<tr>
<td>foreign application loaded</td>
<td>123</td>
</tr>
<tr>
<td>miniboot updated</td>
<td>121</td>
</tr>
<tr>
<td>operating system updated</td>
<td>119</td>
</tr>
<tr>
<td>exponent types, RSA</td>
<td>68</td>
</tr>
<tr>
<td>functions</td>
<td></td>
</tr>
<tr>
<td>AES</td>
<td>42</td>
</tr>
<tr>
<td>DES</td>
<td>57</td>
</tr>
<tr>
<td>hash</td>
<td>36</td>
</tr>
<tr>
<td>generate DSA keypair</td>
<td>86</td>
</tr>
<tr>
<td>generate random number</td>
<td>104</td>
</tr>
<tr>
<td>glossary</td>
<td>156</td>
</tr>
<tr>
<td>hash functions</td>
<td>20, 36</td>
</tr>
<tr>
<td>header files, coprocessor API</td>
<td>22</td>
</tr>
<tr>
<td>host and coprocessor interaction</td>
<td>2</td>
</tr>
<tr>
<td>2 host API</td>
<td>6</td>
</tr>
<tr>
<td>introduction</td>
<td>6</td>
</tr>
<tr>
<td>3.2 host communication functions</td>
<td>19, 25</td>
</tr>
<tr>
<td>xcAttachWithCDUOption</td>
<td>25</td>
</tr>
<tr>
<td>xcGetRequest</td>
<td>27</td>
</tr>
<tr>
<td>xcInitMappings</td>
<td>26</td>
</tr>
<tr>
<td>xcPutReply</td>
<td>29</td>
</tr>
<tr>
<td>IBM root public keys</td>
<td>149</td>
</tr>
<tr>
<td>initialization</td>
<td>114</td>
</tr>
</tbody>
</table>
outs to segment 1.....................................114
UDX environment...............................................1
system architecture.............................................1
symmetric key functions...................................20
software attacks..................................................4
sample code, coprocessor API.........................22
sample applications............................................5
RSA key tokens...................................................6
RSA key generation....................................72
RSA key tokens...............................................68
sample applications..........................................5
sample code, coprocessor API....................22
software attacks..................................................4
symmetric key functions...................................20
system architecture.............................................1
timestamps..................................................132
traffic, minimizing...........................................2
UDX environment..............................................1
updates to segment 1.....................................114
virtual packets..................................................3
wrapping operation............................................46

keypair names.....................................................129
large integer modular math functions.......21, 91
large integers..................................................91
layer names..................................................130
mapped kernel buffers................................23
NIST formatted buffer......................................46
notices..................................................152
OA certificates................................................123
3.10 Outbound Authentication functions.....22, 113
application generates configuration key....118
application generates new configuration key
.............................................................120
authentication scheme overview...............114
certificate structure........................................124
changes to segments 2 and 3...................115
configuration end...........................................115
configuration keypair and epoch keypair
created.....................................................122
configuration start......................................115
coprocessor architecture........................113
ePOCH end...................................................116
foreign application loaded........................123
initial certificate chain.........................117
miniboot updated..................................121
operating system updated........................119
xcOAGenerate...........................................136
cxoAPrivOp..................................................140
principal agents...........................................1
public key algorithm functions.................21
public key Algorithm functions...............68
random number generator functions.........21, 94
related publications........................................viii
request priority...........................................4
RNG functions...............................................104
xcRandomNumberGenerate..........................104
RSA key generation....................................72
RSA key tokens...............................................68
sample applications..........................................5
sample code, coprocessor API....................22
software attacks..................................................4
symmetric key functions...................................20
system architecture.............................................1
timestamps..................................................132
traffic, minimizing...........................................2
UDX environment..............................................1
updates to segment 1.....................................114
virtual packets..................................................3
wrapping operation............................................46

X9.102 formatted buffer...............................46
xcAdapterCount.............................................8
xcAES............................................................42
xcAESKeyUnwrapNIST..................................55
xcAESKeyUnwrapX9102...................................50
xcAESKeyWrap.............................................46
xcAESKeyWrapNIST.....................................53
xcAESKeyWrapX9102...................................47
xcAttachWithCDUOption.............................25
xcClearLatch.............................................111
xcClearLowBatt...........................................112
xcCloseAdapter...........................................14
xcComputeBlindingValues........................82
xcDES............................................................61
xcDES3Key..................................................66
xcDES8bytes.............................................64
xcDetach...................................................33
xcDRBGgenerate.........................................96
xcDRBGinstantiate....................................94
xcDRBGseed...............................................99
xcDRBGtest...............................................101
xcDRBGuninstantiate.................................103
xcDSA.............................................................89
xcDSAKeyGenerate.......................................86
xcGetAdapterData.......................................15
xcGetConfig...............................................107
xcGetHardwareVersion...............................17
xcGetRequest...............................................27
xcInitMappings............................................26
xcKillMappings............................................32
xcModMath...................................................92
xcOAGenerate............................................139
xcOAPrivOp.................................................140
xcOADelete.................................................139
xcODAGenerate............................................136
xcODAGenerate.............................................140
xcOAInit.....................................................135
xcOAGetCert...............................................135
xcOAGetDir...............................................133
xcOAPrivOp..................................................140
xcOASStatus...............................................142
xcOpenAdapter..........................................18
xcPutReply...................................................29
xcQueryMRBstatus....................................34
xcRandomNumberGenerate..........................104
xcRandomNumberGenerate..........................104
xcRequest...................................................11
xcResetAdapter..........................................18
xcRSA...........................................................77
xcRSAKeyGernerate.....................................72
xcSha1.......................................................36
xcSha2.......................................................39
xcTDES.......................................................57
xcTestRandomNumber...............................106