IBM 4764 PCI-X Cryptographic Coprocessor
CCA User Defined Extensions
Reference and Guide
Note!

Before using this information and the products it supports, be sure to read the general information under Appendix I, “Notices.”

Second Edition (February, 2007)

IBM does not stock publications at the address given below. This and other publications related to the IBM 4764 PCI-X Cryptographic Coprocessor can be obtained in PDF format from the Library page located at http://www-03.ibm.com/security/cryptocards.

Reader’s comments can be communicated to IBM by using the e-mail address located on the Web site at http://www-03.ibm.com/security/cryptocards, or you can respond by mail to:

Department VM9A, MG81
IBM Corporation
8501 IBM Drive
Charlotte, NC 28262-8563
U. S. A.

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 2005, 2007. All rights reserved.

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.
Byte Order Changes .............................................................. H-1
Makefile Changes .............................................................. H-1
  Changes for the coprocessor makefile ................................ H-1
  Changes for the host workstation makefile ...................... H-2
Changes to function calls ................................................ H-2
  In zOS ....................................................................... H-2
  On the coprocessor ...................................................... H-2
Data Block Changes ............................................................ H-2
Structure Changes .............................................................. H-3

Notices ........................................................................ I-1
Copying and Distributing Softcopy Files ............................... I-1
Trademarks ......................................................................... I-2

List of Abbreviations and Acronyms .................................... X-1

Glossary .......................................................................... X-3
About This Book

The IBM 4764 PCI-X Cryptographic Coprocessor CCA User Defined Extensions Reference and Guide describes the Common Cryptographic Architecture (CCA) application programming interface (API) function calls that are available to user defined extensions to CCA. A user defined extension (UDX) allows a developer to add customized operations to IBM’s CCA Support Program. UDXs are written and invoked in the same manner as base CCA functions and have access to the same internal functions and services as the CCA Support Program.

This document begins with an overview of the UDX programming environment and the sample files that are provided for use by UDX authors. The remainder of the document is a reference manual that describes a variety of functions that a UDX developer may exploit. The callable functions may be grouped into four classes:

1. Functions that may be called by the portion of a UDX that runs inside the PCI-X cryptographic coprocessor.
2. Functions that may be called by the portion of a UDX that runs on a host workstation.
3. Functions that are available both inside the coprocessor and on the host workstation.
4. Functions that may be called by the portion of a UDX that runs on a zSeries system.

Most of the functions are in the first class.

The primary audience for this manual is developers who need to write a UDX. This manual should be used in conjunction with the manuals listed under “CCA Support Program Publications” on page ix and “Custom Software Publications” on page ix.

Prerequisite Knowledge

The reader of this book should understand how to perform basic tasks (including editing, system configuration, file system navigation, and creating application programs) on the host machine and in the Linux environment, and should understand the use of IBM’s CCA Support Program (as described in the IBM 4764 PCI-X Cryptographic Coprocessor CCA Support Program Installation Manual and the IBM 4764 PCI-X Cryptographic Coprocessor CCA Basic Services Reference and Guide). The reader should also understand the zSeries application environment (as described in the z/OS ICSF Application Programmer’s Guide and the z/OS ICSF System Programmer’s Guide). Familiarity with the xCrypto application development process (as described in the IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide) is also required.

Organization of This Book

Chapter 1, “Understanding the UDX Environment” discusses the design of the CCA application and the separation of the CCA API into host-side and coprocessor-side components.

Chapter 2, “Building a CCA User Defined Extension” discusses how to build each portion of a UDX.

Chapter 3, “xCrypto Functions” summarizes the secure cryptographic coprocessor (xCrypto) API on top of which IBM’s CCA coprocessor application modules are built. A UDX may use the xCrypto API if so desired.
Chapter 4, “Communications Functions” describes the functions that allow the piece of a UDX that runs on the host to exchange information with the piece of the UDX that runs in the coprocessor.

Chapter 5, “Function Control Vector Management Functions” describes the functions that allow a UDX to determine which cryptographic operations have been authorized by the CCA function control vector and how long certain cryptographic keys may be.

Chapter 6, “CCA Master Key Manager Functions” describes the functions that allow a UDX to access and manipulate the CCA master key registers, which are used to encrypt and decrypt data and keys using various forms of the Data Encryption Standard (DES) algorithm.

Chapter 7, “SHA-1 Functions” describes the functions that a UDX can use to compute the hash of a block of data using the Secure Hash Algorithm (SHA-1).

Chapter 8, “DES Utility Functions” describes the functions that a UDX can use to manipulate and obtain information about key tokens and other cryptographic structures.

Chapter 9, “RSA Functions” describes the functions that a UDX can use to perform public key cryptographic operations using the RSA (Rivest-Shamir-Adleman) algorithm.

Chapter 10, “CCA SRDI Manager Functions” describes the functions that an application can use to store and retrieve data in the coprocessor’s nonvolatile memory areas (flash EPROM).

Chapter 11, “Access Control Manager Functions” describes the functions that a UDX can use to manipulate a user’s permissions, change authentication (logon) procedures, or obtain information about permissions and users on the coprocessor.

Chapter 12, “Cache Management Functions” describes the functions that a UDX can use to implement an on-board cache of secure data.

Chapter 13, “Miscellaneous Functions” describes several assorted utility functions available to a UDX.

Appendix A, “UDX Sample Code - Host Piece - Service” contains the host-side portion of a sample UDX.

Appendix B, “UDX Sample Code - Host Piece - Service Stub” contains the host-side, service stub portion of a sample UDX.

Appendix C, “UDX Sample Code - Host Piece - CSFPCI Post-Processing Exit” contains the host-side post-processing exit portion of a sample UDX.

Appendix D, “UDX Sample Code - Coprocessor Piece” contains the coprocessor-side portion of a sample.


Appendix F, “Reserved Values” lists the values reserved for UDX developers.

Appendix G, “Data Structures” contains useful data structures from the toolkit header files.

Appendix H, “Moving a UDX from the IBM 4758 to the IBM 4764 Coprocessor” contains necessary changes to the UDX code when transferring code from an IBM 4758 to a IBM 4764.

Appendix I, “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:

- File names, function names, and return codes are presented in **bold** type.
- Variable information and parameters are presented in **fixed-space** type.
- Web addresses are presented in **italic** type.

**Related Publications**

Many of the publications listed below under “General Interest,” “CCA Support Program Publications” and “Custom Software Publications” are available in Adobe Acrobat** portable document format (PDF) at [http://www-03.ibm.com/security/cryptocards](http://www-03.ibm.com/security/cryptocards).


Click **Library** to view or print books, or to order available hardcopy publications.

**General Interest**

The following publications may be of interest to anyone who needs to install, use, or write applications for a PCI-X Cryptographic Coprocessor:

- IBM 4764 PCI-X Cryptographic Coprocessor General Information Manual (version -01 or later)
- IBM 4764 PCI-X Cryptographic Coprocessor Installation Manual

**CCA Support Program Publications**

The following publications may be of interest to readers who intend to use a PCI-X Cryptographic Coprocessor to run IBM’s Common Cryptographic Architecture (CCA) Support Program:

- IBM 4764 PCI-X Cryptographic Coprocessor CCA Basic Services Reference and Guide
- z/OS Integrated Cryptographic Service Facility Application Programmer’s Guide

**Custom Software Publications**

The following publications may be of interest to persons who intend to write applications that will run on a PCI Cryptographic Coprocessor:

- IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Interface Reference
- IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide
Cryptography Publications

The following publications describe cryptographic standards, research, and practices applicable to the PCI X Cryptographic Coprocessor:

- ANSI X9.31 Public Key Cryptography Using Reversible Algorithms for the Financial Services Industry
- IBM Systems Journal Volume 32 Number 3, 1993, G321-5521
- IBM Journal of Research and Development Volume 38 Number 2, 1994, G322-0191
- USA Federal Information Processing Standard (FIPS):
  - Data Encryption Standard, 46-1-1988
  - Secure Hash Algorithm, 180-1, May 31, 1994
  - Cryptographic Module Security, 140-1
Other IBM Cryptographic Product Publications

The following publications describe products that utilize the IBM Cryptographic Architecture (CCA) Application Program Interface (API).

- IBM Transaction Security System General Information Manual, GA34-2137
- IBM Transaction Security System Basic CCA Cryptographic Services, SA34-2362
- IBM Transaction Security System Finance Industry CCA Cryptographic Programming, SA34-2364
- IBM 4755 Cryptographic Adapter Installation Instructions, GC31-4503
- IBM Transaction Security System Physical Planning Manual, GC31-4505
- IBM ICSF/MVS General Information, GC23-0093
- OS/390 Integrated Cryptographic Service Facility Overview, GC23-3972
- OS/390 ICSF Trusted Key Entry Workstation User’s Guide, SC23-3978

Summary of Changes

This is the first edition of the IBM 4764 PCI-X Cryptographic Coprocessor CCA User Defined Extensions Reference and Guide.

Changes made to this second edition in May, 2006 include:

- Chapter 4—Added notes to functions that run AMODE(64) in HCR7720 and later versions of ICSF.
- Chapter 8—Added possible key types for CasBuildCv and CasBuildToken.

Changes made to this second edition in February, 2007 include:

- Chapter 1—Added AIX information.
- Chapter 2—Added AIX information and paths for source files.
- Appendix H—Updated coprocessor and host makefile information.
Chapter 1.

Understanding the UDX Environment

The UDX Development Toolkit for the IBM 4764 provides scaffold code, object modules, header files, and macros that you can use to extend the IBM-developed Common Cryptographic Architecture (CCA) application program which employs the IBM 4764 PCI-X Cryptographic Coprocessor. You can use as much or as little of the CCA application function as required to meet your processing requirements.

You can write a user defined extension (UDX) to be installed on an IBM @server zSeries 990 server or z9 server or on an AIX, Linux, or Windows 2003 workstation. Implementation on a zSeries server requires initial testing on a workstation, so although the host code is quite different on the different systems, the following explanation is split into the workstation form followed by the zSeries form.

This chapter explains the design of the CCA “middleware” application. If you are not familiar with the CCA implementation for the coprocessor, you should first read portions of the IBM 4764 PCI-X Cryptographic Coprocessor CCA Basic Services Reference and Guide. In particular, read chapter 1, the introductory information of chapters 2 through 8, and become aware of the material in appendices B, C, and D.

If you are implementing a UDX for zSeries, you should also become familiar with the implementation of ICSF and the zSeries cryptographic engines. You can do this by becoming familiar with the information in the z/OS ICSF System Programmer’s Guide and the z/OS ICSF Application Programmer’s Guide.

This manual also assumes that you are familiar with the techniques for creating and testing coprocessor application programs as described in the IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide. This manual describes the xCrypto API on which the CCA application is built. You may also benefit from understanding the services that you can obtain from the Linux operating system on the xCrypto card.

The CCA architecture requires that security-sensitive functions are carried out in an environment where secret or private quantities can safely appear in the clear and where the design of the processing functions cannot be altered by an adversary.

A coprocessor application program operates in such an environment; however, the confidentiality of secret or private quantities (for example, cryptographic keys or computational values) is also the responsibility of the application program design.

The CCA application operates as a request/response mechanism. Once initialized by the Linux operating system as a result of a coprocessor reset sequence, the CCA application within the coprocessor waits for an external request. The application then performs the requested function and returns a response. The application retains persistent data as a set of security relevant data items (SRDI). The application stores SRDIs in RAM memory, with a backup copy retained in encrypted form in flash memory.

The CCA verbs (callable services) that a host application can request are generally serviced, on a one-for-one basis, by a command processor portion of coprocessor application code. A common infrastructure is employed to format a verb request, transport the request to the coprocessor, dispatch the command

---

1. A few CCA verbs are implemented as subroutines in the top layer of CCA host code and do not send a request to the coprocessor.
processor, and return the reply to the host. Command processors and the top layer of CCA host code (security application program interface (SAPI) or z/OS ICSF) make extensive use of a set of common subroutines described in this manual.

The code that implements a UDX to CCA can be separated into two distinct pieces. One (the “host piece”) runs on the host, either as a shared object in the workstation or link-edited into a load module and installed in an APF authorized library on the zSeries server. The other (the “coprocessor piece”) is linked with a library containing IBM’s CCA coprocessor application modules and downloaded to the coprocessor. The host piece converts requests for service from the user’s application into messages to be sent to the coprocessor. These messages are received by the CCA application and routed to the appropriate (CCA or UDX) command processor.

**Structure of a UDX on a Workstation**

Figure 1-1 depicts the major elements of code that form the CCA workstation implementation for the UDX test environment. The boxes with dotted lines designate the UDX components. Each block represents a section of the runtime code. Blocks one through six are host-system shared libraries with block six actually split between a shared object and the physical device driver. An overview of these code blocks follows.

![Figure 1-1. View of Workstation CCA with User Defined Extensions](image-url)
Security API (SAPI)

The Security API (SAPI) code, **libcsulsapi.so**, contains the CCA verb entry points. On input SAPI gathers the request information from the variables identified by the verb parameters and constructs a standardized set of control blocks for communication to the coprocessor CCA application. The formatted request is then passed to the security server (SECY) layer. On output, the formatted reply is parsed and the caller’s variables are updated with the verb results.

The request is communicated using a Cooperative Processing Request/Reply Block (CPRB) data structure and an appended, variable-length *request parameter block*. The formatted reply is likewise communicated with a CPRB and an appended *reply parameter block* of the same general structure as the request block.

The fixed-length CPRB structure carries a primary function code, return and reason code values, and pointers to, and lengths of, the request and reply parameter blocks and data to be read to/from the coprocessor. The variable-length request and reply parameter blocks (see Figure 1-4 on page 1-11) carry:

- A sub-function code, the identifier of the command processor
- The rule-array elements, encoded in ASCII
- Verb-unique data (VUD)
- Cryptographic key information, key labels (on the workstation) or key tokens, in “key blocks.”

The subroutines used to construct and to parse these control blocks are used by all of the verb routines in SAPI. These same subroutines are entry points that can be called by the UDX-SAPI code. See Chapter 4, “Communications Functions” on page 4-1 for more information.

The CCA SAPI routines perform minimal checking on the input variables. The design concept is to perform almost all variable checking within the coprocessor. SAPI is responsible for ensuring that character-based control and data information is encoded in the manner expected by the coprocessor application, regardless of the encoding of this data on the host system. Likewise, SAPI must ensure that integers and other numbers are communicated in the form expected by the coprocessor application; in general, integers must be in big-endian format with the most-significant byte first.

Because the CCA SAPI code is compiled on an Intel system for use on a PowerPC system, C macros are used to ensure that the integers exchanged with the coprocessor are in big-endian format (native byte order for the PPC coprocessor). Note, however, that most CCA data structures, such as RSA key tokens, define integer values as big-endian (S/390 integer format) quantities. In these cases, the coprocessor and application program are responsible for ensuring and interpreting the appropriate integer byte-order.

UDX-SAPI

The UDX callable services are assumed to be analogous to CCA services. Your UDX host-piece code constructs and parses CPRB and request and reply parameter blocks using the same subroutines as employed by the SAPI code.

Once the CPRB and request parameter block are constructed, you use the CSNC_SP_SCSRFBSS() subroutine to pass control to the security server (SECY) layer. Upon regaining control, your code should update the caller’s variables with the information that is parsed from the CPRB and reply parameter block.

---

2. Typical CCA host code file names begin with CSUx where the “x” is “N” for Windows NT, “E” for OS/2, “F” for AIX, and “L” for Linux.
See “Appendix E, UDX Sample Code - Workstation Host - Test Code on page E-1.”

Security Server (SECY)

The Security Server (SECY), *libcsulsecy.so*, receives control from SAPI with a pointer to the CPRB. The security server examines the key-block fields of the request and reply parameter blocks to determine if the key storage server should be called to allocate, delete, or list labels in key storage, or to fetch or store key records under key labels already existing in key storage. The security server also passes the name of the key storage files to the directory server. On input, except for a few key-storage services which do not require use of the coprocessor, the security server calls the adapter interface after completing any required key storage actions. Likewise, on output the adapter interface returns control to the security server which completes any required key storage requests and then returns control to SAPI. Information in a key block header (see “Key Blocks” on page 1-15) triggers the security server to process a key block.

**Note:** Coprocessor executables built with the *libcsuzlib.a* file (that is, files built for use on a z/OS system) are incompatible with the key storage server. If you are testing code for use on a z/OS system, the Security Server must not receive requests to access the key storage files on the workstation.

Key Storage Server

The key storage server, *libcsuldir.so*, receives control from the security server with pointers to the key storage file names and to the key block on which it should take action. The server is responsible for opening and closing the directory files, allocating records in the indexed sequential files, listing the file names, and fetching and storing key tokens. Separate files are maintained for the DES fixed-length records and the variable-length PKA (public key architecture, RSA) records.

**Note:** Coprocessor executables built with the *libcsuzlib.a* file (that is, files built for use on a z/OS system) are incompatible with the key storage server. If you are testing code for use on a z/OS system, the Directory Server must not access the key storage files.

Adapter Interface

The adapter interface, *libcsulcall.so*, receives control from the security server and examines the CPRB to determine the nature of the call it will create to the device driver.

All CCA requests to the adapter interface require that the CPRB and request parameter block be read to the coprocessor. The adapter interface layer examines the CPRB request and reply data block pointer and length fields and calls the device driver so that the coprocessor application program can cause data transfers from/to the identified fields. The adapter interface layer creates the control blocks and issues the I/O request to the device driver.

Device Driver and Access Layer

The device driver code is split between a shared object and a physical device driver. The API and function of the device driver is explained in the *IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Interface Reference*. The device driver and Linux operating system work together to ensure that the reply to a request is routed back to the source process and thread that initiated the associated request.
Structure of a UDX on a zSeries System

Figure 1-2 on page 1-5 depicts the major elements of code that form the CCA implementation for zSeries. Each block is described below the figure. Blocks 1 through 5 are functions which execute on the zSeries host; blocks 7 through 11 are functions which execute on the PCI-X Cryptographic Coprocessor.

![Diagram of ICSF/CCA with User Defined Extensions](image)

*Figure 1-2. View of ICSF/CCA with User Defined Extensions*
**UDX Service Stub**

The service stub connects the application program with the callable service. Each callable service which is part of ICSF has a service stub. Each UDX callable service must also have a related service stub. A callable service receives parameters from the application program when the program calls the service stub associated with the service. The service stub performs a space-switch PC (Program Call) operation to transfer control from the application’s address space to ICSF’s address space. The parameters that are associated with a callable service provide the only communication between the application program and ICSF. See Appendix B, “UDX Sample Code - Host Piece - Service Stub” on page B-1.

**Callable Service**

ICSF provides access to cryptographic functions through callable services, which are also known as verbs. A callable service is a routine that receives control using a CALL statement in an application language. The callable service contains the CCA verb entry point. On input, the callable service gathers the request information from the variables identified by the verb parameters and constructs a standardized set of control blocks for communication to the coprocessor CCA application. The formatted request is then passed to the cryptographic coprocessor interfaces layer (\(^5\)). On output, the formatted reply is parsed and the caller’s variables are updated with the verb results.

The request is communicated from the host to the coprocessor using a Cooperative Processing Request/Reply Block (CPRB) data structure and an appended, variable-length request parameter block. The formatted reply is likewise communicated from the coprocessor to the host with a CPRB and an appended reply parameter block of the same general structure as the request block.

The fixed-length CPRB structure carries a primary function code, return and reason code values, and pointers to, and lengths of, the request and reply parameter blocks and data to be read to/from the coprocessor. The variable-length request and reply parameter blocks (see Figure 1-4 on page 1-11) carry:

- A sub-function code, the identifier of the command processor
- The rule-array elements, encoded in ASCII
- Verb-unique data (VUD)
- Cryptographic key information (key tokens) in “key blocks”

The subroutines used to construct and to parse these control blocks are used by all of the ICSF callable services. These same subroutines are entry points that can be called by the UDX callable services. See Chapter 4, “Communications Functions” on page 4-1 for a description of the CSFGAPSI (Communication interface between services and coprocessor) function.

The ICSF callable service routines perform minimal checking on the input variables. The design concept is to perform almost all variable checking within the coprocessor. The callable service routine must ensure that character-based control and data information is encoded in the manner expected by the coprocessor application, regardless of the encoding of this data on the host system. Likewise, the callable service routine must ensure that integers and other numbers are communicated in the form expected by the coprocessor application. In general, integers must be in big-endian format. Also, most CCA data structures, such as key tokens, define integer values as big-endian (zSeries integer format) quantities.

The UDX callable services are analogous to the ICSF-provided CCA callable services. The UDX host-piece constructs and parses CPRB and request and reply parameter blocks using the same subroutines as employed by the ICSF callable service routines. Once the CPRB and request parameter block are constructed, the UDX callable service routine uses common subroutines to interface with the cryptographic
coprocessor hardware. See Chapter 4, “Communications Functions” on page 4-1 for a description of the CSFGAPSI (Communication interface between services and the coprocessor) function. See Appendix A, UDX Sample Code - Host Piece - Service on page A-1.

Cryptographic Key Data Set (CKDS)

The Cryptographic Key Data Set (CKDS) is a VSAM data set that contains DES encrypting keys used by an installation. Besides the encrypted key value, an entry in the CKDS contains information about the key, such as key type, creation date and time, last update date and time. If a UDX callable service (UDX host piece) needs to pass key data to the UDX command processor (UDX coprocessor piece), the callable service must resolve a key label into a key token. This involves reading the key record from the CKDS. ICSF maintains an in-storage copy of the CKDS to improve performance of key access. The callable service uses the CSFACKDS subroutine to access the in-storage CKDS. (See Chapter 4, “Communications Functions” on page 4-1 for a description of the CSFACKDS (Access ICSF Cryptographic Keys Data Set) function.)

Public Key Data Set (PKDS)

The Public Key Data Set (PKDS) is a VSAM data set that contains PKA encrypting keys used by an installation. If a UDX callable service (UDX host piece) needs to pass PKA key data to the UDX command processor (UDX coprocessor piece), the callable service must resolve a key label into a key token. This involves reading the key record from the PKDS. The callable service uses the CSFAPKDS subroutine to access the PKDS. (See Chapter 4, “Communications Functions” on page 4-1 for a description of the CSFAPKDS (Access ICSF Public Key Data Set) function.)

Cryptographic Coprocessor Interfaces

ICSF provides service modules which interface with the cryptographic coprocessor hardware. The module which interfaces with the PCI-X cryptographic coprocessor is CSFGCCPN. This module receives control from the callable service routines and examines the CPRB data passed as input to determine the nature of the call it will create to the PCI-X cryptographic coprocessor. The interface module makes use of machine instructions to cause zSeries Licensed Internal Code (LIC) to receive control to pass control via the PCI-X Bus to CCA Manager in the PCI-X cryptographic coprocessor.
Structure of a UDX on the Coprocessor

Figure 1-3 on page 1-8 depicts the major elements of code that form the CCA implementation on the PCI-X cryptographic coprocessor.

Figure 1-3. View of CCA with User Defined Extensions on Coprocessor

**Linux Operating System Services**

The init.sh shell file is run automatically in Segment 3 following a reset sequence (which also runs the init.sh file in Segment 2). The csulcca application’s entry point is called by the startcca function and CCA registers itself with the xCrypto Communications Manager.

When the xCrypto Communications Manager receives a request from the host it checks for a registered application identifier; the identifier is a constant prearranged between the adapter interface layer and the CCA application. CCA host requests include the CPRB and request parameter block. The application interface layer presents sufficient information that the CCA Dispatcher can request the Communications Manager to obtain the CPRB and request parameter block.

Other low-level services for date and time, storage of data in BBRAM, and communication with external functions (particularly through the serial port) are available through Linux (C) functions to the UDX code. Note that CCA service subroutines are already available to perform many common functions and therefore command processor code generally does not call Linux functions directly.

**CCA Dispatcher**

When the Communications Manager responds to the CCA dispatcher’s request for input because of the receipt of a host request, the dispatcher obtains the CPRB and request parameter block. The dispatcher also locates the role that governs the processing of the CCA request, either the default role or the role associated with a logged-on profile.

Each thread of each workstation host process can logon to a role through an associated profile. However, a single profile can be associated with only one host thread at a time; a correct logon to a profile from another thread will be honored and a new session key generated without any indication of this action reported to the “older” logged-on thread (until and if the older thread makes a new request).
The dispatcher uses the sub-function code in the first two bytes of the request parameter block in a table lookup operation to locate a UDX command processor entry point. If a match is not found, the dispatcher checks the CCA entry point table for a match. (Of course, if again no match is found, the dispatcher constructs a reply CRPB and fills it with a return and reason code indicating that no such function exists.) The dispatcher then calls the command processor and passes pointers to the CPRB and request parameter block, and to the role that governs processing for this request.

Later the command processor returns control to the dispatcher which uses the Communications Manager to transfer the reply CPRB, and (optionally) the reply parameter block, back to the host.

The CCA implementation is multi-threaded. The multi-threading design permits overlapped operation by the CPU, the modular-exponentiation hardware, the DES/TDES/SHA-1 hardware, the random number (bit) generator, and the external communication hardware.

The main() function in the cammgr.c module receives control when the Linux OS loads the CCA application. The main() function calls various initialization functions in the other managers (for example, to obtain the master keys and role security relevant data items (SRDI), and so on), and then initializes 6 (high priority) threads that each wait for the operating system to deliver requests to the CCA message queue, as well as two (low priority) requests which wait for requests to generate RSA keys. Finally, main() waits on any catastrophic event.

Each of the 6 (high priority) threads are available to process incoming requests and runs the same code. When a request is received, one of the threads removes the request from the queue, determines which profile and role applies, examines the subfunction code, and calls an appropriate CCA or UDX command processor. When the command processor completes, the code passes the reply to the Communication Manager and again enqueues on the inbound message queue.

The command decoder calls the UDX entry points. Following these rules ensures that your UDXs are thread-safe.

- Do not use static variables.
- Define semaphores globally, for example, define all semaphores in one particular UDX source code module which can be reached by an extern statement in other UDX source code modules.
- Create no more than 50 semaphores, as the table for storing semaphores is limited.

CCA Services

The CCA application supplies many subroutines that command processors use to perform functions in a consistent manner. These routines are described later in this manual. The command processors also make use of three “managers” that localize certain classes of function to the managers:

SRDI Manager

The CCA coprocessor application code generally uses the SRDI Manager to access information that is held in persistent flash memory. The manager is responsible for serializing the use of the SRDIs to accommodate the multi-tasking environment. See Chapter 10, “CCA SRDI Manager Functions” on page 10-1.
Access Control Manager

All operations on roles and profiles are carried out by the Access Control Manager. Command processors call the manager to determine if individual control points are authorized. When a command processor is designed, one or more control points are assigned, as required for security purposes, to authorize function within the command processor. The sample SAPI code (Appendix E, UDX Sample Code - Workstation Host - Test Code on page E-1) documents a range of control points (and also reason codes and subfunction codes) reserved for UDX developers.

Master Key Manager

All operations pertaining to the master keys are performed by this manager. Code in other parts of CCA does not access the master key values directly, but rather calls the manager for operations that affect or use the master keys and their registers. See Chapter 6, “CCA Master Key Manager Functions” on page 6-1. Note that code built with the libcsuzlib.a library for a z/OS system cannot affect the master key registers or their contents, which may only be changed by TKE functions. All functions which affect the master key registers are disabled on a z/OS UDX.

Note that all of the CCA coprocessor code and UDX code operates as user 500 on the Linux operating system, rather than as “root”; therefore, UDX code has access to memory areas belonging to any portion of CCA. As additional code is created, it should be inspected to ensure that it performs only the intended function and accesses only information appropriate to the intended function.

CCA Command Processors

In general, each CCA verb results in a call to one command processor, the code in the coprocessor CCA application that performs the function unique to a verb.

Command processor code can call any of the other CCA subroutines and manager functions as well as functions available on the Linux operating system. In general, a command processor will perform the following steps. See Appendix D, UDX Sample Code - Coprocessor Piece on page D-1.

- Copy the request CPRB to form the reply CPRB in the memory provided by the dispatcher.
- Set the return code and reason code to 0, 0 using Cas_proc_retc() and copy the sub-function code into the reply block.
- Check that the caller is authorized to use this domain.
- Initialize the master key selector to the set of keys for this domain.
- Call the Access Control Manager to determine if the appropriate control point is authorized using CHECK_ACCESS_AUTH().
- Because most command processors will need to decrypt or encrypt a key, determine that there is a valid master key(s) using mkmGetMasterKeyStatus().
- Check that the request parameter block is formed in a valid manner by calling parm_block_valid().
- Check the length of the rule array data area by examining the rule array area length bytes. For CCA, this value is 8x+2 where x=0, 1, ...,n; however, you could make this portion of the request parameter block contain data of almost any length. You can check the rule array elements using rule_check().
- Check the length of any VUD, data formatted to the needs of the command processor. You should establish addressability to the VUD using a structure definition.
- Check the length and content of the zero or more key blocks.
- Perform the desired command function.
• Determine that the reply will not exceed the permissible reply size.
• Fill in the reply block with the rule array length and any elements, fill in the VUD length and any data, and fill in the key-block area length and any key blocks.
• Return to the dispatcher.

UDX Command Processors

UDX command processors are coded in the same way as the existing CCA command processors and have all of the same rights and responsibilities. In addition, you must establish the `ccax_cp_list[]` and the `ccax_cp_list_size` variable to inform the dispatcher of the length and content of the sub-function lookup table with the UDX command processor entry points.

CCA Communication Structures

Two of the commonly used data structures internal to the CCA implementation are described in this section:

• Request and reply parameter blocks
• Key blocks and their header

CCA key tokens and access control structures are described in Appendix B of the *IBM 4764 PCI-X Cryptographic Coprocessor CCA Basic Services Reference and Guide*.

Request and Reply Parameter Block Format

The request and reply parameter blocks immediately follow a data structure of type `CPRB_structure`. Figure 1-4 on page 1-11 shows the request and reply parameter block format.

**Note:** Be careful that the host-side code processes the lengths in big-endian format.

**Figure 1-4. Request and Reply Parameter Block Formats**

<table>
<thead>
<tr>
<th>Field</th>
<th>Sub-function code</th>
<th>Rule Array Length</th>
<th>Rule Array Data</th>
<th>Verb Unique Data Length</th>
<th>Verb Unique Data</th>
<th>Key Block Fields Length</th>
<th>Key Block Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>2</td>
<td>2</td>
<td>X</td>
<td>2</td>
<td>Y</td>
<td>2</td>
<td>Z</td>
</tr>
<tr>
<td>Offset</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>4+X</td>
<td>6+X</td>
<td>6+X+Y</td>
<td>8+X+Y</td>
</tr>
</tbody>
</table>

**Field Name**

- **Subfunction code**
  A code that identifies the command processor through a CCA dispatcher table lookup operation.

- **Rule Array Length**
  Length in bytes of the rule array portion of the block. Incorporation of rule-array information is optional, but this field must be present. If no rule-array information is specified, this field must be set to 2 (that is, the size of the length field).
While it is possible to construct a request/reply parameter block “by hand” using pointer arithmetic, it is recommended that the UDX developer instead use the CCA-provided utility routines. **BuildParmBlock** is the routine available on the coprocessor as well as the workstation host. The developer calls **BuildParmBlock** three times to build a request/reply parameter block: once for rule information, once for the verb unique data, and once for the key data. The order is important: rules first, then verb unique data, followed by key data. This routine simplifies request/reply parameter block creation by accepting an arbitrary number of argument pairs (length + data pointer pairs) and constructs the sub-blocks in the previous table.

Similarly, while it is possible to extract data from the request/reply parameter blocks “by hand” using pointer arithmetic, it is recommended that the UDX developer instead use CSFGAPSI (for z/OS) and the CCA-provided utility routines **FindFirstDataBlock**, **FindNextDataBlock**, **find_first_key_block**, and **find_next_key_block**.

**Note:** An example of the use of these functions (**BuildParmBlock** and **CSUC_BULDCPRB**) is in Appendix E, UDX Sample Code - Workstation Host - Test Code on page E-1.

On the z/OS host, the request parameter block can be constructed with the CSFGAPSI callable service. This function takes an array of pointers to verb unique data strings, and an array of pointers to key tokens, and constructs the CPRB and the request block simultaneously.

### Passing Large Data Blocks

If more data must be passed, it is possible to pass the host address to the coprocessor for reading or writing with the **CSUC_BULDCPRB** command on the workstation, or the CSFACPRB callable service on the zSeries server. The buffer so addressed for sending to the coprocessor is referred to as a request data block. The length and pointer for the reply data block can be used for reading data from the coprocessor. The data buffers must not overlap and must be a multiple of eight bytes long.

On the host workstation:

```c
// First, set the CPRB structures properly, with the Rule Array,
// Verb Unique Data, and Key Blocks.
// To set the Request Data Block:
LocalRequestTextLength = htoal(*pTextLength);
```

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule Array Data</td>
<td>Zero or more 8-byte character arrays (not NULL-terminated). If no rule-array elements are specified, this field is empty (0-length).</td>
</tr>
<tr>
<td>Verb Unique Data Length</td>
<td>Length in bytes of the (optional) data that is unique to this verb call and the length field. This field must always be present. If no data is specified, this field must be set to 2.</td>
</tr>
<tr>
<td>Verb Unique Data</td>
<td>Optional data block to be passed to the verb. For instance, if the verb is to encrypt 8 bytes as a key, the verb unique data might be the clear value of the key. If no data is specified, this field is empty (0-length).</td>
</tr>
<tr>
<td>Key Block Fields Length</td>
<td>Length in bytes of the optional key block(s) portion of the request or reply parameter block. This field must always be present. If no keys are specified, this field must be set to 2.</td>
</tr>
<tr>
<td>Key Block Fields</td>
<td>Optional key block(s) exchanged between the host and coprocessor code. If no key tokens are specified, this field should be empty (0-length).</td>
</tr>
</tbody>
</table>
LocalReplyTextLength = htoal(*pTextLength);

CSUC_BULDCPRB( pCrb,
    (UCHAR *) ESSS_FUNCTION_ID_S,
    RequestBlockLength, // Req.Parm block len + adr
    pRequestParmBlock,
    LocalRequestTextLength, // Req.Data block len + adr
    (UCHAR *) pInpText,
    sizeof( pRequestReplyBuffer->reply_buf ),
    pRequestReplyBuffer->reply_buf,
    LocalReplyTextLength, // Reply parm
    (UCHAR *) pOutText);

On the z/OS system:

// Invoke the CSFGAPSI function to set the CPRB structures properly, with the Rule
// Array, Verb Unique Data, and Key Blocks, and then submit the request to the
// PCI-X Cryptographic Coprocessor.

// Then, submit the request to the coprocessor, using the CSFACCPN function.

// The CCP must first be set up to address the request and reply data blocks.

CALL CSFGAPSI( return_code,
    reason_code,
    flags,
    subfunction_code,
    ap_index,
    ap_serial_number,
    ap_domain,
    rule_count,
    rules,
    number_vuds,
    vud_list,
    number_keys,
    key_list,
    reply_vud_block_length,
    reply_vud_block_address,
    reply_key_block_length,
    reply_key_block_address,
    request_data_block_length,
    request_data_block,
    request_data_block_alet,
    reply_data_block_length,
    reply_data_block,
reply_data_block_alet,
SPB);

/* Then, submit the request to the coprocessor, using the CSFACCPN callable */
/* service. The CCP must first be set up to address the request and reply */
/* data blocks. */
INPUT_DATA_BLOCK CHAR(LENGTH(CCNP));
OUTPUT_DATA_BLOCK CHAR(LENGTH(CCNP));

CCNPPTTR = ADDR(INPUT_DATA_BLOCK);
CCNP_ALET = PRIMALET; /* Primary ALET */
CCNP_ADDRESS = REQUEST_DATA_BLOCK; /* Address of data */
CCNP_LENGTH = REQUEST_DATA_BLOCK_LENGTH; /* Length of data */

CCNPPTTR = ADDR(OUTPUT_DATA_BLOCK);
CCNP_ALET = PRIMALET; /* Primary ALET */
CCNP_ADDRESS = REPLY_DATA_BLOCK; /* Address of data */
CCNP_LENGTH = REPLY_DATA_BLOCK_LENGTH; /* Length of data */
CALL CSFACCPN( RETURN_CODE,
REASON_CODE,
REQUEST_CPRB,
REQUEST_CPRB_LENGTH,
REPLY_CPRB
REPLY_CPRB_LENGTH,
PRIMALET,
PCICC_INDEX,
PCICC_SERIAL_NUMBER,
ACCPN_BLANK_ID,
ACCPN_SHARED,
ACCPN_DOMAIN_NOT_APPLIC
SUBFUNCTION_CODE,
INPUT_DATA_BLOCK,
OUTPUT_DATA_BLOCK,
SPB);
On the card:

```c
// Get the length of the bulk text first, from
// the CPRB structure.
BulkBlockLength = pRequestCprb->req_data_block_length;

// Check that the length of the reply data block
// in the CPRB is long enough (depends on your function)
if (BulkBlockLength > pRequestCprb->reply_data_block_length)
{
    Cas_proc_retc( pReplyCprb, E_SIZE );
    return;
}

OutTxt = malloc(BulkBlockLength);
if (OutTxt == NULL)
{
    Cas_proc_retc(pReplyCprb, E_ALLOCATE_MEM);
    return;
}

// Process the text:
ReturnMsg = my_function(pRequestCprb->req_data_block_addr, OutTxt, BulkBlockLength);
if (ReturnMsg != S_OK)
{
    free(OutTxt);
    Cas_proc_retc(pReplyCprb, ReturnMsg);
    return;
}

// Write the Length of OutTxt in the CPRB
pReplyCprb->replied_data_block_length = BulkBlockLength;

// Write the address of OutTxt in the CPRB
pReplyCprb->reply_data_block_addr = OutTxt;

return;
```

**Key Blocks**

The key blocks portion of the request and reply parameter blocks is used to transport zero or more key identifiers: key labels and/or key tokens. Note that on the z/OS system, only key tokens may be transferred to the coprocessor. The CSFACKDS and CSFAPKDS services must be used to recover the token from storage before constructing the key block. A key block is a data structure consisting of a header and appended key label and/or key token data.

The key block header is a data structure containing a `USHORT` Length field in big-endian format followed by a `USHORT` Flags field in big-endian format. The Length field indicates the length of the header plus...
the length of the key token or label which follows it, while the Flags field informs SECY what functions are required of it. The Flags field options are detailed in the following table:

<table>
<thead>
<tr>
<th>Flags indicating the type of key (one required)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PKA96_TYPE</td>
<td>Indicates that the key token or label is an PKA token.</td>
</tr>
<tr>
<td>DES96_TYPE</td>
<td>Indicates that the key token or label is a DES key token.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flags indicating the action to be taken (one required)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION_READ</td>
<td>Requests SECY to retrieve the key token from the storage file. This header must be followed by a key label.</td>
</tr>
<tr>
<td>ACTION_WRITE</td>
<td>Requests SECY to add or overwrite the key token in the storage file. This header must be followed by a key label concatenated with a key token.</td>
</tr>
<tr>
<td>ACTION_NOOP</td>
<td>Requests no action from SECY.</td>
</tr>
</tbody>
</table>

For z/OS, the keys must be passed as tokens; therefore, the Flags field should be set to binary zeros.
Chapter 2. Building a CCA User Defined Extension

This chapter describes the process you can follow in creating a User Defined Extension (UDX) for the CCA application that performs within and accesses the coprocessor. The chapter begins with an explanation of the files that you will use and then continues with the steps that you can follow in developing the host and the coprocessor pieces of code. It is assumed that you are familiar with developing and testing applications for the coprocessor, that you have knowledge of the xCrypto API as explained in the other toolkit publications (see “Related Publications” on page ix), and that you are familiar with the z/OS application environment (as described in the z/OS ICSF Application Programmer’s Guide and the z/OS ICSF System Programmer’s Guide).

When a CCA UDX is built for zSeries, it is initially tested on a workstation. The steps taken include building the workstation piece of a UDX, testing, and finally, building the z/OS piece of a UDX; thus, the arrangement of this chapter includes instructions in that order.

Files You Use in Building a UDX
A developer will need the following files, which are not provided with the toolkit, to produce a UDX.

- **mkfs.jffs2** is a Linux filesystem builder which will build a filesystem to be used on the flash memory of the coprocessor.
- A cross-compiler for Linux on the PowerPC chip.
- A compiler of your choice to build the code which will run on the host machine.

A developer will need the following files to produce a UDX:

- A header file (for example, **csulextn.h**) that defines the interface the UDX exports to a user’s application. This header file is #included by the user’s application and by the workstation host piece of the UDX and should contain a function prototype for each service the UDX provides. Such services are implemented in the same manner as CCA verbs; an example appears in Figure 2-1 on page 2-4. This file is needed for testing the coprocessor code prior to installing the code on a zSeries server.

- A header file (for example, **cxt_cmds.h** or **cxt_cmds_zudx.h**) that defines the interface between the workstation host piece of the UDX and the coprocessor piece of the UDX. This header file is #included by the workstation host piece and the coprocessor piece and should define UDX subfunction codes and the access control points and completion codes used by the UDX. The sample provided with the UDX Development Toolkit includes comments that indicate the range of acceptable values for each of these elements.

- One or more C source files (for example, **zos/host/test/sxt_samp.c** or **wks/host/tst_udx.c**) that implement the workstation host piece of the UDX. Two sample files are provided with the UDX Development Toolkit: **sxt_samp.c** is a skeleton that exports a single function to the user's application, for testing a zSeries UDX on a workstation and **tst_udx.c** exports 3 functions to the user's application. Each function checks its input parameters, constructs a request block, sends the request to the coprocessor and receives the reply, extracts the result, and returns it to the user’s application.
Building a CCA User Defined Extension page 2-2

- One or more C source files (for example, card/zos/cxt_cmds.c or card/wks/card.c) that implement the coprocessor piece of the UDX. Two sample files are provided with the UDX Development Toolkit: cxt_cmds.c is a skeleton that implements a single command processor and card.c implements 3 command processors. Each command processor receives a request from the host, validates the request and extracts the arguments it contains, performs a simple operation, constructs a reply block, and returns the reply to the host piece of the UDX. Each command processor must be added to the ccax_cp_list array, which must be defined in this or another program file which is compiled with the coprocessor piece of the UDX.

- If you are building a UDX for the zSeries platform, two or more Assembler language source files that implement the zSeries host piece of the UDX. The samples provided with the UDX Development Toolkit are a callable service stub (zudxstub.bal), a UDX callable service (zudxsvc.bal), and a CSFPCI post-processing exit (zudxexit.bal). The UDX Development Toolkit also provides some macros that you may use in creating your UDX callable service. These macros can be uploaded to a zSeries host system where you perform your development. (For further information about the control blocks defined by these macros, refer to z/OS Integrated Cryptographic Service Facility System Programmer’s Guide.)

A developer may need to modify the following files that are provided with the UDX Development Toolkit to produce a UDX:

- A makefile that builds the shared object or dynamic load library that implements the workstation host piece of the UDX. The UDX Development Toolkit provides csufextn.mak (which creates libcsufextn.so for use on AIX), csulextn.mak (which creates libcsulextn.so for use on SUSE Linux Enterprise Server 9) and csunextn.mak (which creates csunextn.dll for use on Windows).

- A makefile (for example, zos/card/cardzos.mak or wks/card/cardwks.mak) that builds the executable that implements the coprocessor piece of the UDX. This executable includes the object for the UDX itself as well as a library that implements the coprocessor CCA object modules. This makefile uses the gcc cross-compiler to build code which will run on the PowerPC coprocessor. This makefile assumes that the environment variables XCTK_FS_ROOT and UDXTK_FS_ROOT point to the appropriate base of the toolkit directory structure (for example, /opt/xctk, /opt/udxtk, respectively). It also assumes that the environment variable CROSS_COMPILER points to the root of the tree where the cross-compiler for ppc is stored on your system (for example, /opt/cross-compiler/powerpc-linux).

- A shell script (for example, shells/debugwks.shell or shells/prodzos.shell) that will run on the coprocessor to start the required processes. This file must be renamed init.sh and copied into the /user0 directory of the filesystem which is ported to the coprocessor. The toolkit provides debug shells to set up a daemon on the card to run the ICAT debugger, and production shells to set up a production version of the UDX, without the debugger. The z/OS production version of the shell also installs the Outbound Authentication Manager, which is required for use on a z/OS system.

- A makefile (for example, build/Makefile) that incorporates the required binaries and shell files into a file structure and formats them for incorporation onto the read-only disk using the mkfs.jffs2 call. The provided makefile requires that the user have sudo chown permissions.
The following binary files are used to produce a UDX:

- A library that contains definitions of the interface the UDX exports to a user’s application on a workstation. This library is linked with the user’s application. The UDX Development Toolkit makefile generates the appropriate library: `csulextn.mak` creates `libcsulextn.so`, while `csunextn.mak` creates `csunextn.dll`. If you are creating a zSeries UDX, you will require one of these libraries only for testing on the workstation; otherwise, one of these files will be needed on your UDX.

- A library that contains the coprocessor CCA object modules. This library is linked with the object files that constitute the coprocessor piece of the UDX. The result is a coprocessor application executable that contains all of the standard CCA functions and those functions provided by the UDX. The UDX Development Toolkit provides two libraries: `libcsullib.so` for use when building a UDX to run on a workstation (either Windows or Linux), and `libcsuzlib.so` for use when building a UDX to run on a zSeries server.

  **Important Note:** It is possible to make and test a z/OS UDX without using `libcsullib.so`. However, due to restrictions on the user inherent in the `libcsuzlib.so` library, a complex application may be easier to debug if it is first compiled with the `libcsullib.so` library to test the access control restrictions, then rebuilt with the `libcsuzlib.so` library and TEST_LNX defined to prevent access testing for further testing on the workstation, and finally rebuilt with the `libcsuzlib.so` library and TEST_LNX undefined for installation on a zSeries system.

- A program named `startcca`, which changes the effective user ID and group ID of the process running on the coprocessor. This assures that the process does not run as root, but can still access required files (such as SRDIs). The toolkit provides this file in the `bin` directory.

- A program named `startoa`, which sets up a thread to run the Outbound Authentication Manager on the coprocessor. The toolkit provides this file in the `bin` directory.

- A program named `xcmgrd`, which manages the coprocessor application and tests for tamper conditions. The toolkit provides this file in the `bin` directory of the UDX toolkit.

- Optionally, the `pdaemon` program, which is the coprocessor side of the ICATPCX debugger. The toolkit provides this file in the `$(ICAT_ROOT)/card` directory.

A UDX developer defines certain constants (for example, subfunction codes, access control points, and completion codes) during development. There is no guarantee that the values the developer chooses for these constants do not collide with the values the developer of another UDX has chosen. This is generally not a problem since all UDXs used by a particular customer are developed by a single organization and procedures to avoid collisions are adopted.

In order to avoid collisions between UDX constants and constants used by future versions of CCA, the following have been reserved for use by developers writing UDXs:

<table>
<thead>
<tr>
<th>IBM Developers Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subfunction Codes</td>
</tr>
<tr>
<td>“WA” - “WZ”, “W0” - “W9”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other Developers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subfunction Codes</td>
</tr>
<tr>
<td>“XA” - “XZ”, “X0” - “X9”, “YA” - “YZ”, and “Y0” - “Y9”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>All Developers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completion Codes</td>
</tr>
<tr>
<td>0x5000 - 0xFFFF</td>
</tr>
</tbody>
</table>

Building a CCA User Defined Extension  page 2-3
Workstation Piece of a UDX

The host piece of a UDX is a shared object or dynamic link library that converts requests for service from the user’s application into one or more calls to the standard CCA host API module (libcsulsapi.so or csunsapi.dll). The host piece of a UDX typically checks its input parameters, constructs a request block, sends the request to the coprocessor and receives the reply, extracts the result, and returns the result to the user’s application.

This section lists the steps a developer must complete in order to create the host piece of a UDX which will be run on a workstation. This is required to test a UDX which is to be installed on a zSeries server, or to run a UDX on a workstation.

1. Define the UDX API.

A prototype for each function the UDX exports to the user’s application must be placed in a header file (for example, csulextx.h) that is #included by the user’s application and by the host piece of the UDX. The header file should also contain any ancillary declarations (for example, constant values) the exported interface requires.

Prototypes for the standard CCA API functions may serve as examples and are located in csucincl.h. Both header files are part of the UDX Development Toolkit. Figure 2-1 illustrates a representative prototype.

```c
extern void SECURITYAPI
CCAXFCN1(long * return_code,
    long * reason_code,
    long * exit_data_length,
    unsigned char * exit_data,
    long * rule_array_count,
    unsigned char * rule_array,
    unsigned long * key_id_length,
    unsigned char * key_identifier
);
```

Figure 2-1. Example CCA/UDX Verb

UDX prototypes may have any number of parameters, although for consistency reasons it is recommended that all UDX functions include the first six parameters that appear in Figure 2-1 (return_code, reason_code, exit_data_length, exit_data, rule_array_count, and rule_array). Every parameter must either be a pointer to a 32-bit integer, a pointer to an array of bytes, or a pointer to an array of integers. In the case of arrays, the number of elements in the array is by convention passed in a separate parameter.

The prototype in Figure 2-1 defines a function named CCAXFCN1. The function takes these parameters: the array of bytes pointed to by key_identifier and the integer pointed to by key_id_length which contains the number of bytes in the array, in addition to the standard 6 parameters.

Refer to the IBM 4764 PCI-X Cryptographic Coprocessor CCA Basic Services Reference and Guide for more information about parameter types.
2. **Define the subfunction codes for the UDX.**

The coprocessor piece of a UDX consists of one command processor. The host piece of the UDX uses a "subfunction code" to identify the command processor to which it wants to send a particular request. The values of subfunction codes must be defined in a header file (for example, `cxt_cmds.h`) that is #included by both pieces of the UDX. Figure 2-2 on page 2-5 contains an example of such a definition.

A list of the subfunction codes for the standard CCA API functions appears in `cmncryt2.h`, which is part of the *UDX Development Toolkit*.

```c
/* 'XA' - Sample CCA
 ********************************************
** ENTER your CCA command extension sub-function
** codes after this comment.
** --------------------------------------------------
** The xxxx_ID entry is for unsigned short operations
** on the 2 byte sub-function code.
** The xxxx_ID_S entry is for character string
** operations on the 2 byte sub-function code.
** This is the actual order the character
** code appears in the field.
** The following 2 character code points have been
** reserved for CCA extensions. If you use other code
** points they may conflict with existing CCA commands.
** WA - WZ, WS - W9 (for IGS developers)
** XA - XZ, XS - X9
** YA - YZ, YS - Y9
*********************************************/
#define CCAXFNC1_ID 0x5841
#define CCAXFNC1_ID_S "XA"
```

*Figure 2-2. Example UDX Subfunction Codes*
3. **Define new completion codes for the UDX.**

   A UDX function returns a completion code indicating whether the function succeeded or not (and giving some idea of what caused the failure if one occurred). The standard CCA completion codes are defined in `cmnerrcd.h` and their meanings and use are further clarified in an appendix to the *IBM 4764 PCI-X Cryptographic Coprocessor CCA Basic Services Reference and Guide*. If no standard code is applicable to a particular situation, new completion codes must be defined in a header file (for example, `cxt_cmds.h`) that is included in both pieces of the UDX. Figure 2-3 on page 2-6 contains an example of such a definition.

```c
/**********************************************************
** Enter your CCA extension completion codes after
** this comment.
** ==================================================
**
** The definition of a completion code (0x00yyzzzz) consists of
** 2 parts. Where:
**
** yy is the return code ( 00, 04, 08, 0C, 10 ).
** zzzz is the reason code.
** The following range of 2 byte hex reason codes
** have been reserved for CCA extensions.
**
** 0x5000 - 0x5FFF
*******************************/
/* 00/20480 sample  */
#define CXT_INFO_xxxxxx 0x00005000L
/* 04/20481 sample */
#define CXT_WARN_xxxxxx 0x00045001L
/* 08/20482 sample */
#define CXT_ERR8_xxxxxx 0x00085002L
/* 12/20483 sample */
#define CXT_ERR12_xxxxxx 0x000C5003L
/* 16/20484 sample */
#define CXT_ERR16_xxxxxx 0x00105004L
```

**Figure 2-3. Example UDX Completion Codes**

4. **Design and code the logic of the host piece of the UDX.**

   The host piece of a UDX is typically straightforward - it essentially constructs a request block, sends the block to the coprocessor, and parses the result. See Appendix E. “UDX Sample Code - Workstation Host - Test Code” on page E-1 for a sample (`sxt_samp.c`).

   In general, the host piece of a UDX should be as small as possible. Most of the work should be performed by the coprocessor piece. This approach makes it much easier to port the test code to/from the zSeries server.

5. **Build the UDX shared object file or host library.**

   The *UDX Development Toolkit* includes sample makefiles for gcc on SLES 9 (`csulextn.mak` and `hostlnx.mak`) as well as sample makefiles for MSVC on Windows (`csunextn.mak` and `host_msvc.mak`). Statements should be added to compile the source files that contain the host piece of the UDX and create the UDX shared object or library file. The user's application links with the library file to resolve references to the functions the UDX exports.
For further information about the build environment, including required compiler options, refer to chapter 3 of the IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide.

6. **Install the code on a workstation host for testing.**

When running a UDX application on a Linux workstation, the following shared objects must be available (in your LD_LIBRARY_PATH path): `libcsulsapi.so, libcsulsecy.so, libcsulcall.so, libds30.so,` and `libxcryp.so`. These files are provided with CCA, and are typically installed in the `/usr/lib` directory.

When running a UDX application on a Windows workstation, the following libraries must be available (in your path): `csunsapi.dll`, `csunsecy.dll`, `csuncall.dll`, `csundir.dll` which are provided by the installation of CCA.

When testing a zSeries UDX on a Linux workstation the following shared objects must be available (in your LD_LIBRARY_PATH path): `libcsuzsapi.so, libcsuzsecy.so, libcsuzcall.so,` and `libds30z.so`. These files are provided by the UDX Toolkit, in your `/udxtk/###/lib/zos` directory.

When testing a zSeries UDX on a Windows workstation the following libraries must be available (in your path): `csuzsapi.dll`, `csuzsecy.dll`, `csuzcall.dll`, and `csuzdir.dll`. These files are provided by the UDX Toolkit, in your `/udxtk/###/lib/zos` directory.

**Coprocessor Piece of a UDX**

The coprocessor piece of a UDX is a collection of one or more command processors that is linked with IBM’s CCA coprocessor application modules (either `libcsuzlib.so` or `libcsullib.so`) to create an executable that can be loaded into the coprocessor. The coprocessor piece of a UDX may invoke any of the CCA services and can also invoke Linux functions.

This section lists the steps a developer must complete in order to create the coprocessor piece of a UDX.

1. **Define the UDX command processor API.**

   A prototype for each command processor the coprocessor piece of the UDX makes available to the host piece of the UDX must be placed in a header file (for example, `cxt_cmds.h`) that is #included by the coprocessor piece of the UDX. The prototype must have the same parameters and return type as the example shown in Figure 2-4.

```c
/***************************************************************************
** Enter your CCA command extension function prototypes
** after this comment.
** =======================================================================
** The entry points must have the following parameter
** definitions.
**
** *pCprbIn - (input) Pointer to the input CPRB. The request
** parameter block exists immediately after the
** CPRB area.
** *pCprbOut - (output) Pointer to an area for returning of the
** CPRB followed by the reply parameter block.
** RequestId - (input) Adapter request identifier. Required
** as input to some xCrypto calls.
** roleID - (input) The user's role identifier. Required
** as input when checking the requestor's access
** authority to this function.
```
Figure 2-4. Example UDX Command Processor Prototype

On entry to a command processor:

- \texttt{pCprbIn} contains the address of a cooperative processing request block (CPRB). The CPRB's contents match the contents of the CPRB pointed to by the \texttt{pCprb} argument the host piece of the UDX passed to the call that caused the command processor to gain control.

- \texttt{pCprbOut} contains the address of a buffer large enough to hold a CPRB header and the result of the operation.

- \texttt{RequestId} contains a handle generated by the coprocessor operating system that uniquely identifies the message that the host sent to the coprocessor whose receipt caused the command processor to gain control. A command processor that invokes basic coprocessor operating system functions may need to pass this handle as an argument to those functions.

- \texttt{roleID} contains the identifier of the role associated with the host process that caused the command processor to gain control. It can be used to verify that the host process has the proper authority to perform the requested function.

2. Define access control points for the UDX.

Associated with each profile on the host is a role, or set of coprocessor operations the profile is allowed to invoke. If access to the functions exported by the coprocessor piece of the UDX needs to be restricted in any way, new “access control point” values must be defined in a header file (for example, \texttt{cxt_cmds.h}) that is \#included by the coprocessor piece of the UDX. Figure 2-5 on page 2-9 contains an example of such a definition.

A command processor can use access control points in conjunction with the role identifier supplied as an argument to the command processor to determine whether or not a particular operation is authorized. See “check\_access\_auth\_fcn - Verify User Authority” on page 12-3 for details.

Note: For zSeries, the access control point for a UDX should never be enabled in the DEFALTxx role. Enablement of a UDX access control point requires a TKE. The access control point will be disabled until a security administrator enables it by way of the TKE. (See item 6 on page 2-12 - a CSFPCI post-processing exit is also required.) Since a UDX access control point can only be enabled by way of the TKE, while testing the coprocessor piece of the UDX on a workstation the code that checks the enablement status of the access control point should be temporarily disabled during testing. Search for “TEST\_LNX” in the \texttt{zudx.c} file provided.

---

1. \texttt{RequestId} is the value returned in the \texttt{pRequestHeader->RequestId} output from the call to \texttt{xcGetNextHeader} that received the message. Refer to the \textit{IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Interface Reference} for details.
If you are testing the access control points on a workstation, you must add these values to the 
\texttt{csuap.def} file in the \texttt{cnm} subdirectory of the CCA (for example, \texttt{cca/cnm}). The \texttt{cnm} utility uses this 
file to enable editing of roles. Refer to the \textit{IBM 4764 PCI-X Cryptographic Coprocessor CCA Support} 
\textit{Program Installation Manual} for more information.

\begin{verbatim}
/****************************************************************************
** Enter your CCA command extension access control points after this comment.
**============================================================================
** The following range of 2 byte hex code points have been reserved
** for CCA extension access control points.
**
** 0x8000 - 0xFFFF
****************************************************************************
#define CXT_COMMAND_XXXXXXX 0x8000 /* Sample definition. */
\end{verbatim}

\textbf{Figure 2-5. Example UDX Access Control Points}

3. \textbf{Add the UDX command processors to the command decoding array.}

IBM's CCA coprocessor application modules uses an array to determine which UDX command 
processor to invoke when a request with a particular subfunction code is received. An entry for each 
command processor must be added to the \texttt{ccax_cp_list} array, which must be defined in a program 
file (for example, \texttt{cxt_cmds.c}) that is compiled with the coprocessor piece of the UDX. Each entry 
contains a subfunction code and the name of the corresponding command processor.

The \texttt{ccax_cp_list_size} variable must be initialized to the number of entries in the array.

\begin{verbatim}
/**************************************************************
** Enter your CCA command extension array entry after this comment.
**====================================================================
** Each element of the table is a CCAX_CP_DEF type. That is,
** it contains one 2 character sub-function code, and a
** pointer to the corresponding command processor function.
**
*****************************************************************************/
CCAX_CP_DEF ccax_cp_list[] = { { CCAXFNC1_ID, ccax_fcn_1 },
                            { CCAXFNC2_ID, ccax_fcn_2 } };
/***************************************************************/
ULONG ccax_cp_list_size = (sizeof(ccax_cp_list)/sizeof(CCAX_CP_DEF));
\end{verbatim}

\textbf{Figure 2-6. Example UDX Command Decoding Array Definition}

4. \textbf{Design and code the logic of the coprocessor piece of the UDX.}

The coprocessor piece of a UDX has access to the same internal functions and services as the CCA 
coprocessor application modules and may be quite complex. A sample (\texttt{zudx.c}) appears in Appendix 
5. Build the UDX coprocessor executable.

The UDX Development Toolkit includes a sample makefile (cardinx.mak for workstation, or cardzos.mak for z/OS) that works with the cross-compiler on SLES 9. Statements should be added to compile the source files that contain the coprocessor piece of the UDX. The executable file must be named csulcca to work with the startcca program and the init.sh file on the coprocessor.

6. Install the code on a workstation host for testing.

When running a UDX application on a AIX workstation, the following shared objects must be available (in your LIBPATH path): libcufsapi.a, libcufsecy.a, libcufcall.a, libcufdir.a, and in libxcryp.a. These files are provided by the installation of CCA.

When running a UDX application on a Linux workstation, the following shared objects must be available (in your LIBPATH path): libcufsapi.a, libcufsecy.a, libcufcall.a, libcufdir.a, and in libxcryp.a. These files are provided by the installation of CCA.

When running a UDX application on a Windows workstation, the following shared objects must be available (in your path): csunsapi.dll from the UDX Development Toolkit, and csunsecy.dll, csuncall.dll, csundir.dll which are provided by the installation of CCA.

7. Modify the initialization shell script.

When the coprocessor is reset, the boot sequence includes the running of a segment 2 shell file which calls user0/init.sh as its final command. This file must be set up to start the startcca, startoa, and xcmgrd processes on the coprocessor, as well as (optionally), the daemons to run the debugger (pdaemon) and the logger (cryptologkd). The UDX Development Toolkit includes 4 shell files: debugwks.shell, debugzos.shell, prodwks.shell, and prodzos.shell. The “debug” shell files include instructions to start the debugger, pdaemon, and the Outbound Authentication Manager, xcoad, while the “zos” files include instructions to start the Outbound Authentication Manager.

8. Build the UDX image.

The UDX Development Toolkit includes a sample makefile (Makefile) in the build directory which will place the executables required into the required format within the user0 directory. You may change compiler variables within this makefile to obtain a debug image or a production image. The makefile builds udxlinx3.bin or udxzos3.bin.

9. Load the image onto the coprocessor.

The UDX Development Toolkit includes druid to install the segment3.bin image onto the coprocessor. Refer to the IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Developer's Toolkit Guide for an explanation of this DRUID utility.

Note: To securely load your application into a coprocessor requires that the application be signed with keys certified by Development in IBM Charlotte. Refer to the IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Developer's Toolkit Guide for an explanation of the process to obtain certified keys and to sign your application.

zSeries Piece of a UDX

The zSeries host piece of a UDX consists of two (or more) modules which implement a callable service that performs one or more cryptographic functions. An application program calls and passes parameters to the callable service. The main portion of the host piece of the UDX is the module which provides the callable service. The UDX callable service module typically checks its input parameters, constructs a request block, sends the request to the coprocessor and receives the reply, extracts the result, and returns the result to the user’s application. The second module required to implement the host portion of the UDX is a service stub. The service stub connects the application program with the UDX callable service.
UDX callable service is defined in the ICSF Installation Options Data Set via the UDX keyword. Using the UDX keyword, a number to identify the service and the load module containing the service are specified.

During ICSF startup, ICSF loads the load module containing the UDX service into the ICSF address space with the ICSF callable services. ICSF binds the service with the service number specified in the Installation Options Data Set.

This section lists the steps a developer performs in order to create the host piece of the zSeries UDX.

1. Define the UDX API.
   Each UDX verb may have any number of parameters. Callable service parameters are positional, and must be specified even if not used. For consistency, it is recommended that all UDX verbs include the following six parameters as the first six parameters of the API: `return_code`, `reason_code`, `exit_data_length`, `exit_data`, `rule_array_count`, and `rule_array`. It is recommended that the parameters in this API be equivalent to the parameters as previously defined in the workstation host code.

2. Define the subfunction code for each UDX verb.
   There will be one subfunction code associated with each UDX verb command processor (the coprocessor piece of the UDX). The following 2-character code points have been reserved for CCA extensions. You should not use other code points as they may conflict with existing CCA commands. This subfunction code must be the same as that defined in the coprocessor section as previously discussed. The subfunction code that you define here must be the same as the subfunction code added to the `ccax_cp_list` array for the coprocessor piece of the UDX (see item 3 on page 2-9).
   - `XA - XZ, X0 - X9` (reserved for customer-written UDXs)
   - `YA - YZ, Y0 - Y9` (reserved for customer-written UDXs)

   The UDX subfunction code should be declared as a constant in the host callable service module, for example:

   ```
   X'4B47 EQU CCP_FUNCTION_KEY_GENERATE
   ```

   (that is, subfunction code of “KG” in ASCII, in “big-endian” format).

3. Design and code the logic of the UDX callable service.
   The host piece of a UDX is typically straightforward - it essentially constructs a request block, sends the block to the coprocessor, and parses the result. See Appendix A, “UDX Sample Code - Host Piece - Service” on page A-1 for a sample (ZUDXSVC.BAL). This sample can be used as a skeleton and customized to meet the requirements of most UDXs.

   In general, the host piece of a UDX should be as small as possible. Most of the work should be performed by the coprocessor piece.

4. Code the UDX service stub.
   Besides writing the callable service itself, you must write a service stub, which is the connection between the application program and the UDX callable service. The application program calls the service stub, which accesses the callable service. The service stub can be identified by any name you choose to call it. All callable service stubs for callable services which execute in ICSF’s address space look identical. See Appendix B, “UDX Sample Code - Host Piece - Service Stub” on page B-1 for a sample stub which may be copied to create a new UDX service stub.
5. **Assemble the UDX callable service and service stub.**
   Use the macros provided with the *UDX Development Toolkit* and the High-Level Assembler to assemble the UDX callable service and service stub. (For further information about the control blocks defined by these macros, refer to the *z/OS Integrated Cryptographic Service Facility System Programmer’s Guide.*)

6. **Code a post-processing exit for the CSFPCI service, if required.**
   If your UDX command processor has an associated access control point, enablement of the access control point is accomplished by way of the TKE.
   
   Refer to the *z/OS ICSF Trusted Key Entry Workstation User’s Guide* for information about the procedure for enabling/disabling access control points using the TKE Version 4.0. To construct the TKE panel which shows the Security Administrator the access control points which may be enabled/disabled, code running in the TKE queries ICSF for information about access control points. The TKE code invokes the ICSF CSFPCI (PCI interface) callable service, specifying the “ACPOINTS” keyword. If you have coded your UDX command processor to require an access control point to be enabled, you must code a post-processing exit for the CSFPCI callable service to add information about your UDX access control point to the data returned to the TKE by the service.

   *(The *z/OS ICSF System Programmer’s Guide* contains information about the callable service pre-processing and post-processing exits.) Appendix C, “UDX Sample Code - z/OS Host Piece - CSFPCI Post-Processing Exit” contains a sample post-processing exit. This sample can be used as a skeleton and customized to meet the requirements of most UDXs.*

**Installing the zSeries Host Piece of a UDX**

UDX files must be installed on the zSeries host and the ICSF Installation Options Data Set must be customized in order to use your UDX. The files and the steps to be followed are specified below. (The *z/OS ICSF System Programmer’s Guide* may also provide valuable information about steps required for UDX installation.)

1. **Add the UDX callable service to an APF authorized library.**
   The OBJ file for the UDX callable service must be link-edited into a load module and installed into an APF authorized library. ICSF uses the normal z/OS search order to locate the service:

   - Job pack area
   - Steplib (if one exists)
   - Link pack area (LPA)
   - Link list (SYS1.LINKLIB concatenation)

2. **Link-edit the service stub with the application program.**
   The OBJ file for the service stub must be link-edited with the application program which calls the service stub. Any application program that calls a service stub must be link-edited with the service stub.

   To call a UDX service from an application program, use the following statement:

   ```
   CALL <service_stub_name><service_parameters>
   ```

   where **service_stub_name** is the name of the service stub for the UDX callable service and **service_parameters** are the parameters you want to pass to the UDX callable service. You supply the parameters according to the syntax of the programming language that you use to write the application program.
3. **Add the UDX to the ICSF Installation Options Data Set.**
   You must identify the UDX service in the ICSF Installation Options Data Set using the UDX keyword. For information about the specification of the UDX keyword, refer to the *z/OS ICSF System Programmer’s Guide*. You will specify information including the UDX subfunction code, a service number, and a load module name.

4. **Install the CSFPCI Post-Processing Exit (if needed).**
   If you coded a post-processing exit for the CSFPCI callable service, the exit must be installed. Link-edit the OBJ file into a load module, and install the load module into an APF-authorized library. ICSF uses the normal z/OS search order to locate the service:
   - Job pack area
   - Steplib (if one exists)
   - Link pack area (LPA)
   - Link list (SYS1.LINKLIB concatenation)
   The ICSF Installation Options Data Set must be updated to define the exit. Use the EXIT keyword, specifying “CSFPCI” for the ICSF name of the callable service exit. For information about the specification of the EXIT keyword, refer to the *z/OS ICSF System Programmer’s Guide*.

5. **(Optional) Add the UDX verb to the CICS Wait List.**
   If your UDX will be invoked by a CICS transaction, you may wish to make use of the CICS-ICSF Attachment Facility and ICSF’s CICS Wait List. The purpose of the CICS-ICSF Attachment Facility is to enhance the performance of CICS transactions in the same region as a transaction using long-running ICSF services. Without the CICS-ICSF Attachment Facility, the application that requests a long-running ICSF service is placed into an OS WAIT. This affects any other transactions that run in the same region. The CICS-ICSF Attachment Facility consists, in part, of a CICS Task-Related User Exit (TRUE). The TRUE attaches a task control block (TCB) which performs the actual call to the ICSF service. This allows the CICS application that requests the long-running service to be placed into a CICS WAIT, rather than an OS WAIT, for the duration of the operation. For information on how to install the CICS-ICSF Attachment Facility and how to add your UDX to ICSF’s CICS Wait List, refer to the *z/OS ICSF System Programmer’s Guide*. 
Chapter 3. xCrypto Functions

The CCA API is built on top of the secure cryptographic coprocessor (xCrypto) API, a lower-level API that allows the host piece of IBM’s CCA Support Program to interact with the coprocessor piece of the CCA Support Program and allows the coprocessor piece of the CCA Support Program to perform various cryptographic operations and to manipulate persistent storage on the coprocessor. xCrypto API functions can also be invoked by a UDX. The xCrypto 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 coprocessor may invoke (the coprocessor-side API).

This section briefly describes the xCrypto API. A more detailed description may be found in the *IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Interface Reference*.

Host-Side xCrypto API Functions

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

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

*Note*: Host-side xCrypto API functions are not available on a z/OS system. When running on a z/OS system, the card must be accessed through the ICSF CCA CCPN functions.

Coprocessor-Side xCrypto API Functions

The coprocessor API includes functions in the following categories:

<table>
<thead>
<tr>
<th>Functions Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communications</td>
<td>Allows a coprocessor application to interact with a host application and obtain permission to request services from the coprocessor device managers.</td>
</tr>
<tr>
<td>Hash</td>
<td>Allows a coprocessor application to compute a condensed representation of a block of data using various standard hash algorithms.</td>
</tr>
<tr>
<td>DES</td>
<td>Allows a coprocessor application to request services from the Secure Key Cryptographic Hash (SKCH) Manager, which uses the coprocessor’s SKCH chip to support DES operations with key lengths of 56, 112, or 168 bits.</td>
</tr>
<tr>
<td>Public Key Algorithm</td>
<td>Allows a coprocessor application to request services from the Public Key Algorithm (PKA) Manager, which uses the coprocessor’s large-integer modular math hardware to support public key cryptographic algorithms.</td>
</tr>
</tbody>
</table>
On the coprocessor, each piece of hardware is a device, which is accessed using a file descriptor. To determine the correct file descriptor when running a CCA UDX, use the function `CasGetFileDescriptor`, which is described in Chapter 12, Miscellaneous Functions.
Chapter 4. Communications Functions

In CCA, the host and coprocessor communicate by exchanging well-formed request and reply data blocks. For consistency, UDX routines also follow this paradigm.

This section describes functions needed to allow the host and coprocessor to exchange requests and replies.

Header Files for Communications Functions

When using these functions, your program must include the following header files.

```c
#include "cmncryt2.h"  /* Cryptographic definitions */
#include "cmnfunct.h"  /* Common library routines */
#include "cassub.h"    /* for Cas_proc_retc */
```

When using the functions that are available on the z/OS host, your program should include the CSFGSVT macro which contains the ENTRY statements for the host communications functions.

```c
%INCLUDE SYSLIB(CSFGSVT);  /*Generic Service Vector Table */
```

Summary of Functions

Functions available on the z/OS host:

- **CSFACKDS** Access the in-storage ICSF Cryptographic Keys Data Set.
- **CSFAPKDS** Access the ICSF Public Key Data Set.
- **CSFGCCPN** Send a request to the coprocessor.
- **CSFGCPRB** Build a CPRB.
- **CSFADSCP** Destroy a CPRB.
- **CSFGVLPB** Validate a CPRB.
- **CSFAPBLK** Parse a CPRB.
- **CSFAPKTV** Validate/initialize an RSA/DSS key token.
- **CSFGAPSI** Communication interface between services and the coprocessor.
- **CSFASEC** Check authorization to a RACF-protected or security-exit-protected resource.
Functions available on the coprocessor or workstation:

**Note:** Functions marked with a C are available on the coprocessor, those marked with a W are available on the workstation host.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>BuildParmBlock</td>
<td>Build a parameter block.</td>
<td>C, W</td>
</tr>
<tr>
<td>Cas_proc_retc</td>
<td>Prioritizes a return code in the reply CPRB.</td>
<td>C</td>
</tr>
<tr>
<td>CSNC_SP_SCSRFBSS</td>
<td>Send a request to the coprocessor.</td>
<td>W</td>
</tr>
<tr>
<td>CSUC_BULDCPRB</td>
<td>Construct a well-formed CPRB block.</td>
<td>W</td>
</tr>
<tr>
<td>CSUC_PROCRETC</td>
<td>Prioritize a return code.</td>
<td>W</td>
</tr>
<tr>
<td>FindFirstDataBlock</td>
<td>Search for the first data block.</td>
<td>C, W</td>
</tr>
<tr>
<td>FindNextDataBlock</td>
<td>Search for the next data block.</td>
<td>C, W</td>
</tr>
<tr>
<td>find_first_key_block</td>
<td>Search for the first key block.</td>
<td>C, W</td>
</tr>
<tr>
<td>find_next_key_block</td>
<td>Search for the next key block.</td>
<td>C, W</td>
</tr>
<tr>
<td>InitCprbParmPointers</td>
<td>Initialize CPRB parameter pointers.</td>
<td>C</td>
</tr>
<tr>
<td>keyword_in_rule_array</td>
<td>Search for a keyword in the rule array.</td>
<td>C, W</td>
</tr>
<tr>
<td>parm_block_valid</td>
<td>Examine and verify a parameter block.</td>
<td>C, W</td>
</tr>
<tr>
<td>rule_check</td>
<td>Verify a rule array.</td>
<td>C, W</td>
</tr>
<tr>
<td>saf_process_key_label</td>
<td>Process a key label.</td>
<td>W</td>
</tr>
</tbody>
</table>
**CSFACKDS - Access ICSF Cryptographic Keys Data Set**

**Note:** This function is available on the z/OS host.

**Note:** This function runs AMODE(64) in HCR7720 and later versions of ICSF.

CSFACKDS supports dynamic updating, deleting, and adding of records to the in-storage copy of the ICSF Cryptographic Key Data Set (CKDS), which holds the DES private keys.

**Function Prototype**

call CSFACKDS

(  return_code,  
  reason_code,  
  exit_data_length,  
  exit_data,  
  entry_code,  
  label,  
  key_type  
  output_area,  
  SPB )

**Input**

On entry to this routine:

- **exit_data_length** is an integer that represents the length of the data that is passed to the installation exit.

- **exit_data** is a character string containing the data that is passed to the installation exit.

- **entry_code** is an integer that represents the 4-byte hexadecimal value indicating the action to be taken. Possible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'00000001'</td>
<td>TOKEN</td>
<td>Retrieve a token from the in-storage CKDS. (If the token is not found, return a return code rather than abending.)</td>
</tr>
<tr>
<td>X'0000000A'</td>
<td>ADD</td>
<td>Add a null record to the linear area of the in-storage CKDS (CKT). The input record is passed in the output_area field.</td>
</tr>
<tr>
<td>X'0000000B'</td>
<td>UPDATE</td>
<td>Update a record in the CKT. The input record is passed in the output_area parameter field.</td>
</tr>
<tr>
<td>X'0000000C'</td>
<td>DELETE</td>
<td>Delete a record in the CKT.</td>
</tr>
</tbody>
</table>

- **label** is a character string containing 64-bytes, left-justified and padded on the right with blanks, containing the name of the key/record.

- **key_type** is a character string containing 8 EBCDIC characters specifying the type of key record to be processed.
ANY for generic retrieval.

Otherwise, allowable key types are:

DATA
DATAXLAT
EXPORTER
IMPORTER
IPINENC
MAC
MACVER
NULL
OPINENC
PINGEN
PINVER
CV

output_area is a character string containing an actual record for entry codes of ADD and UPDATE and is ignored for DELETE.

SPB is a character string containing the service parameter block (SPB) or zero.

Output
On successful exit from this function:

return_code is an integer that represents the general result of the callable service.

reason_code is an integer that represents the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it to indicate specific processing problems.

output_area is a character string containing the actual key record returned, if the value of entry_code was TOKEN; otherwise, this value is ignored.

Return and Reason Codes
Common return codes (decimal values in register 15) and reason codes (returned in register 0 and in the reason_code parameter) generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code (dec)</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>8</td>
<td>10012</td>
<td>Key not found.</td>
</tr>
<tr>
<td>8</td>
<td>16004</td>
<td>Request failed by RACF.</td>
</tr>
<tr>
<td>8</td>
<td>16020</td>
<td>Function not allowed for system key.</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>CSF not active.</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>Exit has failed.</td>
</tr>
<tr>
<td>12</td>
<td>10020</td>
<td>MAC failed.</td>
</tr>
</tbody>
</table>
### Return Code (dec)  | Reason Code | Meaning
--- | --- | ---
12  | 10024 | Key failed by installation exit.
12  | 10036 | Label not unique.
12  | 10052 | No space in CKT for dynamic adds.
16  | 4     | Unrecoverable failure in this routine.

### Abend Code  | Reason Code | Meaning
--- | --- | ---
x’18F’ | 160 | Invalid entry code.
      | 256 | Invalid return code from exit.
      | 258 | Invalid return code from CSFPCMF.
**CSFAPKDS - Access ICSF Public Key Data Set**

Note: This function is available on the z/OS host.

CSFAPKDS supports dynamic updating, deleting, and adding of records to the ICSF Public Key Data Set (PKDS), which holds the PKA and DSS keys.

**Function Prototype**

call CSFAPKDS
(    return_code,
    reason_code,
    exit_data_length,
    exit_data,
    label,
    token_length,
    token,
    function,
    SPB )

**Input**

On entry to this routine:

exit_data_length is an integer that represents the length of the data that is passed to the installation exit.

exit_data is a character string containing the data that is passed to the installation exit.

label is a character string containing the name of the key/record which is 64-bytes, left-justified, and padded on the right with blanks.

token_length is the length of the block available at token. The maximum token size is 2500 bytes.

token is a character string containing the key token for UPDTENUL and UPDATE requests.

function is a character string containing eight EBCDIC characters specifying the function to be performed, and is left-justified and padded on the right with blanks, as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ</td>
<td>Read record</td>
</tr>
<tr>
<td>CREATE</td>
<td>Create record</td>
</tr>
<tr>
<td>UPDTENUL</td>
<td>Replace null token</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Update token</td>
</tr>
<tr>
<td>DELLABEL</td>
<td>Delete record</td>
</tr>
<tr>
<td>DELTOKEN</td>
<td>Replace token with nulls</td>
</tr>
</tbody>
</table>

SPB is a character string containing the service parameter block (SPB) or zero.
Output
On successful exit from this routine:

return_code is an integer that represents the general result of the callable service.

reason_code is an integer that represents the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it to indicate specific processing problems.

token_length is the actual length of the token returned in token, if the function specified was READ.

token is a character string containing the actual token returned if the request was a READ request.

Return and Reason Codes
Common return codes (decimal values in register 15) and reason codes (returned in register 0 and in the reason_code parameter) generated by this routing are:

<table>
<thead>
<tr>
<th>Return Code (dec)</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>4</td>
<td>14008</td>
<td>Authentication code mismatch.</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Application error.</td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td>Cryptographic facility not available.</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>Your call to an ICSF callable service resulted in an abnormal ending.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abend Code</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x’18F’</td>
<td>174</td>
<td>Unknown FUNCTION code in the parameter list.</td>
</tr>
</tbody>
</table>
CSFGCCPN - *Send a Request to the Coprocessor*

**Note:** This function is available on the z/OS host.

**Note:** This function runs AMODE(64) in HCR7720 and later versions of ICSF.

CSGACCPN sends a request to the coprocessor and analyzes and reports the reply.

**Function Prototype**

call CSFGCCPN

(   return_code,  
    reason_code,  
    request_addr,  
    request_len,  
    response_addr,  
    response_len,  
    caller_alet,  
    ccp_index,  
    ccp_serial_nr,  
    identifier,  
    serialization,  
    domain_index,  
    pcifunction,  
    input_data_block,  
    output_data_block,  
    SPB )

**Input**

On entry to this routine:

- **request_addr** is a character string containing the address where the message header is built.

- **request_len** is an integer that represents the total length of the message.

- **response_addr** is a character string containing the address of where the response message is copied.

- **response_len** is an integer that represents the amount of available space for the response message.

- **caller_alet** is an integer that represents the ALET corresponding to **request_addr** and **response_addr**.

- **ccp_index** is an integer that represents a target cryptographic coprocessor (CCP) index or -1. The CCP index is one greater than the coprocessor number. If it is irrelevant as to which coprocessor the service is directed, specify an index of -1.

- **ccp_serial_nr** is a character string containing 8 bytes of EBCDIC characters specifying the CCP serial number. If you identify the coprocessor by index, or if it is irrelevant as to which coprocessor the service is directed, specify NOT APPL as the serial number.
**identifier** is a character string containing 8 bytes of binary zeroes.

**serialization** is an integer that represents the type of serialization. Specify 0 for shared.

**domain_index** is an integer that represents -1 to indicate that the domain is not applicable.

**pcifunction** is a character string containing 2 bytes of hexadecimal data specifying the subfunction code of the coprocessor command processor to which the request is to be sent. A list of subfunction codes for the standard CCA API functions are located in the file `cmncryt2.h`. The hexadecimal data is in big endian form. (For example, the subfunction code for the Clear PIN encrypt service is “PE” in ASCII, or X’5045’ as a two-byte integer.

**input_data_block** is a character string containing 20 bytes of binary zeroes.

**output_data_block** is a character string containing 20 bytes of binary zeroes.

The **input_data_block** and **output_data_block** are only binary zeroes if there is no request data block or reply data block. If there is a request data block, the **input_data_block** field is a structure as defined below. If there is a reply data block, the **output_data_block** field is a structure as defined below:

- Integer containing the ALET of the data
- 4 bytes of hexadecimal data containing the address of the data
- Integer containing the length of the data (length must be a multiple of 8 bytes)
- Integer containing the storage protect key
- Character string of 4 bytes of binary zeroes

**SPB** is a character string containing the service parameter block (SPB) or zero.

**Output**

On successful exit from this routine:

**return_code** is an integer that represents the general result of the callable service.

**reason_code** is an integer that represents the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it to indicate specific processing problems.

**response_addr** is a character string containing the response message.

**response_len** is an integer that represents the actual length of the response message.

**ccp_index** is an integer that represents the index of the CCP that performed the service.

**ccp_serial_nr** is a character string containing the serial number of the coprocessor that performed the service.

**output_data_block** is a character string containing 20 bytes of zeroes if there is no reply data block, or else it is a character string containing the reply data block.
**Return and Reason Codes**
In general the return and reason codes from this function will have been generated by the coprocessor, in the course of completing the request identified by `pcifunction`.

Common return codes (decimal values in register 15) and reason codes (returned in register 0 and in the `reason_code` parameter) generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code (dec)</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Application error.</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>CSF not active</td>
</tr>
<tr>
<td>12</td>
<td>X’2B28’</td>
<td>Service has failed.</td>
</tr>
<tr>
<td>12</td>
<td>X’2B34’</td>
<td>Cryptographic coprocessor is not available.</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>Unrecoverable failure in this routine.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abend Code</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>X’18F’</td>
<td>80</td>
<td>NQ incomplete. Error in the interface to the coprocessor.</td>
</tr>
<tr>
<td></td>
<td>165</td>
<td>Bad internal parameters between internal service calls.</td>
</tr>
<tr>
<td></td>
<td>258</td>
<td>Invalid condition code from the CSF (instruction) macro</td>
</tr>
<tr>
<td></td>
<td>267</td>
<td>Full CCP queue CSFACCPS</td>
</tr>
<tr>
<td></td>
<td>268</td>
<td>Bad response code DQ/NQ/PQ AP</td>
</tr>
<tr>
<td></td>
<td>408</td>
<td>PSMID in returned message does not match sent message.</td>
</tr>
</tbody>
</table>
CSFGCPRB - Build a CPRB

Note: This function is available on the z/OS host.

Note: This function runs AMODE(64) in HCR7720 and later versions of ICSF.

CSFGCPRB builds the CPRB for a call to a CCA application on a coprocessor.

Function Prototype

call CSFGCPRB
  (return_code,
   flags,
   subfunction_code,
   rule_count,
   rules,
   number_vuds,
   vud_list,
   number_keys,
   key_list,
   request_data_block_length,
   request_data_block,
   reply_data_block_length,
   reply_data_block,
   request_CPRB_len,
   request_CPRB,
   reply_CPRB,
   SPB )

Input

On entry to this routine:

flags is an integer that represents 4 bytes of checkpoint flags. Specify 4 bytes of binary zeroes.

subfunction_code is a character string containing 2 bytes of hexadecimal data specifying the subfunction code of the processor command processor to which the request is to be sent. A list of subfunction codes for the standard CCA API functions is located in the file cmncryt2.h. The hexadecimal data is in big endian form. (For example, the subfunction code for the Clear PIN encrypt service is ‘PE’ in ASCII, or X’5045’ as a two-byte integer.)

rule_count is an integer that represents the number of keywords passed.

rules is a character string containing the keywords to be put into the request parameter block.

number_vuds is an integer that represents the number of elements to be put into the verb unique data block.

vud_list is a character string containing the elements to be put in the verb unique data block. The
vud_list is an array of 12-byte entries:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>length of verb unique data field</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>flag1 may be used to indicate the type of data</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>flag2 may be used to indicate the type of data</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>Address of the data item to be added</td>
</tr>
</tbody>
</table>

number_keys is an integer that represents the number of elements to be put into the key block.

key_list is a character string containing the elements to be put into the key block. The key_list is an array of 12-byte entries:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Length of key token</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>flag used to indicate the type of data</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>reserved</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>Address of key token to be added</td>
</tr>
</tbody>
</table>

request_data_block_length is an integer that represents the length of the data in the request data block.

request_data_block is a character string containing the address of the request data block.

reply_data_block_length is an integer that represents the length of the data in the reply data block.

reply_data_block is a character string which is to contain the address of the reply data block.

SPB is a character string containing the service parameter block (SPB).

Output
On successful exit from this routine:

return_code is an integer that represents the general result of the callable service.

request_CPRB_len is an integer that represents the length of the request CPRB and parameter block.

request_CPRB is a character string containing the address of the request CPRB and parameter block.

reply_CPRB is a character string containing the address of the reply CPRB and parameter block.

reply_data_block_length is an integer that represents the length of the data in the reply data block.

reply_data_block is a character string containing the address of the reply data block.
**Return and Reason Codes**

Common return codes (decimal values in register 15) and reason codes (returned in register 0 and in the `reason_code` parameter) generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code (dec)</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Too much data for the request parameter block.</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>Your call to an ICSF callable service resulted in an abnormal ending.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abend Code</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x'18F'</code></td>
<td>178</td>
<td>Unable to obtain storage for CPRB</td>
</tr>
</tbody>
</table>
**CSFADSCP - Destroy a CPRB**

*Note:* This function is available on the z/OS host.

*Note:* This function runs AMODE(64) in HCR7720 and later versions of ICSF.

CSFADSCP releases the storage acquired for the request and reply CPRBs by CSFACPRB. The address of the storage is stored in the SPB.

**Function Prototype**

call CSFADSCP  
(return_code,  
SPB )

**Input**

On entry to this routine:

**SPB** is a character string containing the service parameter block (SPB).

**Output**

On successful exit from this routine:

**return_code** is an integer that represents the general result of the callable service.

**Return and Reason Codes**

Common return codes (decimal values in register 15) and reason codes (returned in register 0 and in the reason_code parameter) generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code (dec)</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>No address in the SPB.</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>No CPRB found at this address.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abend Code</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'18F'</td>
<td>180</td>
<td>Invalid pointer in SPB for CPRB</td>
</tr>
</tbody>
</table>
**CSFGVLPB - Validate a CPRB**

*Note:* This function is available on the z/OS host.

*Note:* This function runs AMODE(64) in HCR7720 and later versions of ICSF.

CSFGVLPB checks the reply CPRB and parameter block for validity. The service checks the following:

1. The CPRB fields for valid values.
2. That the domain matches the CDX.
3. The reply parameter block address and length.
4. If there is a reply parameter block:
   a. Step through the parameter block and check that the element lengths add up to the overall length of the parameter block.
   b. Determine the address of the verb unique data block.
   c. Step through the verb unique data block and check that the element lengths add up to the overall length of the block.
   d. Determine the address of the key block.
   e. Step through the key block and check that the element lengths add up to the overall length of the block.
5. Parse the service return and reason codes.
6. Return to the caller.

**Function Prototype**

call CSFGVLPB
(return_code,
reply_cprb,
service_return_code,
service_reason_code,
parm_block_address,
vud_block_address,
key_block_address,
SPB )

**Input**

On entry to this routine:

*reply_cprb* is a character string containing the address of the reply CPRB.

*SPB* is a character string containing the service parameter block (SPB).
Output
On successful exit from this routine:

return_code is the general result of the callable service.

service_return_code is an integer that represents the return code within the CPRB.

service_reason_code is an integer that represents the reason code within the CPRB.

parm_block_address is a character string containing the address of the reply parameter block.

vud_block_address is a character string containing the address of the verb unique data block.

key_block_address is a character string containing the address of the key block.

Return and Reason Codes
Common return codes (decimal values in register 15) and reason codes (returned in register 0 and in the reason_code parameter) generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code (dec)</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>The CPRB is not valid.</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>The parameter block is not valid.</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>No CPRB found at this address.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abend Code</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'18F'</td>
<td>179</td>
<td>Domain in CPRB doesn't match CCVE</td>
</tr>
<tr>
<td></td>
<td>180</td>
<td>Invalid pointer in SPB for CPRB</td>
</tr>
<tr>
<td></td>
<td>183</td>
<td>Reply CPRB or parameter block is bad</td>
</tr>
</tbody>
</table>
**CSFGPBLK - Parse a CPRB**

**Note:** This function is available on the z/OS host.

**Note:** This function runs AMODE(64) in HCR7720 and later versions of ICSF.

CSFGPBLK parses the next element from the verb unique data block or the key block.

**Function Prototype**

call CSFGPBLK
(return_code,
block_address,
element_ptr,
element_length,
element_flag,
element_data_ptr,
SPB )

**Input**

On entry to this routine:

- **block_address** is a character string containing the address of the block to be parsed.
- **element_ptr** is a character string containing the address of the last record found in the block, or null if the first record in the block is to be found.
- **SPB** is a character string containing the service parameter block (SPB).

**Output**

On successful exit from this routine:

- **return_code** is an integer that represents the general result of the callable service.
- **element_ptr** is a character string containing the address of the next record found in the block.
- **element_length** is an integer that represents the length of the data returned at the address specified by the **element_data_ptr** parameter.
- **element_flag** is a character string containing the flag from the record found.
- **element_data_ptr** is a character string containing the address of the data from the record found.
Return and Reason Codes

Common return codes (decimal values in register 15) and reason codes (returned in register 0 and in the `reason_code` parameter) generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code (dec)</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>The operation was successful. There are no more records in the block.</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>The last record address is not valid.</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>The block is not valid.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abend Code</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'18F'</td>
<td>180</td>
<td>Invalid pointer in SPB for CPRB</td>
</tr>
</tbody>
</table>
**CSFAPKTV - Validate/Initialize an RSA or DSS Key Token**

**Note:** This function is available on the z/OS host.

CSFAPKTV establishes addressability to DSS or RSA key token parts and verifies the correctness of the key token.

**Function Prototype**

call CSFAPKTV
(return_code, reason_code, function, input_token_length, input_token, SPB )

**Input**

On entry to this routine:

*function* is a character string of eight characters containing the function to be performed by the service. Specify VALIDATE.

*input_token_length* is an integer that represents the actual byte length of the token body being passed.

*input_token* is a character string containing the actual token at offset zero.

*SPB* is a character string containing the service parameter block (SPB).

**Output**

On successful exit from this routine:

$return_code$ is an integer that represents the general result of the callable service.

$reason_code$ is an integer that represents the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems.

*input_token* is a character string containing the key token updated with debugging information and entries in the offset tables.

**Return and Reason Codes**

Common return codes (decimal values in register 15) and reason codes (returned in register 0 and in the $reason_code$ parameter) generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code (dec)</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>8</td>
<td>2040</td>
<td>Invalid first byte (must be X’1E’ or X’1F’).</td>
</tr>
<tr>
<td>8</td>
<td>11012</td>
<td>Invalid flags in the token.</td>
</tr>
<tr>
<td>8</td>
<td>11016</td>
<td>Invalid hash value.</td>
</tr>
<tr>
<td>Return Code (dec)</td>
<td>Reason Code</td>
<td>Meaning</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>8</td>
<td>11020</td>
<td>Token too old—invalid RMK or SMK hash.</td>
</tr>
<tr>
<td>8</td>
<td>11024</td>
<td>Token missing one or more required parts.</td>
</tr>
<tr>
<td>8</td>
<td>11044</td>
<td>Invalid PKA token.</td>
</tr>
<tr>
<td>8</td>
<td>11092</td>
<td>Invalid PKA subsection or TLV.</td>
</tr>
</tbody>
</table>
**CSFGAPSI - Communication Between Services and Coprocessor**

**Note:** This function is available on the z/OS host.

**Note:** This function runs AMODE(64) in HCR7720 and later versions of ICSF.

CSFGAPSI handles communication between the callable services and the coprocessor in the following manner:

1. Builds a CPRB
2. Passes the CPRB to the coprocessor
3. Validates the CPRB on return from the coprocessor
4. Returns a verb data block and a key block
5. Releases the storage obtained for the CPRB

**Note:** This function may be invoked using the CSFMAPSI macro which is provided with the UDX Development Toolkit.

**Function Prototype**

```
call CSFGAPSI
(return_code, reason_code, flags, subfunction_code,
 CCP_index, CCP_serial_number, CCP_domain, rule_count,
 rules, number_vuds, vud_list, number_keys, key_list,
 reply_vud_block_length, reply_vud_block_address,
 reply_key_block_length, reply_key_block_address,
 request_data_block_length, request_data_block,
 request_data_block_alet, reply_data_block_length,
 reply_data_block, reply_data_block_alet, SPB )
```

**Input**

On entry to this routine:

- **flags** is a character string containing 4 bytes of checkpoint flags. Specify 4 bytes of binary zeroes.

- **subfunction_code** is a character string containing 2 bytes of hexadecimal data specifying the subfunction code of the coprocessor command processor to which the request is to be sent. A list of the
subfunction codes for the standard CCA API functions are in file cmncryt2.h. The hexadecimal data is in big endian form. (For example, the subfunction code for the Clear PIN encrypt service is ‘PE’ in ASCII, or X’5045’ as a two-byte integer.)

**CCP_index** is an integer that represents the index number of the coprocessor which is to execute the request. If it is irrelevant as to which coprocessor the service is directed, specify an index of -1.

**CCP_serial_number** is a character string containing 8 bytes of EBCDIC characters specifying the serial number of the coprocessor which is to execute the request. If it is irrelevant as to which coprocessor the service is directed, specify NOT APPL as the serial number.

**CCP_domain** is an integer that represents the domain used to call to CSFGCCPN (a number from 0 to 15). Usually specified as -1 to indicate that the domain is not applicable.

**rule_count** is an integer that represents the number of keywords supplied in the **rule_array**.

**rules** is a character string containing the keywords that supply control information to the callable service. Specify a blank character if the **rule_count** field specifies a count of zero.

**number_vuds** is an integer that represents the number of items of verb unique data in the VUD list (**vud_list**).

**vud_list** is a character string containing the verb unique data (VUD) list. The verb unique data specified depends upon the verb being invoked. Specify a value of zero if the **number_vuds** field specifies a count of zero. Each element in the list consists of four items:

- **vud_length** is an integer that represents the length of the VUD element.
- **vud_flag** is a two-byte character string containing flag information for the verb. The flag information is related to the **vud_data** item. If there is no **vud_flag** information, specify a value of ‘FFFF’x here and in the **vud_no_flag** item.
- **vud_no_flag** is a two-byte character string that contains the value ‘0000’x if **vud_flag** data is present, or the value ‘FFFF’x if there is no **vud_flag** data.
- **vud_data** is the address of the VUD data for this VUD element.

**number_keys** is an integer that represents the number of items in the KEY list (**key_list**).

**key_list** is a character string containing the key list. Specify a value of zero if the **number_keys** field specifies a count of zero. Each element in the list consists of three items:

- **key_length** is an integer that represents the length of the key in the **key_data** element.
- **key_flag** is a two-byte character string containing the value ‘0000’x. (zSeries does not use flag information for keys.)
- **key_data** is a character string containing the key to be passed to the verb.

**reply_vud_block_length** is an integer that represents the data area length the caller provided for the reply_VUD block or zero.

**reply_key_block_length** is an integer that represents the data area length the caller provided for the reply_key block or zero.

**request_data_block_length** is an integer that represents the length of the request data block. The maximum length of the data which may be passed is 12K bytes.
request_data_block is a character string containing the Request Data Block. Specify a blank character if the request_data_block_length specifies a length of zero.

request_data_block_alet is an integer that represents the ALET that is used when building the request data block parameter for the call to CSFACCPN. Specify 0.

reply_data_block_length is an integer that represents the length of the storage allocated for the reply.

reply_data_block is a character string containing the address of the storage allocated for the reply.

reply_data_block_alet is an integer that represents the ALET that is used when building the request data block parameter for the call to CSFACCPN. Specify 0.

SPB is a character string containing the service parameter block (SPB).

**Output**

On successful exit from this routine:

return_code is an integer that represents the general result of the callable service.

reason_code is an integer that represents the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems.

CCP_index is an integer that represents the index of the coprocessor that performed the service.

CCP_serial_number is a character string containing the serial number of the coprocessor that executed the request.

rules is a character string containing the keywords that supply control information to the callable service.

reply_vud_block_address is a character string containing the address of the caller’s data area to hold the reply VUD data.

reply_key_block_address is a character string containing the address of the caller’s data area to hold the reply KEY data.

reply_data_block_length is an integer that represents the length of the Reply Data Block.

reply_data_block is a character string containing the Reply Data Block. This will be a blank character if the reply_data_block_length specifies a length of zero.
**Return and Reason Codes**

Common return codes (decimal values in register 15) and reason codes (returned in register 0 and in the `reason_code` parameter) generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code (dec)</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>8</td>
<td>11000</td>
<td>Invalid length field.</td>
</tr>
<tr>
<td>12</td>
<td>11056</td>
<td>Incomplete response from the PCICC.</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>Your call to an ICSF callable service resulted in an abnormal ending.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abend Code</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'18F'</td>
<td>165</td>
<td>Bad internal parameters between internal service calls</td>
</tr>
</tbody>
</table>
**CSFASEC - Check Authorization**

**Note:** This function is available on the z/OS host.

CSFASEC checks authorization to a RACF-protected or security exit-protected resource.

**Function Prototype**

call CSFASEC
(return_code,
reason_code,
resource,
resource_length,
resource_class,
SPB )

**Input**

On entry to this routine:

- **resource** is a character string containing the name of the resource to be authority-checked.
- **resource_length** is an integer that represents the length of the resource name.
- **resource_class** is a character string containing 8 EBCDIC characters, CSFKEYS or CSFSERV.
- **SPB** is a character string containing the service parameter block (SPB).

**Output**

On successful exit from this routine:

- **return_code** is an integer that represents the general result of the callable service.
- **reason_code** is an integer that represents the result of the callable service that is returned to the application program. Each return code has different reason codes assigned to it that indicate specific processing problems.

**Return and Reason Codes**

Common return codes (decimal values in register 15) and reason codes (returned in register 0 and in the **reason_code** parameter) generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code (dec)</th>
<th>Reason Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>8</td>
<td>16000</td>
<td>Authorization failed.</td>
</tr>
</tbody>
</table>
BuildParmBlock - Build a Parameter Block

Note: This function is available on both the workstation host and the coprocessor.

BuildParmBlock constructs a parameter block, containing a two-byte length field, followed by a variable number of data fields. The function accepts pairs of data descriptors, each consisting of a pointer to the data item, and a value containing the item's length. For each pair, the first value is an unsigned short containing the length, and the second value is a pointer giving the location of the data.

BuildParmBlock is used in building the Reply Parameter Block for the response to a host request as well as the Request Parameter Block.

The function result contains the total length of the block built by the function.

Function Prototype

USHORT BuildParmBlock ( UCHAR *pBuffer,
                        USHORT  pairs,
                        USHORT  Data1_length,
                        UCHAR   *pData1
                        ... *pData1 )

Input

On entry to this routine:

pBuffer is the starting address of the parameter block section to be built.

pairs is the number of argument pairs which are to be added to the parameter block section.

Datai_length is the length of the ith item, in bytes.

Datai is a pointer to the ith data item to be added.

Note: If no items are to be added, Data1_length = 0 and Data1 = NULL.

If 2 or more items of verb unique data are to be added, each item should be preceded by a short field containing the length of the individual item +2. This will allow the function FindNextDataBlock to parse the result.

BlockLength = 0;
pCprb = (CPRB *) & (Buffer.request_parm_buffer[0]);
pRequestBlock = & (Buffer.request_parm_buffer[0]) + sizeof(CPRB_structure);
**Step one: add the subfunction code**

\[
\text{BlockLength} += 2 \\
* ((\text{USHORT *} \ p\text{ReqBlk}) = \text{htoas ( CCAFNC1\_ID )};
\]

**Step two: add the rule array**

\[
\text{BlockLength} += \text{BuildParmBlock}(p\text{RequestBlock}+\text{BlockLength}, \\
1, /* adding 1 rule array */ \\
(*p\text{RuleArrayCount}) \times 8, /* length of rule array */ \\
p\text{RuleArray});
\]
Step three: add the verb unique data

\[
\text{Data1Length} = \text{htoas ( Data1Size + sizeof(short) )}; \\
\text{Data2Length} = \text{htoas ( Data2Size + sizeof(short) )}; \\
\text{BlockLength} += \text{BuildParmBlock(pRequestBlock + BlockLength,} \\
\quad 4, /* adding 2 data items, plus their lengths */ \\
\quad \text{sizeof(short), \\n\quad &Data1Length, /* length of 1st item, including this field */ \\
\quad \text{Data1Size, pData1,} \\
\quad \text{sizeof(short), \\n\quad &Data2Length, /* length of 2nd item, including this field */ \\
\quad \text{Data2Size, pData2});}
\]

Step four: add the key blocks

\[
\text{KeyHeaderI.Length} = \text{htoas ( KeyTokenLength + sizeof(KEY_FIELD_HEADER) )}; \\
\text{KeyHeaderI. Flags} = \text{htoas ( storageOptions )}; \\
\text{BlockLength} += \text{BuildParmBlock( pRequestBlock + BlockLength,} \\
\quad 2, /* adding a key block header and a key token */ \\
\quad \text{sizeof(KEY_FIELD_HEADER), \\n\quad &KeyHeaderI,} \\
\quad \text{KeyTokenLength, \\n\quad &KeyToken) ;}
\]

Output

On successful exit from this function:

\text{BuildParmBlock returns the total length of the block built by the function. The buffer at pBuffer contains the parameter block.}

Return and Reason Codes

This function has no return codes.
Notes

Building the Parameter Blocks

There are three types of parameter blocks: the rule array block, the verb unique data block, and the key block. They must all be present in the CPRB message, in this order. If any of the blocks is unnecessary, a length field of 2 must be present to indicate an empty parameter block. This may be achieved by calling
BuildParmBlock(pBuffer, 0, 0, NULL);

The rule array is a byte array, with 8 bytes for each rule present. Each rule is 8 bytes long, padded on the right with spaces. It is important to note that the entire 8 bytes are compared - these are not strings as C and C++ define them. No allowance is made for a null terminator, so be careful when copying rule data into the array.

For more information about key block structures, see “Key Blocks” on page 1-15.

See Appendix D, “UDX Sample Code - Coprocessor Piece” on page D-1 for sample code which includes parameter block building.

Byte Alignment of Structures

It is important that all structures which are passed from the host to the coprocessor or the coprocessor to the host be aligned consistently. If you are passing a user-defined structure to the coprocessor, either as verb unique data or as key data, you must ensure that your compilers align the host and coprocessor structures on the same boundaries.
**Cas_proc_retc - Prioritize Return Code**

**Note:** This function is available on the coprocessor.

Cas_proc_retc is used when you encounter an error, and need to set a return code in the reply CPRB. The function compares your new return code, passed in `msg`, with the return code already present in the CPRB. It uses a priority evaluation scheme to decide whether your new return code, or the one already in the CPRB indicates a more critical error, and it leaves whichever is higher priority in the CPRB.

**Function Prototype**

```c
long Cas_proc_retc ( CPRB_structure *pCprb, 
                    long msg )
```

**Input**

On entry to this routine:

- `pCprb` is a pointer to the reply CPRB structure.
- `msg` is the CCA (SAPI) return code for the error just encountered.

**Output**

On successful exit from this routine:

- `pCprb->return_code` and `pCprb->reason_code` contain the reason codes of `msg`, if the return code of `msg` was greater than the return code formerly in `pCprb->return_code`.

**Return and Reason Codes**

Common return codes generated by this routine are:

- **OK** The operation was successful.
- **ERROR** The return code in `msg` was greater than Warning level (level 4).
**CSNC_SP_SCSRFBSS - Send a Request to the Coprocessor**

**Note:** This function is available on the workstation host.

CSNC_SP_SCSRFBSS passes a request to the coprocessor, and receives the response.\(^1\) The input and output are passed using a pointer to the CPRB structure. The SAPI error code is returned in the variable pointed to by \(pMsg\).

If the user is currently logged on to the CCA application in the coprocessor, requests and replies are protected using a MAC computed with the user’s session key. This processing is handled automatically when you use CSNC_SP_SCSRFBSS.

**Function Prototype**

```c
long CSNC_SP_SCSRFBSS ( CPRB_ptr pCprb,
                        long *pMsg )
```

**Input**

On entry to this routine:

- \(pCprb\) is a pointer to the CPRB structure. It contains the request CPRB, with the concatenated Request Parameter Block.

- \(pMsg\) is a pointer to a variable for the return code of the function.

**Output**

On successful exit from this routine:

- \(pCprb\) contains the reply CPRB, with the concatenated Reply Parameter Block.

- \(pMsg\) is a pointer to a location where the SAPI return code and reason code is stored, on return from the requested function.

CSNC_SP_SCSRFBSS returns **OK** if there were no errors or **ERROR** if the \(pMsg\) buffer contains an error.

**Return and Reason Codes**

Common return codes generated by this routine are:

- **E_ALLOCATE_MEM**  
  Unable to allocate memory for checking data.

- **RT_SWERR**  
  An error was encountered in the CPRB.

- **E_INVALID_MAC_VAL**  
  The data returned from the coprocessor could not be validated.

Other error codes may be returned, depending on the functions called in the coprocessor section of the code.

---

\(^1\) The name of this function is rather obscure. It is inherited from the FBSS interfaces, which later became LAN/ DP, the LAN Distributed Platform. This function was a remote procedure call under FBSS, to pass a request to a server where it would be processed. This is also the origin of the CPRB.
CSUC_BULDCPRB - Build CPRB

Note: This function is available on the workstation host.

CSUC_BULDCPRB builds a new CPRB from a request or reply block built with BuildParmBlock and an optional data field.

Function Prototype

```c
void CSUC_BULDCPRB
( CPRB_ptr pCprb,
  unsigned char *fid_ptr,
  unsigned short rqpb_l,
  unsigned char *rqpb_ptr,
  unsigned long rqdb_l,
  unsigned char *rqdb_ptr,
  unsigned short rppb_l,
  unsigned char *rppb_ptr,
  unsigned long rpdb_l,
  unsigned char *rpdb_ptr )
```

Input

On entry to this routine:

- `pCprb` is a pointer to the buffer where the new CPRB is returned.
- `fid_ptr` is a pointer to the two-byte main function ID.
- `rqpb_l` is the length of the Request Parameter Block, in bytes.
- `rqpb_ptr` is the address of the Request Parameter Block.
- `rqdb_l` is the length of the Request Data Block, in bytes.
- `rqdb_ptr` is a pointer to the Request Data Block. This block must be a multiple of 8 bytes long.
- `rppb_l` is the length of the Reply Parameter Block, in bytes.
- `rppb_ptr` is the address of the Reply Parameter Block.
- `rpdb_l` is the length of the Reply Data Block, in bytes.
- `rpdb_ptr` is a pointer to the Reply Data Block. This block must be a multiple of 8 bytes long.
Output
On successful exit from this routine:

The buffer pointed to by pCprb contains a CPRB structure with the following values:

• function_id contains the function ID specified in the call.
• req_parm_block_length is the length of the request parameter block.
• req_parm_block is the address of the request parameter block (it immediately follows the CPRB).
• req_data_block_length is the length of the data block provided in the call.
• req_data_block is a pointer to the data in the host memory.
• reply_msg_block_length is the length of the reply parameter block.
• reply_parm_block is the address of the reply parameter block (it immediately follows the request parameter block).
• reply_data_block_length is the length of the reply data block.
• reply_data_block is a pointer to the reply data block in the host memory.

Note: The request data block and reply data block are not copied into the message which is sent to the coprocessor. The coprocessor will read them directly from the host machine. The communications method requires that they be a multiple of 8 bytes long.

Return and Reason Codes
This function has no return codes.
CSUC_PROCRETC - Prioritize Return Code

Note: This function is available on the workstation host.

CSUC_PROCRETC examines an error code, compares it to the return code already in effect, and sets that return code to whichever of the two is higher priority. If the new return code, passed in msg, is more serious than the return code already in the variables pointed to by return_code_ptr and reason_code_ptr, then the values pointed to by those parameters are replaced by the new code.

Function Prototype

long CSUC_PROCRETC ( long *return_code_ptr,
                       long *reason_code_ptr,
                       long    msg )

Input
On entry to this routine:

return_code_ptr is a pointer to the current SAPI return code for this verb.

reason_code_ptr is a pointer to the current SAPI reason code for this verb.

msg is an error code corresponding to a new problem, just detected. This code contains both the return code and the reason code for that error, concatenated in a single four-byte integer. The return code occupies the two high-order bytes, while the reason code occupies the two low-order bytes.

Output
On successful exit from this routine:

return_code_ptr contains the higher of the original value or the value of the two high bytes of msg.

reason_code_ptr contains the reason code matching the priority of return_code_ptr.

CSUC_PROCRETC returns OK if the message return code was a warning level (4) or lower, or ERROR if the return code was an error code.

Return and Reason Codes
This function has no return codes.
**FindFirstDataBlock - Search for Address of First Data Block**

*Note:* This function is available on both the workstation host and the coprocessor.

FindFirstDataBlock locates the address of the first data block in the Verb Unique Data (VUD) section of the parameter block attached to the specified CPRB. If the parameter block contains Verb Unique Data, the address of the first data block is returned and the function result is set to TRUE. If there is no Verb Unique Data, the function result is set to FALSE.

**Function Prototype**

```c
boolean FindFirstDataBlock( CPRB_structure *pCprb,
                         unsigned int ParmBlockChoice,
                         VUD_DATA_RECORD **ppFirstDataBlock )
```

**Input**

On entry to this routine:

- `pCprb` is a pointer to the CPRB, which has the parameter block attached.

- `ParmBlockChoice` is a value of either SEL_REQ_BLK or SEL_REPLY_BLK, indicating whether the structure you have passed is a Request Parameter Block or a Reply Parameter Block.

**Output**

On successful exit from this routine:

- `ppFirstDataBlock` contains the address of the first data block in the Verb Unique Data.

**Return and Reason Codes**

This function has no return codes.
**FindNextDataBlock - Search for Address of Next Data Block**

*Note:* This function is available on both the workstation host and the coprocessor.

Given the address of a block in the Verb Unique Data (VUD) section of a parameter block, FindNextDataBlock will find and return the address of the next data block within the same parameter block. If another data block exists, its address is returned and the function result is set to TRUE. If there is no other data block, the function result is set to FALSE.

**Function Prototype**

```c
boolean FindNextDataBlock(
    CPRB_structure   *pCprb,
    unsigned int    ParmBlockChoice,
    VUD_DATA_RECORD   *pThisDataBlock,
    VUD_DATA_RECORD **ppNextDataBlock )
```

**Input**

On entry to this routine:

- `pCprb` is a pointer to the CPRB, which has the parameter block attached.
- `ParmBlockChoice` is a value of either SEL_REQ_BLK or SEL_REPLY_BLK, indicating whether the structure you have passed is a Request Parameter Block or a Reply Parameter Block.
- `pThisDataBlock` is a pointer to the current data block. The function attempts to find the data block following the one that this parameter points to.

**Output**

On successful exit from this routine:

- `ppNextDataBlock` contains the address of the data block after `pThisDataBlock` or NULL if none was found.
- `FindNextDataBlock` returns a boolean value indicating whether a block was found.

**Return and Reason Codes**

This function has no return codes.
find_first_key_block - Search for First Key Data Block

Note: This function is available on both the workstation host and the coprocessor.

find_first_key_block finds the address of the first key data block attached to the specified Parameter Block. If there is key data in the parameter block, it returns the address of the first key block, and sets the function result to TRUE. If there is no key data, it sets the function result to FALSE.

This function is used in conjunction with find_next_key_block, which is used to locate key blocks after the first one in the parameter block.

Function Prototype

boolean find_first_key_block( CPRB_structure *pCprb,
                               key_data_structure **first_keyblock,
                               unsigned int parm_block_choice )

Input

On entry to this routine:

pCprb is a pointer to the CPRB. The parameter block is expected to be concatenated to the CPRB.

parm_block_choice is a value of either SEL_REQ_BLK or SEL_REPLY_BLK, indicating whether the structure you have passed is a Request Parameter Block or a Reply Parameter Block.

Output

On successful exit from this routine:

first_keyblock contains the address of the first key block contained in the parameter block attached to pCprb.

find_first_key_block returns a boolean value of true if key data was found, false otherwise.

Return and Reason Codes

This function has no return codes.
**find_next_key_block - Find Address of Next Key Data Block**

**Note:** This function is available on both the workstation host and the coprocessor.

Given the address of a key data block, find_next_key_block will find and return the address of the next key data block within the specified Parameter Block. If the requested block exists, its address is returned and the function result is set to TRUE. If the block does not exist, the function result to FALSE.

This function is used in conjunction with **find_first_key_block**, which is used to locate the first key block in the parameter block.

Argument **parm_block_choice** indicates whether the parameter block being examined is a Request Parameter Block or a Reply Parameter Block.

**Function Prototype**

```c
boolean find_next_key_block( CPRB_structure   *pCprb,
                          key_data_structure   *this_keyblock,
                          key_data_structure **next_keyblock,
                          unsigned int    parm_block_choice)
```

**Input**

On entry to this routine:

- `pCprb` is a pointer to the CPRB. The parameter block is expected to be concatenated to the CPRB.
- `this_keyblock` is a pointer to a key block within the parameter block. The function attempts to locate the key block following this one.
- `parm_block_choice` is a value of either SEL_REQ_BLK or SEL_REPLY_BLK, indicating whether the structure you have passed is a Request Parameter Block or a Reply Parameter Block.

**Output**

On successful exit from this routine:

- `next_keyblock` contains the address of the key block following the one specified by `this_keyblock` or NULL if none was found.
- `find_next_key_block` returns a boolean value indicating whether new key data was found.

**Return and Reason Codes**

This function has no return codes.
InitCprbParmPointers - Initialize CPRB Parameter Pointers

Note: This function is available on the coprocessor.

InitCprbParmPointers initializes the pointers to the request and reply data buffers for both the input and the output CPRBs. It assumes that these buffers immediately follow the CPRB blocks.

Function Prototype

void InitCprbParmPointers
(
    CPRB_structure  *pInputCprb,
    CPRB_structure  *pOutputCprb
)

Input

On entry to this routine:

pInputCprb is a pointer to the input CPRB block, which has been passed to the coprocessor.

pOutputCprb is a pointer to the output CPRB block, which is returned to the host.

Output

This function has no output. On successful exit from this routine:

The req_parm_block and reply_parm_block fields of InputCprb and OutputCprb are correctly initialized.

Return and Reason Codes

This function has no return codes.
**keyword_in_rule_array - Search for Rule Array Keyword**

**Note:** This function is available on both the workstation host and the coprocessor.

`keyword_in_rule_array` determines whether a specified rule array keyword is present in the rule array passed with the given CPRB. The CPRB contains a pointer to the request parameter block, which in turn contains the rule array and related data.

Input parameters are a pointer to the CPRB, and a string containing the desired keyword. Note that comparisons are case-sensitive (although this should not matter, since all keywords should be in uppercase).

The function returns TRUE if the keyword is in the rule array, and FALSE if it is not.

**Note:** Before using this function, the caller should have verified the integrity of the CPRB using function `parm_block_valid`. See page 4-41 for information about `parm_block_valid`.

**Function Prototype**

```c
boolean keyword_in_rule_array ( CPRB_structure *pCprb,
                                rule_array_element keyword )
```

**Input**

On entry to this routine:

- `pCprb` is a pointer to the CPRB structure. The parameter block is expected to be concatenated to the end of the CPRB.

- `keyword` is the keyword you are looking for in the rule array.

**Output**

On successful exit from this routine:

- `keyword_in_rule_array` returns a boolean value indicating TRUE if the keyword is in the rule array, and FALSE if it is not.

**Return and Reason Codes**

This function has no return codes.
**parm_block_valid - Examine and Verify a Parameter Block**

**Note:** This function is available on both the workstation host and the coprocessor.

parm_block_valid examines the parameter block associated with a specified cooperative processing request block (CPRB), and verifies that the parameter block is valid. In particular, it verifies that all the sub-fields and their data are present, so that other functions can use that data with confidence that it is valid. It also verifies that the function ID in the CPRB is that which is expected.

The function returns a value of TRUE if the parameter block is OK, and returns FALSE if it is not.

**Function Prototype**

```c
boolean parm_block_valid ( CPRB_structure *pCprb,
                          unsigned int   parm_block_choice )
```

**Input**

On entry to this routine:

- `pCprb` is a pointer to the CPRB structure. The parameter block is expected to be concatenated to the end of the CPRB.

- `parm_block_choice` is a value of either SEL_REQ_BLK or SEL_REPLY_BLK, indicating whether the CPRB contains a Request Parameter Block or a Reply Parameter Block.

**Output**

On successful exit from this routine:

- `parm_block_valid` returns a boolean value of TRUE if the parameter block is OK, and returns FALSE if it is not.

**Return and Reason Codes**

This function has no return codes.
**rule_check - Verify Rule Array**

**Note:** This function is available on both the workstation host and the coprocessor.

rule_check can be used to verify the contents of the rule array in a received Request Parameter Block. In the simplest use, it gives a quick indication whether your rule array contains a valid combination of keywords. The function returns a value of TRUE if the rule array appears to be valid, or FALSE if it does not. If it returns FALSE, parameter `pReturn_Message` indicates the cause of the error.

The more complex way to use `rule_check` enables you to determine exactly what rule array elements appear in the request parameter block, without having to search through them yourself. It provides an ordered index, returned in parameter `pRule_value`, where each element of the index corresponds to one keyword, or one group of keywords where only one should be in the rule array. In each index element, the function returns a value indicating exactly what rule array keyword appeared, which is useful for the case where one keyword should be used out of a group. Examples later in this section may help clarify the process.

The function operates on the basis of a rule map, which describes the rule array elements you expect, and how they should be reported. The map is an array of `RULE_MAP` structures, where `RULE_MAP` is defined as follows.

```
typedef struct
{
    UCHAR keyword [ 9 ] ; /* 8 characters plus null terminator */
    BYTE order_no;        /* Rule array grouping number. */
    int map_value;        /* Element value within rule array grp */
} RULE_MAP;
```

*Figure 4-1. The RULE_MAP Structure*

The rule map contains one of these structures for each keyword that you expect for your verb. The three elements of the structure have the following meanings.

- **keyword** This is the eight-character rule array keyword.
- **order_no** This integer indicates which element of the returned `pRule_value` array should be set if the keyword in `keyword` is present in your rule array.
  
  A value of 1 refers to the first element of the array, corresponding to a C-language array index of 0.
- **map_value** This is the value that is stored in the output array `pRule_value` if the rule array keyword in `keyword` is in your rule array. The value is stored in the element indicated by `order_no`.

**Function Prototype**

```c
boolean rule_check( RULE_BLOCK *pParm_block,
                    unsigned int   rule_map_count,
                    RULE_MAP *pRule_map,
                    int            *pRule_value,
                    long           *pReturn_message )
```

Communications Functions  page 4-42
Input
On entry to this routine:

pParm_block is a pointer to the start of the rule array block in your Request Parameter Block. This should point to the start of the length field, not to the start of the first rule array element.

rule_map_count is the number of elements in the array specified by the pRule_map parameter.

pRule_map is a pointer to the rule map for this verb.

pRule_value is a pointer to the array that receives the output rule array index.

Note: On input, all elements of pRule_value must be set to the value INVALID_RULE.

Output
On successful exit from this routine:

pReturn_message is a pointer to the location where the function stores the error code, if the rule array is not correct.

pRule_value contains an array of integers, the $i$th integer is the map value of the keyword from the $i$th set which is present in the rule array, or INVALID_RULE if there is no keyword from that set.

Return and Reason Codes
Common return codes generated by this routine are:

- **E_RULE_ARRAY_KWD** Indicates that a required rule array keyword was missing. This also applies if only one keyword must be present out of a group of keywords, but none from the group are in your rule array.

- **E_RULE_ARRAY_COMBINE** Indicates that a rule array keyword appears more than one time in the input rule array. It can also indicate that more than one keyword appears from a group, where only one from the group is supposed to be present.

Examples
The following examples may help clarify the use of this function.

Checking the Rule Array for Verb CSNBPKI

CSNBPKI (Key Part Import) requires a rule array that contains exactly one of the following keywords.

- FIRST
- MIDDLE
- LAST
To check the incoming rule array for validity, rule_check can be used with the following three-element rule map.

```c
static RULE_MAP RuleMap [ 3 ] == (  
  { "FIRST   ", 1, 1 },  
  { "MIDDLE  ", 1, 1 },  
  { "LAST    ", 1, 1 } ) ;
```

Figure 4-2. Example Rule Map for Verb CSNBPKI

This is a group of keywords that are mutually exclusive. Only one can appear in the rule array, and for this verb, there are no other keywords that can appear. In the rule map, the values for order_no are the same for each keyword; they all specify a value of 1. This means that when any of these keywords appear in the rule array, the first element of the output array pRule_value is set. The value that goes into the first element of the output array is 1 for FIRST, 2 for MIDDLE, and 3 for LAST, as defined by the map_value elements of the rule map.

Since all three keywords have the same value for order_no, error code pReturn_message is set to E_RULE_ARRAY_COMBINE if more than one of the three keywords is present in your rule array.

Checking the Rule Array for Verb CSUAACI

CSUAACI (Access Control Initialization) has a slightly more complicated rule array than CSNBPKI described previously. It has the following characteristics.

- The rule array must contain exactly one of the following keywords.
  - INIT-AC
  - CHGEXPDT
  - CHG-AD
  - RESET-FC
- The rule array can optionally contain the keyword PROTECTD.
- The rule array can optionally contain the keyword REPLACE.

To check this rule array, we can use the following six-element rule map.

```c
static RULE_MAP RuleMap [ 6 ] == (  
  { "INIT-AC ", 1, 1 },  
  { "CHGEXPDT", 1, 2 },  
  { "CHG-AD ", 1, 3 },  
  { "RESET-FC", 1, 4 },  
  { "PROTECTD", 2, 5 },  
  { "REPLACE ", 3, 6 } ) ;
```

Figure 4-3. Example Rule Map for Verb CSUAACI

The first four elements describe the keywords for which only one must be present. The order_no for each of these is the same; a value of 1. Thus, the first element of output array pRule_value is set when any of these keywords are found in the rule array. The value for map_value is the value that goes into that element of the output array. Thus, if the rule array contains CHGEXPDT, the first element of the output array is set to 2. If more than one of these four keywords is in the rule array, the return code variable pReturn_message is set to E_RULE_ARRAY_COMBINE.

The last two elements, for PROTECTD and REPLACE, describe optional keywords. Any combination of these two is valid - neither, one, or both can be in the rule array. Thus, we treat these independently from any other keywords. They are assigned, respectively, to elements 2 and 3 of the output array, and the values to be stored there are 5 if PROTECTD is present, and 6 if REPLACE is present.
**saf_process_key_label - Process Key Label**

**Note:** This function is available on the workstation host. However, the z/OS library is incompatible with the workstation key storage files, so it is of limited utility in a z/OS build.

`saf_process_key_label` accepts a key label, verifies that it has no errors, and returns an updated label processed according to the following steps.

1. Initializes the output field to all blanks.
2. Makes the label all uppercase.
3. Verifies that the first character is either a letter or a national character.
4. Allows only letters and national characters for the first character.
5. Validates each name token and checks for wildcards.
6. Removes the name token delimiter (".").
7. Left justifies and copies each name token into an 8-byte output area.

The function result is set to TRUE if the input key label is valid, and to FALSE if it is not.

**Function Prototype**

```c
boolean saf_process_key_label ( unsigned char *pKeyLabel, boolean wildcard_flag, unsigned char *pLabelOut)
```

**Input**

On entry to this routine:

- `pKeyLabel` is a pointer to the input key label.
- `wildcard_flag` is a boolean value of TRUE if the input key label can have wildcard characters, and FALSE if they are not allowed.

**Output**

On successful exit from this routine:

- `pLabelOut` is the process key label produced by the function, if `pKeyLabel` was a valid label.
- `saf_process_key_label` returns a boolean value indicating TRUE if the input key label is valid, and FALSE otherwise.

**Return and Reason Codes**

This function has no return codes.
Chapter 5.

Function Control Vector Management Functions

Note: All functions within this chapter are available only on the coprocessor.

This section describes functions used to interact with the function control vector (FCV) in the coprocessor. The FCV contains information describing what operations are permitted on this coprocessor, based on the export regulations governing the coprocessor’s location and the business of its owner.

The FCV is loaded into the coprocessor when the code is loaded on the mainframe. For testing purposes, you may load the FCV into a workstation coprocessor using the csulcnm utility. Refer to IBM 4764 PCI-X Cryptographic Coprocessor CCA Support Program Installation Manual for more detailed information.

Header Files for Function Control Vector Management Functions
When using these functions, your program must include the following header files.

```c
#include "cmncryt2.h" /* Crypto ESSS definition */
#include "cam_fcv.h" /* Function control vector */
```

Summary of Functions
Functions that interact with FCV include the following:

- `getSymmetricMaxModulusLength` Gets the maximum RSA key length.
- `isFunctionEnabled` Determines whether the FCV allows a particular function.
**getSymmetricMaxModulusLength - Get RSA Key Length**

*Note:* This function is available on the coprocessor.

getSymmetricMaxModulusLength returns the maximum RSA key modulus length (in bits) that can be used for encrypting symmetric algorithm encryption keys.

**Function Prototype**

```c
long getSymmetricMaxModulusLength( unsigned short *pModLength)
```

**Input**

`pModLength` is a pointer to an unsigned short variable.

**Output**

On successful exit from this routine:

`pModLength` contains the modulus maximum length.

**Return Codes**

Common return codes generated by this routine are:

- **OK**: The operation was successful.
- **srdi_ALLOC_ERROR**: Out of memory to open the FCV.
- **srdi_READ_ERROR**: Error reading the FCV.
- **srdi_GENERAL_ERROR**: Could not read the FCV.
- **srdi_NOT_FOUND**: The FCV was not found.
**isFunctionEnabled - Check Whether a Function is Enabled**

*Note:* This function is available on the coprocessor.

isFunctionEnabled returns a boolean value indicating whether the specified function is enabled or disabled in the Function Control Vector. This is used to determine whether a function is permitted under the export rules governing this particular coprocessor.

The function result is TRUE if the specified function is enabled, and FALSE if it is disabled.

**Function Prototype**

```c
boolean isFunctionEnabled( long FunctionByteIndex,
                           unsigned char FunctionBitSelect)
```

**Input**

On entry to this routine:

- **FunctionByteIndex** is an index into the Function Control Vector, giving the location of the byte to be checked. See Figure 5-1 for a list of possible values.

- **FunctionBitSelect** is the bit to be checked in the specified Function Control Vector byte. See Figure 5-1 for a list of possible values.

**Output**

On successful exit from this routine:

- **isFunctionEnabled** returns a boolean value indicating whether the specified function is enabled or disabled in the Function Control Vector.

**Notes**

The following figure shows how the byte or bit corresponds to a particular function.

*Figure 5-1. Possible Values*

<table>
<thead>
<tr>
<th>Function Byte Name</th>
<th>Function Bit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCA_BASE_FUNCTION_BYTE</td>
<td>FCV_CCA_BASE</td>
<td>Byte index of the CCA base services bits.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base CCA services-enabled bit.</td>
</tr>
<tr>
<td>DES_FUNCTION_BYTE</td>
<td></td>
<td>Byte index of the DES-enabled bits.</td>
</tr>
<tr>
<td></td>
<td>FCV_56_BIT_DES</td>
<td>56-bit DES enabled bit.</td>
</tr>
<tr>
<td></td>
<td>FCV_TRIPLE_DES</td>
<td>Triple DES enabled bit.</td>
</tr>
<tr>
<td>SET_FUNCTION_BYTE</td>
<td></td>
<td>Byte index of the bits that are SET enabled.</td>
</tr>
<tr>
<td></td>
<td>FCV_SET_SERVICES</td>
<td>SET services enabled bits.</td>
</tr>
</tbody>
</table>

**Return Codes**

This function has no return codes.
Examples
To determine whether SET functions for encoding and decoding are enabled in the coprocessor:

```c
if ( ! isFunctionEnabled(SET_FUNCTION_BYTE, FCV_SET_SERVICES) )
{
/* cancel this section, SET functions are not allowed. */
}
```

To see if 56-bit DES encryption is allowed:

```c
if ( isFunctionEnabled(DES_FUNCTION_BYTE, FCV_56_BIT_DES) )
{
/* use 56-bit DES encryption */
}
```
Chapter 6. CCA Master Key Manager Functions

The CCA Master Key Manager provides access to the CCA master key registers on the PCI-X cryptographic coprocessor, as required by the CCA application. The CCA command processors never access the master keys directly, and in fact they have no need to know how or where the master keys and related information are stored. The Master Key Manager provides a set of functions to load the key values, and to use the keys to encipher and decipher data. It can be viewed as an object, with internal data, and with methods that can be used to operate on and with that data.

Since the master key storage mechanism is hidden from master key users, that mechanism can be changed without affecting any command processors that make use of the master keys. In the coprocessor, the master key data is stored in flash EPROM.

Note: All functions within this chapter are available only on the coprocessor. Certain functions are available only on a coprocessor in a workstation, not on a coprocessor running in a z/OS system.

Header Files for Master Key Manager Functions

When using these functions, your program must include the following header files.

```c
#include "cmncrypt.h"    /* Cryptographic definitions */
#include "cam_xtrn.h"     /* SRDI manager definitions */
```

Overview of the CCA Master Keys

The coprocessor uses triple-length master keys, each consisting of three independent eight-byte DES keys. The master keys are used to protect other data in the following two ways.

- Single-length (eight-byte) keys are protected using EDE encryption, with three independent keys. To encrypt an eight-byte key \( K \) with master key \( M \), the process is as follows:

  1) Encrypt \( K \) using part 1 of key \( M \).
  2) Decrypt the result of step 1 using part 2 of key \( M \).
  3) Encrypt the result of step 2 using part 3 of key \( M \)

- Data longer than eight bytes, such as PKA key components, is encrypted using the EDE3 triple encryption algorithm.

z/OS supports 16 cryptographic domains. There is a separate set of master keys for each domain. The \( mk_set \) parameter of the \( mk_selectors \) variable type identifies the set of master keys to be processed by a function. There is only one set of master keys on a workstation.

Within each master key set, CCA supports two types of master keys, PKA keys (\( ASYM\_MK \)) and DES keys (\( SYM\_MK \)). Each type of key consists of three master key values.

- Old Master Key (OMK)—The version of the master key that was in use prior to the current value. It is maintained to permit recovery of keys that were enciphered under the old master key.
- Current Master Key (CMK)—The current, operational master key. All keys in use in the system are enciphered under this key.
• New Master Key (NMK)—A new master key, which is being entered into the system to replace the current master key. It may be entered in the form of one or more key parts which are combined to form the final key, as a randomly generated value, or as a set of key shares, which were generated on a different coprocessor.

Each of the three master keys is stored in a logical register within the Master Key Manager. In addition, the Master Key Manager holds data associated with each of these key values.

• A Verification Pattern is stored for each of the three keys. The verification pattern is a 20-byte value which is calculated using a strong one-way function on the key value. This value can be used to verify that the key value matches another key, or the key originally used in some process. The verification pattern can be public, without endangering the value of the key itself.

For the SYM_MK master key, if the first and third key parts are the same, this value is calculated using the zOS ICSF algorithm. Otherwise, this value is calculated using SHA-1. For the ASYM_MK master key, this value is calculated using MDC-4.

• The status of the key. For the CMK and the OMK, two status values are possible.
  - The register contains a valid key value.
  - The register does not contain a valid key value.

For the NMK register, three status values are possible.
  - The register is empty. It does not contain any portion of a new master key value.
  - The register is partially full. The last key part has not yet been combined into the value in the register.
  - The NMK register is full. All key parts have been combined to form the final key value.

The verification pattern and the status can be read from the Master Key Manager using its interface functions. The values of the keys themselves can never be read.

**Location of the Master Keys**

The master keys and their associated data are Security Relevant Data Items (SRDIs). Their secure storage and retrieval are handled through use of the SRDI Manager, and its API functions.

Each SRDI has an eight character name. The master key data SRDI for DES keys is named MSTRKEYS. The master key data SRDI for asymmetric keys is ASYMKEYS.

**Initialization of the Master Key SRDI**

When the CCA application is first loaded into a new coprocessor, no master key SRDI exists in the flash EPROM. The Master Key Manager includes an initialization function init_master_keys(), which creates and initializes this SRDI the first time it is called. The SRDI is initialized with the following values.

• The three master key registers, NMK, CMK, and OMK, are all set to binary zeroes.
• The state of CMK and OMK is set to invalid. The state of NMK is set to Empty.
• The master key verification patterns are set to binary zeroes.
CCA Master Key Manager Interface Functions

The following sections describe the functions that comprise the Master Key Manager interface. CCA command processors use these functions to manage master key values, and to encipher or decipher data using the master keys.

Each of these functions returns an error code as the function result.

Common Entry Processing

A portion of the processing is common to all of the Master Key Manager interface functions. This code is in a common function, which is called by each of the API functions listed as follows.

The common entry processing performs the following functions.

1. If the Master Key Manager has already opened the Master Key SRDI, then error code mk_NO_ERROR is returned to the caller. Otherwise, continue with step 2.

2. Open the Master Key SRDI, either MSTRKEYS or ASYMKMKS, depending on input to the function. If no error occurs opening the SRDI, then error code mk_NO_ERROR is returned to the caller. Otherwise, error code mk_SRDI_OPEN_ERROR is returned.

Required Variables

In order to specify which master key register is to be used, many of the master key functions require a variable of type mk_selectors. This variable has three parameters:

- The master key set (mk_set) that specifies which set of master keys is to be accessed, for environments where more than one set of master keys may exist. Where there is only one set of master keys, mk_set must be set to MK_SET_DEFAULT. Otherwise, mk_set must be set to the Domain field passed in the CPRB structure (for example, mkSelector.mk_set = pCprbIn->Domain).

- The master key register (mk_register) within the specified master key set. This can be any of the defined values old_mk, current_mk, or new_mk, representing the old master key, the current master key, and the new master key.

- The master key type (type_mks) which defines the type of key to be encrypted with this set of master keys. This variable can be any of ASYM_MK, SYM_MK, or BOTH_MK.

The following functions are summarized in this chapter. Functions available on a coprocessor with the zOS code loaded are marked with a “z”. Other functions are only available on a coprocessor with the workstation load.

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear_master_keys</td>
<td>6-6</td>
</tr>
<tr>
<td>combine_mk_parts</td>
<td>6-7</td>
</tr>
<tr>
<td>compute_mk_verification_pattern</td>
<td>6-19 z</td>
</tr>
<tr>
<td>ede3_triple_decrypt_under_master_key</td>
<td>6-26 z</td>
</tr>
<tr>
<td>ede3_triple_encrypt_under_master_key</td>
<td>6-27 z</td>
</tr>
<tr>
<td>generate_mk_shares</td>
<td>6-9</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>generate_random_mk</td>
<td>6-11</td>
</tr>
<tr>
<td>get_master_key_status</td>
<td>6-23</td>
</tr>
<tr>
<td>get_mk_verification_pattern</td>
<td>6-21</td>
</tr>
<tr>
<td>init_master_keys</td>
<td>6-12</td>
</tr>
<tr>
<td>load_first_mk_part</td>
<td>6-13</td>
</tr>
<tr>
<td>load_mk_from_shares</td>
<td>6-14</td>
</tr>
<tr>
<td>mkmCombineMKParts</td>
<td>6-7</td>
</tr>
<tr>
<td>mkmGenerateMKShares</td>
<td>6-9</td>
</tr>
<tr>
<td>mkmGenerateRandomMK</td>
<td>6-11</td>
</tr>
<tr>
<td>mkmGetAsymVerificationPattern</td>
<td>6-22</td>
</tr>
<tr>
<td>mkmGetMasterKeyStatus</td>
<td>6-23</td>
</tr>
<tr>
<td>mkmLoadFirstMKPart</td>
<td>6-13</td>
</tr>
<tr>
<td>mkmLoadMKFromShares</td>
<td>6-14</td>
</tr>
<tr>
<td>mkmSetMasterKey</td>
<td>6-16</td>
</tr>
<tr>
<td>reinit_master_keys</td>
<td>6-17</td>
</tr>
<tr>
<td>mkmTDESDecryptUnderMasterKey</td>
<td>6-28</td>
</tr>
<tr>
<td>mkmTDESEncryptUnderMasterKey</td>
<td>6-29</td>
</tr>
<tr>
<td>triple_decrypt_under_master_key</td>
<td>6-30</td>
</tr>
<tr>
<td>triple_decrypt_under_master_key_with.CV</td>
<td>6-31</td>
</tr>
<tr>
<td>triple_encrypt_under_master_key</td>
<td>6-32</td>
</tr>
<tr>
<td>triple_encrypt_under_master_key_with.CV</td>
<td>6-33</td>
</tr>
</tbody>
</table>
Functions to Set and Manage the Master Key Values

Note: These functions are not available on a coprocessor with the z/OS load.

The following functions are used to load, clear, or initialize master key registers. Other functions in this category are used in other ways related to generation and distribution of master keys.

Summary of Functions

- **clear_master_keys** Clears a specified master key register. See page 6-6.
- **combine_mk_parts** Combines an additional master key part into the value already in the New Master Key register. See page 6-7.
- **generate_mk_shares** Splits a 24-byte master key into shares. See page 6-9.
- **generate_random_mk** Generates a random 24-byte master key. See page 6-11.
- **init_master_keys** Creates and initializes the master key SRDI. See page 6-12.
- **load_first_mk_part** Loads the first part of a multi-part master key into the new master key register. See page 6-13.
- **load_mk_from_shares** Reconstructs the shares that were produced using generate_mk_shares and loads the master key into the new master key register. See page 6-14.
- **mkmCombineMKParts** Combines an additional master key part into the value already in the New Master Key register. See page 6-7.
- **mkmGenerateMKShares** Splits a 24-byte master key into shares. See page 6-9.
- **mkmGenerateRandomMK** Generates a random 24-byte master key. See page 6-11.
- **mkmLoadFirstMKPart** Loads the first part of a multi-part master key into the new master key register. See page 6-13.
- **mkmLoadMKFromShares** Reconstructs the shares that were produced using generate_mk_shares and loads the master key into the new master key register. See page 6-14.
- **mkmSetMasterKey** Activate the master key. See page 6-16.
- **reinit_master_keys** Deletes all master key data and then creates and initializes the master key SRDI. See page 6-17.
- **set_master_key** Activates the master key. See page 6-16.
clear_master_keys - Clear Master Key

clear_master_keys clears a specified master key register. All bytes of the specified register are set to a value of X'00', and the state of the register is set to mks_NMK_EMPTY for NMK, or mks_CMK_INVALID/mks_OMK_INVALID for CMK or OMK.

Three separate calls for each type of master key are required in order to clear all of the master key registers.

Function Prototype

long clear_master_keys( mk_selectors MKSelector) ;

Input

On entry to this routine:

MKSelector is a pointer to a variable which must be initialized as follows:

• mk_set is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.

• mk_register is set to either old_mk, current_mk, or new_mk, representing the key which should be cleared.

• type_mks should be set to ASYM_MK if this set of master keys is intended for PKA key encryption or SYM_MK if this set of master keys is used for DES key encryption. If type_mks is BOTH_MK and the state of the specified master key register of both the master key sets is the same, the specified master key register will be cleared and the state set to empty for both master key registers; otherwise an error message will be returned.

Output

This function returns no output. On successful exit from this routine:

The specified master key register has been cleared.

Return Codes

Common return codes generated by this routine are:

mk_NO_ERROR The operation was successful.

mk_SRDI_OPEN_ERROR Could not open the master key SRDI.

mk_INVALID_KEY_SELECTOR The input parameters are not valid.

mk_SEM_CLAIM_FAILED Could not access the master key SRDI.
**mkmCombineMKParts - Combine Master Key Parts**

mkmCombineMKParts combines an additional master key part into the value already in the NMK register for the set of master keys being processed. The NMK register must be in the Partially Full state when this function is called; otherwise, an error is returned. The key part is designated as the final part, if the key is complete after this part has been combined into the register.

Return codes are used to notify the caller if the combined key value has bad parity, or if it has equal left and right halves. These are informative return codes, and are not considered errors by the Master Key Manager.

**Note:** The purpose behind requiring a load_first_mk_part call separately from combine_mk_parts is to enforce security. Different roles may be required for each of these functions, ensuring that no one person has input all of the parts of the master key.

### Function Prototype

```c
long mkmCombineMKParts( TRIPLE_LENGTH_KEY *key_part, 
                        mk_selectors MKSelector, 
                        boolean final_part) ;
long combine_mk_parts ( TRIPLE_LENGTH_KEY *key_part, 
                        boolean final_part) ;
```

The combine_mk_parts function is the equivalent of calling mkmCombineMKParts with the MKSelector variable set to {MK_SET_DEFAULT, new_mk, SYM_MK}.

### Input

On entry to this routine:

- **key_part** is a 24-byte cleartext key part, which is to be combined into the value in the NMK register.
- **MKSelector** is a parameter of type mk_selectors, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - **mk_set** is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
  - **mk_register** must be set to new_mk.
  - **type_mks** should be set to ASYM_MK if this set of master keys is intended for PKA key encryption or SYM_MK if this set of master keys is used for DES key encryption. If type_mks is BOTH_MK and the state and the value of the new master key register of both the master keysets is the same, the key part will be combined into both new master key registers; otherwise an error message will be returned.
- **final_part** is a boolean value, which the caller sets to TRUE when the key part is the final part of a new master key.

The NMK register must have been initialized with a call to mkmLoadFirstMKPart and zero or more calls to mkmCombineMKParts.

### Output

This function returns no output. On successful exit from this routine:

The value in **key_part** has been combined into the value already in the NMK register. If **final_part** was TRUE, the NMK register is left in a Full state. Otherwise, the state of NMK is Partially Full.
**Return Codes**

Common return codes generated by this routine are:

- **mk_NO_ERROR**  The operation was successful.
- **mk_SRDI_OPEN_ERROR**  Could not open the master key SRDI.
- **mk_SEM_CLAIM_FAILED**  Could not access the SRDI Manager, the operation cannot be completed.
- **mk_INCORRECT_STATE**  The New Master Key register was not Partially Full.
- **mk_WEAK_KEY**  final_part was TRUE and the resulting key was a weak key. The prior state has been restored.
**mkmGenerateMKShares - Generate Master Key Shares**

mkmGenerateMKShares splits a 24-byte master key into shares for the `mk_set` being processed. The key is split into \( n \) separate shares, where any \( m \) of the shares can be used to recreate the master key value at a later time.

The shares are distributed to separate individuals for safekeeping. When the coprocessor has to be initialized with the master key, any \( m \) of these individuals must present their master key shares in order to create the complete key in the coprocessor.

The source of the master key is specified with the `KeySource` parameter.

**Function Prototype**

```c
long mkmGenerateMKShares( mk_selectors MKSelector, 
mk_src_t KeySource, 
UCHAR mShareKey, 
UCHAR nShareGen, 
UCHAR *pShares [] );
```

The `generate_mk_shares` function is the equivalent of calling the `mkmGenerateMKShares` with the `MKSelector` parameter set to `{ MK_SET_DEFAULT, current_mk, SYM_MK }`.

**Input**

On entry to this routine:

`MKSelector` is a parameter of type `mk_selectors`, indicating which set of master keys to use in this function. This variable must be initialized as follows:

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
- `mk_register` must be set to `new_mk`, `old_mk`, or `current_mk`.
- `type_mks` should be set to `ASYM_MK` if this set of master keys is intended for PKA key encryption or `SYM_MK` if this set of master keys is used for DES key encryption. If `type_mks` is `BOTH_MK` and the key source is the new, current, or old master key register, and the state and value of the specified master key register is the same for both the master key sets, the shares will be generated from the specified register; otherwise an error message will be returned.
KeySource is a value which specifies the source for the master key value that is split. The possible values are:

<table>
<thead>
<tr>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Randomly generated value (src_random)</td>
<td>The function generates a new, random key value, and splits it into the specified number of shares. The value is discarded after the shares have been returned.</td>
</tr>
<tr>
<td>New Master Key (NMK) Register (src_nmk)</td>
<td>The value in the NMK register is split into shares. An error is returned if the NMK state is not Full.</td>
</tr>
<tr>
<td>Current Master Key (CMK) Register (src_cmk)</td>
<td>The value in the CMK register is split into shares. An error is returned if the register does not contain a valid value.</td>
</tr>
<tr>
<td>Old Master Key (OMK) Register (src_omk)</td>
<td>The value in the OMK register is split into shares. An error is returned if the register does not contain a valid value.</td>
</tr>
</tbody>
</table>

mShareKey is the number of shares that are required in order to reconstruct the master key value. This is the number of shares that must be given to the function mkmLoadMKFromShares in order to load the key.

nShareGen is the total number of shares to generate and return. Any m of these shares can be used to reconstruct the key.

pShares is a pointer to an area which is large enough to store n key shares, each of which is 25 bytes in length.

Output
On successful exit from this routine:

*pShares[ ] stores the generated key shares. This area must be large enough to hold n key shares, each of which is 25 bytes in length.

Return Codes
Common return codes generated by this routine are:

- **mk_NO_ERROR** The operation was successful.
- **mk_SRDI_OPEN_ERROR** Could not open the master key SRDI.
- **mk_SEM_CLAIM_FAILED** Could not access the SRDI Manager, the operation cannot be completed.
- **mk_INCORRECT_STATE** The master key is not valid.
- **mk_UNSUPPORTED_SCHEME** The values of mShareKey and nShareGen were inconsistent.
- **mk_KEY_SHARE_SPLIT_FAIL** An error occurred in the splitting process.
**mkmGenerateRandomMK - Generate Random Master Key**
mkmGenerateRandomMK generates a random 24-byte master key, and stores the value in the NMK register for the mk_set being processed.

The NMK register must be in the empty state when this function is called; otherwise, an error is returned. If the function completes successfully, the NMK register is left in the Full state.

**Function Prototype**

```c
long mkmGenerateRandomMK( mk_selectors MKSelector) ;
long generate_random_mk ( void) ;
```

The `generate_random_mk` function is the equivalent of calling `mkmGenerateRandomMK` with the MKSelector parameter set to `{MK_SET_DEFAULT, new_mk, SYM_MK}`.

**Input**

On entry to this routine:

MKSelector is a parameter of type `mk_selectors`, indicating which set of master keys to use in this function. This variable must be initialized as follows:

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- `mk_register` must be set to `new_mk`.
- `type_mks` should be set to ASYM_MK if this set of master keys is intended for PKA key encryption or SYM_MK if this set of master keys is used for DES key encryption. If `type_mks` is BOTH_MK and the new master key register of both the master key sets is empty, a random master key will be generated and loaded into both new master key registers; otherwise an error message will be returned.

**Output**

This function returns no output. On successful exit from this routine:

The new master key is generated and stored in the NMK register. In order to use this master key, `set_master_key` must be called.

**Return Codes**

Common return codes generated by this routine are:

- **mk_NO_ERROR** The operation was successful.
- **mk_SRDI_OPEN_ERROR** Could not open the master key SRDI.
- **mk_SEM_CLAIM_FAILED** Could not access the SRDI Manager, the operation cannot be completed.
- **mk_MKS_MISMATCH** BOTH_MK was the value in `type_mks`, but one or both of the new master key registers was not in the empty state.
- **mk_INCORRECT_STATE** The master key is in the incorrect state.
**init_master_keys - Create and Initialize Master Keys**

init_master_keys creates and initializes the master key SRDI, if it doesn't already exist.

**Function Prototype**

long init_master_keys( void) ;

**Input**

This function has no input.

**Output**

This function returns no output. On successful exit from this routine:

The master key SRDI is generated and initialized.

**Return Codes**

Common return codes generated by this routine are:

- **mk_NO_ERROR**
  The operation was successful.

- **srdi_EXISTS**
  The master keys already exist, and cannot be initialized.

- **srdi_GENERAL_ERROR**
  Failed to access the SRDI manager.

- **srdi_ALLOC_ERROR**
  Could not allocate memory for the master keys.
**mkmLoadFirstMKPart - Load First Master Key Part**

mkmLoadFirstMKPart loads the first part of a multi-part cleartext master key into the new master key (NMK) register for the mk_set being processed.

**Note:** The purpose behind requiring mkLoadFirstMKPart separately from mkmCombineMKParts is to enforce security. Different roles may be required for each of these functions, ensuring that no one person has input all of the parts of the master key.

**Function Prototype**

```c
long mkmLoadFirstMKPart( TRIPLE_LENGTH_KEY *key_part,
                        mk_selectors MKSelector);
```

```c
long load_first_mk_part( TRIPLE_LENGTH_KEY *key_part);
```

The load_first_mk_part function is the equivalent of calling mkmLoadFirstMKPart with the MKSelector parameter set to {MK_SET_DEFAULT, current_mk, SYM_MK}.

**Input**

On entry to this routine:

- `key_part` is a 24-byte cleartext key part, which is to be stored in the NMK register.

- `MKSelector` is a parameter of type `mk_selectors`, indicating which set of master keys to use in this function. This variable must be initialized as follows:

  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
  - `mk_register` should be set to new_mk.
  - `type_mks` should be set to ASYM_MK if this set of master keys is intended for PKA key encryption or SYM_MK if this set of master keys is used for DES key encryption. If `type_mks` is BOTH_MK and the new master key register of both the master key sets is empty, the key part will be loaded into both new master key registers; otherwise an error message will be returned.

**Output**

This function returns no output. On successful exit from this routine:

- The value in `key_part` has been copied into the new master key register.

**Return Codes**

Common return codes generated by this routine are:

- **mk_NO_ERROR** The operation was successful.
- **MK_BAD_KEY_PARITY** The input parity has been adjusted to odd parity.
**mkmLoadMKFromShares - Load Master Key Shares**

mkmLoadMKFromShares loads a master key into the New Master Key register, reconstructing the key from m supplied shares, which were originally produced by the `mkmGenerateMKShares` function for the `mk_set` being processed.

The shares are distributed to separate individuals for safekeeping. When the coprocessor has to be initialized with the master key, any m of these individuals must present their master key shares to create the complete key in the coprocessor.

The New Master Key (NMK) register must be in the empty state when this function is called. It is in the Full state if the function completes successfully. The key value is left in the NMK register; you must use the `mkmSetMasterKey` function to make it the current master key.

**Function Prototype**

```c
long mkmLoadMKFromShares( mk_selectors  MKSelector,
                           UCHAR  mShareKey,
                           UCHAR  nShareGen,
                           UCHAR *pShares [] );
long load_mk_from_shares(    UCHAR  mShareKey,
                           UCHAR  nShareGen,
                           UCHAR *pShares [] );
```

The `load_mk_from_shares` function is the equivalent of calling `mkmLoadMKFromShares` with the `MKSelector` parameter set to `{ MK_SET_DEFAULT, current_mk, SYM_MK }`.

**Input**

On entry to this routine:

- `MKSelector` is a parameter of type `mk_selectors`, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` should be set to `new_mk`.
  - `type_mks` should be set to `ASYM_MK` if this set of master keys is intended for PKA key encryption or `SYM_MK` if this set of master keys is used for DES key encryption. If `type_mks` is BOTH_MK and the new master key register of both the master key sets is empty, both new master key registers will be loaded with the master key value; otherwise an error message will be returned.

- `mShareKey` is the number of shares required to reconstruct the master key. This is the number of shares that are provided in the `pShares[ ]` array.

- `nShareGen` is the total number of shares that were generated for this key, by the `mkmGenerateMKShares` function.

- `*pShares[ ]` is an array of pointers to the m shares that are used to reconstruct the master key. Each share is 25 bytes in length.

**Output**

This function has no output. On successful exit from this routine:

The new master key is generated in the NMK register and the state of the NMK register is Full.
Return Codes

Common return codes generated by this routine are:

- **mk_NO_ERROR**
  - The operation was successful.

- **mk_SRDI_OPEN_ERROR**
  - Unable to open the SRDI item.

- **mk_SEM_CLAIM_FAILED**
  - Unable to access the SRDI Manager.

- **mk_INCORRECT_STATE**
  - The NMK is not in the Empty state.

- **mk_UNSUPPORTED_SCHEME**
  - mShareKey or nShareGen have invalid values.

- **mk_KEY_SHARE_RECOVER_FAIL**
  - Unable to recover the master key.

- **mk_VP_CALCULATE_FAIL**
  - SHA calculation error.

- **mk_SAVE_ERROR**
  - An SRDI error occurred while attempting to save the new master key in flash memory.

- **mk_VP_MATCHES_EXISTING_KEY**
  - Verification patterns match one of the existing master keys.

- **mk_EQUAL_KEY_HALFS**
  - Two of the three parts of the new key are equal. This is a warning, no action is required.

- **mk_WEAK_KEY**
  - One of the key parts is a weak key. The key should be regenerated before use.
**mkmSetMasterKey - Set Master Key**

mkmSetMasterKey activates the master key which has been accumulated in the NMK register for the mk_set being processed. The key value is transferred to the Current Master Key (CMK) register. If a valid key is present in the CMK register, it is transferred to the Old Master Key (OMK) register. The key verification patterns are transferred from CMK to OMK, and from NMK to CMK.

**Function Prototype**

```c
long mkmSetMasterKey( mk_selectors pMKSelector) ;
long set_master_key ( void) ;
```

The `set_master_key` function is the equivalent of calling `mkmSetMasterKey` with the `MKSelector` parameter set to `{MK_SET_DEFAULT, current_mk, SYM_MK}`.

**Input**

On entry to this routine:

`MKSelector` is a parameter of type `mk_selectors`, indicating which set of master keys to use in this function. This variable must be initialized as follows:

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
- `mk_register` is ignored.
- `type_mks` should be set to `ASYM_MK` if this set of master keys is intended for PKA key encryption or `SYM_MK` if this set of master keys is used for DES key encryption. If `type_mks` is `BOTH_MK` and the new master key register of both the master key sets is full and the value in the new master key register of both the new master key sets is the same, then the `current_mk` registers will be set to the `new_mk` value; otherwise an error message will be returned.

**Output**

This function returns no output. On successful exit from this routine:

The master keys have been changed.

**Return Codes**

Common return codes generated by this routine are:

- `mk_NO_ERROR` The operation was successful.
- `mk_SRDI_OPEN_ERROR` Unable to open the SRDI item.
- `mk_SEM_CLAIM_FAILED` Unable to access the SRDI Manager.
- `mk_INCORRECT_STATE` The NMK register is not in the Full state.
- `mk_SAVE_ERROR` Unable to save the master key into flash memory.
**reinit_master_keys - Reinitialize Master Keys**

reinit_master_keys deletes all master key data, then recreates and initializes the master key SRDI to the default state.

The function returns TRUE if the operation completed successfully, and FALSE if it did not.

**Note:** This function erases master key data. Once this function is complete, all operational keys which have been encrypted under any master key are unusable.

**Function Prototype**

```c
long reinit_master_keys( void) ;
```

**Input**

This function has no input.

**Output**

This function returns no output. On successful exit from this routine:

All master keys are created and initialized.

**Return Codes**

Common return codes generated by this routine are:

- `srdi_NO_ERROR` The operation was successful.
- `srdi_GENERAL_ERROR` Unable to access the master key SRDI.
Functions to Check Master Key Values and Status

Note: These functions are available on all coprocessors.

Summary of Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute_mk_verification_pattern</td>
<td>Computes a key verification pattern. See page 6-19.</td>
</tr>
<tr>
<td>get_mk_verification_pattern</td>
<td>Returns the 20-byte master key verification pattern for a specified master key. See page 6-21.</td>
</tr>
<tr>
<td>get_master_key_status</td>
<td>Get the status of the three master key registers for the SYM_MK master key. See page 6-23.</td>
</tr>
<tr>
<td>mkmGetAsymVerificationPattern</td>
<td>Returns the verification pattern for asymmetric master keys. See page 6-22.</td>
</tr>
<tr>
<td>mkmGetMasterKeyStatus</td>
<td>Returns the status of the three master key registers for the mk_set being processed. See page 6-23.</td>
</tr>
</tbody>
</table>
**compute_mk_verification_pattern - Compute a Verification Pattern**

compute_mk_verification_pattern computes a key verification pattern for the contents of the specified master key register. The verification pattern can be used to determine which master key (old or current) was used to encrypt a given operational key. The returned verification pattern is 20 bytes in length. For the symmetric-keys master key (SYM_MK), if the first and third key parts of the register specified are the same, the OS/390 ICSF algorithm for the generation of the master key verification pattern will be used. If the key parts are not the same, the verification pattern is a SHA-1 hash of the key, combined with a header. The header is a one-byte value which is used to differentiate this hash from any other hash on the master key, which might be computed for a different purpose. The value of the header byte is X'01'. For the asymmetric-keys master key (ASYM_MK), the MDC-4 algorithm will be used to compute the verification pattern.

**Function Prototype**

```c
long compute_mk_verification_pattern( UCHAR *ver_pattern,
                                  mk_selectors *mk_selector,
                                  VP_ALG_TYPE  vp_algorithm);
```

**Input**

On entry to this routine:

- `ver_pattern` is a pointer to a 20-byte location where the computed key verification pattern is returned.
- `mk_selector` is a pointer to a variable which must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is set to either `old_mk`, `current_mk`, or `new_mk`, representing the key whose verification pattern is desired.
  - `type_mks` should be set to `ASYM_MK` if this set of master keys is intended for PKA key encryption or `SYM_MK` if this set of master keys is used for DES key encryption.

If `type_mks` is `BOTH_MK` and the state and value of the specified master key register of both the master key sets is the same, the verification pattern of the specified register will be calculated on the Symmetric Keys master key set; otherwise, an error message will be returned.

- `vp_algorithm` must be one of `VP_ALG_DEFAULT`, `VP_ALG_MDC4`, `VP_ALG_SHA1`.

Note that the default for PKA tokens is `VP_ALG_MDC4` (for a 16-byte MDC-4 hash), the default for DES tokens is `VP_ALG_SHA1`.

**Output**

On successful exit from this routine:

- `ver_pattern` contains the verification pattern.
Return Codes
Common return codes generated by this routine are:

- **mk_NO_ERROR** The operation was successful.
- **mk_SRDI_OPEN_ERROR** Unable to open the SRDI item.
- **mk_SEMCLAIM_FAILED** Unable to access the SRDI Manager.
- **mk_VP_CALCULATE_FAIL** SHA calculation error.
**get_mk_verification_pattern - Read a Verification Pattern**

get_mk_verification_pattern returns the pre-computed 20-byte master key verification pattern (MKVP) for a specified master key. This value is computed and saved when the master key is first loaded, and may be used to determine which of the master keys was used to encrypt a given operational key.

**Function Prototype**

```c
long get_mk_verification_pattern( UCHAR *ver_pattern,
                                mk_selectors *mk_selector );
```

**Input**

On entry to this routine:

- `ver_pattern` is a pointer to a 20-byte location where the master key verification pattern is returned.
- `mk_selector` is a pointer to a variable which must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is set to either `old_mk`, `current_mk`, or `new_mk`, representing the key whose verification pattern is required.
  - `type_mks` should be set to `ASYM_MK` if the request is to get the verification pattern of the asymmetric-keys master key or to `SYM_MK` if the request is to get the verification pattern of the symmetric-keys master key.

**Output**

On successful exit from this routine:

- `ver_pattern` contains the verification pattern.

**Return Codes**

Common return codes generated by this routine are:

- `mk_NO_ERROR` The operation was successful.
- `mk_INVALID_KEY_SELECTOR` The input parameters are not valid.
- `mk_SEM_CLAIM_FAILED` Unable to access the SRDI Manager.
- `mk_KEY_NOT_VALID` The selected key was not in a valid state.
**mkmGetAsymVerificationPattern - Read the Asymmetric Keys Verification Pattern**
mkmGetAsymVerificationPattern returns the pre-computed verification pattern for a set of keys used to encrypt and decrypt PKA keys. This value is computed and saved when the master keys are first loaded, and may be used to determine which of the master keys was used to encrypt a given operational key.

**Function Prototype**

```c
long mkmGetAsymVerificationPattern( MKSelector,                      
        UCHAR * sha_1,                                           
        UCHAR * mdc_4 ) ;                                       
```

**Input**

On entry to this routine:

- **MKSelector** is a variable which must be initialized as follows:
  - **mk_set** is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to **MK_SET_DEFAULT**.
  - **mk_register** is set to either **old_mk**, **current_mk**, or **new_mk**, representing the key whose verification pattern is required.
  - **type_mks** is set to **ASYM_MK**.

- **sha_1** is a pointer to a 20-byte buffer where the SHA-1 hash verification pattern of the master key can be stored.

- **mdc_4** is a pointer to a 16-byte buffer where the MDC-4 hash verification pattern of the master key can be stored.

**Output**

On successful exit from this routine:

- **sha_1** contains the 20-byte verification pattern of the asymmetric master key requested.

- **mdc_4** contains the 16-byte MDC-4 verification pattern of the asymmetric master key requested.

**Return Codes**

Common return codes generated by this routine are:

- **mk_NO_ERROR** The operation was successful.
- **mk_INVALID_KEY_SELECTOR** The input parameters are not valid.
- **mk_KEY_NOT_VALID** The requested key was not in a valid state.
- **mk_SEM_CLAIM_FAILED** Unable to access the SRDI Manager.
**mkmGetMasterKeyStatus - Get Master Key Status**
mkmGetMasterKeyStatus returns the status of the three master key registers for the mk_set being processed. The results indicate whether the register holds a valid value, and whether a value in the NMK register is complete.

**Function Prototype**
```c
long mkmGetMasterKeyStatus( mk_selectors MKSelector,
                            mk_status_var *mk_status ) ;
long get_master_key_status( mk_status_var *mk_status ) ;
```

The `get_master_key_status` function is the equivalent of calling `mkmGetMasterKeyStatus` with the `MKSelector` parameter set to `{MK_SET_DEFAULT, current_mk, SYM_MK}`.

**Input**
On entry to this routine:

`MKSelector` is a parameter of type `mk_selectors`, indicating which set of master keys to use in this function. This variable must be initialized as follows:

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
- `mk_register` is ignored.
- `type_mks` should be set to `ASYM_MK` if the request is to get the status of the asymmetric-keys master key or to `SYM_MK` if the request is to get the status of the symmetric-keys master key.

**Output**
On successful exit from this routine:

`mk_status` contains the status of the 3 master key registers as a bitmapped value. Individual bits have the meanings defined in Figure 6-1.

**Figure 6-1. Master Key Status Bits**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (LSB)</td>
<td>NMK register is empty.</td>
</tr>
<tr>
<td>1</td>
<td>NMK register is partially full.</td>
</tr>
<tr>
<td>2</td>
<td>NMK register is full.</td>
</tr>
<tr>
<td>3</td>
<td>CMK register holds a valid value.</td>
</tr>
<tr>
<td>4</td>
<td>OMK register holds a valid value.</td>
</tr>
<tr>
<td>5-7</td>
<td>Reserved, set to 0.</td>
</tr>
</tbody>
</table>

`mk_status` returns a code indicating the success or failure of the operation.
Return Codes
Common return codes generated by this routine are:

- mk_NO_ERROR  The operation was successful.
- mk_SRDI_OPEN_ERROR  Unable to open the SRDI item.
- mk_SEM_CLAIM_FAILED  Unable to access the SRDI Manager.
Functions to Encrypt and Decrypt Using the Master Key

Note: These functions are available on all coprocessors.

Summary of Functions

ede3_triple_decrypt_under_master_key
   Triple decrypts multiple 8-byte data strings using EDE3 triple DES. See page 6-26.

ede3_triple_encrypt_under_master_key
   Triple encrypts multiple 8-byte data strings using EDE3 triple DES. See page 6-27.

TDESDecryptUnderMasterKey
   Triple-DES decrypts data using the master key. See page 6-28.

TDESEncryptUnderMasterKey
   Triple-DES encrypts data under the master key. See page 6-29.

triple_decrypt_under_master_key
   Triple-DES decrypts an 8-byte block of data. See page 6-30.

triple_decrypt_under_master_key_with_CV
   Triple-DES decrypts an 8-byte block of data using a control vector. See page 6-31.

triple_encrypt_under_master_key
   Triple-DES encrypts an 8-byte block of data. See page 6-32.

triple_encrypt_under_master_key_with_CV
   Triple-DES encrypts an 8-byte block of data using a control vector. See page 6-33.
ede3_triple_decrypt_under_master_key
ede3_triple_decrypt_under_master_key triple decrypts a string of data using EDE3 triple DES. The data length must be a multiple of eight bytes.

Function Prototype
long ede3_triple_decrypt_under_master_key( mk_selectors *mk_selector,
UCHAR *cleartext,
UCHAR *ciphertext,
ULONG  data_length);

Input
On entry to this routine:

mk_selector is a pointer to a variable which must be initialized as follows:

• mk_set is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.

• mk_register is set to either old_mk, current_mk, or new_mk, representing the key which should be used.

• type_mks should be set to ASYM_MK if the data is to be decrypted with the asymmetric-keys master key or to SYM_MK if the data is to be decrypted with the symmetric-keys master key.

cleartext is a pointer to a buffer large enough to store the cleartext. This may be the same as the ciphertext buffer.

ciphertext is a pointer to a buffer containing the data to be deciphered.

data_length is the number of bytes of data to be deciphered. This value must be a multiple of eight.

Output
On successful exit from this routine:

cleartext contains the deciphered data. This buffer may be the same as the ciphertext buffer, if desired.

Return Codes
Common return codes generated by this routine are:

mk_NO_ERROR The operation was successful.

mk_INVALID_DATA_LENGTH The data length is not a multiple of eight.

mk_SRDI_OPEN_ERROR Could not open the master key SRDI.

mk_SEM_CLAIM_FAILED Unable to access the SRDI Manager.

mk_KEY_NOT_VALID The designated master key is not valid.

mk_INVALID_KEY_SELECTOR The input parameters are not valid.
ede3_triple_encrypt_under_master_key
ede3_triple_encrypt_under_master_key triple encrypts a string of data using EDE3 triple DES. The data length must be a multiple of eight bytes.

Function Prototype
long ede3_triple_encrypt_under_master_key( mk_selectors *mk_selector, UCHAR *cleartext, UCHAR *ciphertext, ULONG data_length);

Input
On entry to this routine:

mk_selector is a pointer to a variable which must be initialized as follows:

• mk_set is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
• mk_register is set to either old_mk, current_mk, or new_mk, representing the key which should be used.
• type_mks should be set to ASYM_MK if the data is to be encrypted with the asymmetric-keys master key or to SYM_MK if the data is to be encrypted with the symmetric-keys master key.

cleartext is a pointer to a buffer containing the data to be enciphered.

ciphertext is a pointer to a buffer large enough to store the ciphertext. This buffer may be the same as the cleartext buffer.

data_length is the number of bytes of data to be enciphered. This value must be a multiple of eight.

Output
On successful exit from this routine:

ciphertext contains the enciphered data. This buffer may be the same as the cleartext buffer, if desired.

Return Codes
Common return codes generated by this routine are:

mk_NO_ERROR The operation was successful.
mk_INVALID_DATA_LENGTH The data length is not a multiple of eight.
mk_SRDI_OPEN_ERROR Could not open the master key SRDI.
mk_SEM_ClAIM_FAILED Unable to access the SRDI Manager.
mk_KEY_NOT_VALID The designated master key is not valid.
mk_INVALID_KEY_SELECTOR The input parameters are not valid.
**mkmTDESDecryptUnderMasterKey**
mkmTDESDecryptUnderMasterKey decrypts data which has been encrypted with a master key using the Triple DES algorithm. The master key which was used for encryption must be specified, and the cipher text provided must be a multiple of 8 bytes long.

**Function Prototype**

```c
long mkmTDESDecryptUnderMasterKey( mk_selectors MKSelector,
                                    UCHAR  *cipher_text,
                                    UCHAR  *clear_text,
                                    ULONG              text_length) ;
```

**Input**

On entry to this routine:

- **MKSelector** is a pointer to a variable which must be initialized as follows:
  - **mk_set** is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to **MK_SET_DEFAULT**.
  - **mk_register** is set to either **old_mk**, **current_mk**, or **new_mk**, representing the key which should be cleared.
  - **type_mks** should be set to **ASYM_MK** if the data is to be decrypted with the asymmetric-keys master key or to **SYM_MK** if the data is to be decrypted with the symmetric-keys master key.

- **cipher_text** is a pointer to a buffer which contains the encrypted text. This text must be a multiple of 8 bytes long.

- **clear_text** is a pointer to a buffer to hold the decrypted text. This buffer must be as long as **text_length**.

- **text_length** is the number of bytes of data in the **cipher_text** buffer. This is also the number of bytes of encrypted text returned in the **clear_text** buffer.

**Output**

On successful exit from this routine:

- **clear_text** contains **text_length** bytes of data decrypted from the **cipher_data** using the Triple DES algorithm.

**Return Codes**

Common return codes generated by this routine are:

- **mk_NO_ERROR** The operation was successful.
- **mk_INVALID_DATA_LENGTH** The data length is not a multiple of eight.
- **mk_SRDI_OPEN_ERROR** Could not open the master key SRDI.
- **mk_SEM_CLAIM_FAILED** Unable to access the SRDI Manager.
- **mk_KEY_NOT_VALID** The designated master key is not valid.
- **mk_INVALID_KEY_SELECTOR** The input parameters are not valid.
**mkmTDESEncryptUnderMasterKey**
mkmTDESEncryptUnderMasterKey takes a variable amount of data in a multiple of 8 bytes and encrypts it using the Triple DES algorithm.

**Function Prototype**

```c
long mkmTDESEncryptUnderMasterKey( mk_selectors   MKSelector,
                                   UCHAR        *clear_text,
                                   UCHAR        *cipher_text,
                                   ULONG        text_length);
```

**Input**

On entry to this routine:

- **MKSelector** is a pointer to a variable which must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is set to either `old_mk`, `current_mk`, or `new_mk`, representing the key which should be used.
  - `type_mks` should be set to `ASYM_MK` if the data is to be encrypted with the asymmetric-keys master key or to `SYM_MK` if the data is to be encrypted with the symmetric-keys master key.

- **clear_text** is a pointer to the text which you wish to encipher. This text must be a multiple of 8 bytes long.

- **cipher_text** is a pointer to a buffer in which to store the encrypted text. This buffer must be at least as long as the `clear_text`.

- **text_length** is the number of bytes of data in the `clear_text` buffer. This is also the number of bytes of encrypted text returned in the `cipher_text` buffer.

**Output**

On successful exit from this routine:

- **cipher_text** contains `text_length` bytes of data encrypted from the `clear_data` using the Triple DES algorithm.

**Return Codes**

Common return codes generated by this routine are:

- **mk_NO_ERROR** The operation was successful.
- **mk_INVALID_DATA_LENGTH** The data length is not a multiple of eight.
- **mk_SRDI_OPEN_ERROR** Could not open the master key SRDI.
- **mk_SEM_CLAIM_FAILED** Unable to access the SRDI Manager.
- **mk_KEY_NOT_VALID** The designated master key is not valid.
- **mk_INVALID_KEY_SELECTOR** The input parameters are not valid.
**triple_decrypt_under_master_key**

*triple_decrypt_under_master_key* triple decrypts eight bytes of data with the EDE algorithm, using the specified master key register.

**Function Prototype**

```c
long triple_decrypt_under_master_key( mk_selectors *mk_selector,
    UCHAR *ciphertext,
    UCHAR *cleartext);
```

**Input**

On entry to this routine:

- **mk_selector** is a pointer to a variable which must be initialized as follows:
  - **mk_set** is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to **MK_SET_DEFAULT**.
  - **mk_register** is set to either **old_mk**, **current_mk**, or **new_mk**, representing the key which should be cleared.
  - **type_mks** should be set to **ASYM_MK** if the data is to be decrypted with the asymmetric-keys master key or to **SYM_MK** if the data is to be decrypted with the symmetric-keys master key.

- **ciphertext** is a pointer to a buffer containing the data to be deciphered.

- **cleartext** is a pointer to a buffer 8 bytes in length. This may be the same as the ciphertext buffer.

**Output**

On successful exit from this routine:

- **cleartext** contains the deciphered data. This buffer may be the same as the ciphertext buffer, if desired.

**Return Codes**

Common return codes generated by this routine are:

- **mk_NO_ERROR** The operation was successful.
- **mk_SRDI_OPEN_ERROR** Could not open the master key SRDI.
- **mk_KEY_NOT_VALID** The master key could not be validated, therefore cleartext is unchanged.
- **mk_INVALID_KEY_SELECTOR** The input parameters are not valid.
**triple_decrypt_under_master_key_with_CV**

triple_decrypt_under_master_key_with_CV triple decrypts eight bytes of data with the EDE algorithm, using a control vector with the specified master key.

**Note:** This function does not check the validity of the control vector.

**Function Prototype**

```c
long triple_decrypt_under_master_key_with_CV( mk_selectors *mk_selector,
                                          eightbyte *cv,
                                          UCHAR *ciphertext,
                                          UCHAR *cleartext )
```

**Input**

On entry to this routine:

- `mk_selector` is a pointer to a variable which must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is set to either `old_mk`, `current_mk`, or `new_mk`, representing the key which should be used.
  - `type_mks` should be set to `ASYM_MK` if the data is to be decrypted with the asymmetric-keys master key or to `SYM_MK` if the data is to be decrypted with the symmetric-keys master key.

- `cv` is a pointer to a double-length CCA control vector, which is exclusive-ORed with the specified key value before the key is used.

- `ciphertext` is a pointer to a buffer containing the data to be deciphered.

- `cleartext` is a pointer to a buffer 8 bytes in length. This may be the same as the ciphertext buffer.

**Output**

On successful exit from this routine:

- `cleartext` is a pointer to the buffer where the deciphered data is placed. This buffer may be the same as the ciphertext buffer, if desired.

**Return Codes**

Common return codes generated by this routine are:

- `mk_NO_ERROR` The operation was successful.
- `mk_SRDI_OPEN_ERROR` Could not open the master key SRDI.
- `mk_SEM_CLAIM_FAILED` Unable to access the SRDI Manager.
- `mk_INVALID_KEY_SELECTOR` The input parameters are not valid.
triple_encrypt_under_master_key

triple_encrypt_under_master_key triple encrypts eight bytes of data with the EDE algorithm, using the specified master key register.

Function Prototype
long triple_encrypt_under_master_key( mk_selectors *mk_selector, UCHAR *cleartext, UCHAR *ciphertext) ;

Input
On entry to this routine:

mk_selector is a pointer to a variable which must be initialized as follows:

- mk_set is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- mk_register is set to either old_mk, current_mk, or new_mk, representing the key which should be used.
- type_mks should be set to ASYM_MK if the data is to be encrypted with the asymmetric-keys master key or to SYM_MK if the data is to be encrypted with the symmetric-keys master key.

cleartext is a pointer to a buffer containing the data to be enciphered.

ciphertext is a pointer to a buffer which is 8 bytes in length. This may be the same as the cleartext buffer.

Output
On successful exit from this routine:

ciphertext is a pointer to the buffer where the enciphered data is placed. This buffer may be the same as the cleartext buffer, if desired.

Return Codes
Common return codes generated by this routine are:

mk_NO_ERROR The operation was successful.
mk_SRDI_OPEN_ERROR Could not open the master key SRDI.
mk_SEM_CLAIM_FAILED Unable to access the SRDI Manager.
mk_KEY_NOT_VALID The master key (OMK, CMK, or NMK) is not a valid key.
mk_INVALID_KEY_SELECTOR The input parameters are not valid.
**triple_encrypt_under_master_key_with_CV**

triple_encrypt_under_master_key_with_CV triple encrypts eight bytes of data with the EDE algorithm, using a control vector with the specified master key.

**Note:** This function does not check the validity of the control vector.

**Function Prototype**

```c
long triple_encrypt_under_master_key_with_CV(    mk_selectors *mk_selector,
                                                eightbyte *cv,
                                                UCHAR *cleartext,
                                                UCHAR *ciphertext)
```

**Input**

On entry to this routine:

- **mk_selector** is a pointer to a variable which must be initialized as follows:
  - **mk_set** is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to **MK_SET_DEFAULT**.
  - **mk_register** is set to either **old_mk**, **current_mk**, or **new_mk**, representing the key which should be used.
  - **type_mks** should be set to **ASYM_MK** if the data is to be encrypted with the asymmetric-keys master key or to **SYM_MK** if the data is to be encrypted with the symmetric-keys master key.

- **cv** is a pointer to a double-length CCA control vector, which is exclusive-ORed with the specified key value before the key is used.

- **cleartext** is a pointer to a buffer containing the data that is enciphered.

- **ciphertext** is a pointer to a buffer which can hold 8 bytes of data. This may be the same as the **cleartext** buffer.

**Output**

On successful exit from this routine:

- **ciphertext** is a pointer to the buffer where the enciphered data is placed. This buffer may be the same as the **cleartext** buffer, if desired.

**Return Codes**

Common return codes generated by this routine are:

- **mk_NO_ERROR** The operation was successful.
- **mk_SRDI_OPEN_ERROR** Could not open the master key SRDI.
- **mk_SEM_CLAIM_FAILED** Unable to access the SRDI Manager.
Chapter 7. 

SHA-1 Functions

The functions described in this chapter allow a UDX to compute the hash of a block of data using the Secure Hash Algorithm (SHA-1) as defined in FIPS Publication 180-1.

Note: Each function is labeled with regard to whether it is available on the workstation host, the coprocessor, or both. None of these functions are available on the z/OS host.

Header Files for SHA-1 Functions

When using these functions, your program must include the following header files:

```
#include "cmncryt2.h"        /* Cryptographic types */
#include "cmn_sha.h"         /* SHA external definitions */
```

Summary of Functions

The following functions are described in this chapter:

- `computeHMAC_SHA1`: Compute a keyed-hash for message authentication code (HMAC) for a given set of data using the Secure Hash Algorithm (SHA-1).
- `do_sha_hash_message`: Compute the hash of a block of data using the SHA-1 algorithm.
- `do_sha_hash_msg_to_bfr`: “wrapper” for `do_sha_hash_message`.
- `hw_sha_hash_message`: Compute a SHA-1 hash of the requested data on the hashing hardware.
- `sha_hash_message`: Compute the hash of a block of data using the SHA-1 algorithm.
- `sha_hash_msg_to_bfr`: “wrapper” for `sha_hash_message`.

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 `do_sha_hash_message` with `MsgPart` set to `first` and the first piece of the block of data to hash identified by `pBlock` and `*pBitCount`. On return, `*pContext` contains context information that must be preserved and passed to `sha_hash_message` when the next piece of the block of data to hash is processed.

Subsequent pieces of the block are processed by calling `sha_hash_message` with `MsgPart` set to `middle` (or to `final` if the piece in question is the last) and the location and length of the piece identified by `pBlock` and `*pBitCount`. `*pContext` must contain the value returned in that structure by the call to `sha_hash_message` that processed the previous piece of the block. The function hashes the piece and updates `*pContext` and `pHash` appropriately.
Examples
To compute the SHA-1 hash of a contiguous block of 150 bytes of text at pBlock:

```c
BitCount = ( (uint64_t) 150) *8;
memset ( (UCHAR *) pContext, 0x00 , sizeof( sha_context) ) ;

do_sha_hash_message( pBlock, & Hash, & BitCount, pContext, only) ;
```

To compute the SHA-1 hash of only the name fields of the following structure:

```c
struct emp_data{
    char ID [ 10 ] ;
    double salary;
    char name [ 64 ] ;
} employee [ MAX_EMP ] ;

BitCount = (uint64_t) 512;
memclr ( ( UCHAR ) & Context, x , sizeof( sha_context) ) ;
/* Start the hash with "first" */
sha_hash_message( employee [ i ] . name, & Hash, & Bitcount, & Context, first) ;
/* hash the middle portions */
for ( i = 1; i<MAX_EMP; i++ )
{
    /* it is important that the value in BitCount is divisible by 512 */
    do_sha_hash_message( employee [ i ] . name, & Hash, & BitCount, & Context, middle) ;
}
/* hash the final portion */
sha_hash_message( employee [ MAX_EMP-1 ] . name, & Hash, & BitCount, & Context, final) ;
/* at this point, , the value in Hash is the SHA-1 hash of the names */
```
**computeHMAC_SHA1 - Compute HMAC using SHA-1 Algorithm**

**Note:** This function is available on the workstation host and on the coprocessor.

computeHMAC_SHA1 computes a keyed-hash for message authentication code (HMAC) for a given set of data, using the SHA-1 algorithm and a provided key.

**Function Prototype**

```c
void computeHMAC_SHA1 ( char *pBuffer,
                        int  buffer_length,
                        char *pKey,
                        int  key_length,
                        char *pHmac ) ;
```

**Input**

On entry to this routine:

- `pBuffer` is a pointer to an array which holds the data to be hashed for message authentication codes.
- `buffer_length` is the number of bytes of data in `pBuffer`.
- `pKey` is a pointer to a key (a random string of bits, preferably 64 bytes long).
- `key_length` is the number of bytes of data in `pKey`.
- `pHmac` is a pointer to a buffer 20 bytes long, which will hold the returned data.

**Output**

On successful exit from this routine:

- `pHmac` contains the HMAC-SHA1 of the data in `pBuffer`.

**Return Codes**

This function has no return codes.
**do_sha_hash_message - Calculate SHA-1 Hash Hardware/Software**

**Note:** This function is available on the coprocessor.

`do_sha_hash_message` calculates the SHA-1 hash of the data using the coprocessor hardware if the data is longer than 8192 bits, or the software if the data is shorter.

**Function Prototype**

```c
ULONG do_sha_hash_message (UCHAR *pBlock,
                          UCHAR *pHash,
                          uint64_t *pBitCount,
                          sha_context *pContext,
                          owl_sequence  MsgPart) ;
```

**Input**

On entry to this routine:

- `MsgPart` controls the operation of the function and must be one of the following constants:
  - **only** The input data constitutes the entire block of data to be hashed. The hash value is computed and returned.
  - **first** The input data constitutes the first portion of a block of data to be hashed. See "Chained Operations" on page 7-1 for details.
  - **middle** The input data constitutes an additional portion of a block of data to be hashed. See "Chained Operations" on page 7-1 for details.
  - **final** The input data constitutes the final portion of a block of data to be hashed. See "Chained Operations" on page 7-1 for details.

- `pBlock` must contain the address of the block of data that is to be incorporated into the hash.
- `pHash` must contain the address of a buffer to which the hash value may be written. The buffer must be at least 20 bytes long. `pHash` is used only if `MsgPart` specifies **only** or **final**.
- `pBitCount` must contain the address of a buffer that contains the length in bits of the block of data referenced by `pBlock`.

If `MsgPart` specifies **first** or **middle**, *pBitCount* must be a multiple of 512, or data will be lost.

- `pContext` must contain the address of a context buffer from which the function may initialize its internal state and to which the function may write its final internal state. See "Chained Operations" on page 7-1 for details.

If `MsgPart` specifies **only** or **first**, the initial value of *pContext is ignored."
Output
On successful exit from this routine:

The buffer referenced by \texttt{pHash} contains the hash value of the input data if \texttt{MsgPart} specifies \texttt{only} or \texttt{final}. In the latter case, the hash value incorporates the initial hash value provided in \texttt{*pContext}.

\texttt{*pContext} has been updated to incorporate changes to the function’s internal state caused by incorporating \texttt{*pBlock} into the hash.

Return Codes
Common return codes generated by this routine are:

- \texttt{sh\_NO\_ERROR} (i.e., 0)  The operation was successful.
- \texttt{sh\_MSG\_PART\_INVALID}  The \texttt{MsgPart} argument was not \texttt{only}, \texttt{first}, \texttt{middle}, or \texttt{last}. 
**do_sha_hash_msg_to_bfr - SHA-1 Hash**  
**Note:** This function is available only on the coprocessor.

`do_sha_hash_msg_to_bfr` is a wrapper for `sha_hash_message` that simplifies the interface when chained operations (see page 7-1) are not necessary.

**Function Prototype**

```c
void do_sha_hash_msg_to_bfr( UCHAR *pBlock, UCHAR *pHash, uint64_t *pBitCount );
```

**Input**

On entry to this routine:

- `pBlock` must contain the address of the block of data that is to be hashed.

- `pHash` must contain the address of a buffer to which the hash value may be written. The buffer must be at least 20 bytes long.

- `pBitCount` must contain the address of a buffer that contains the length in bits of the block of data referenced by `pBlock`.

**Output**

On successful exit from this routine:

The buffer referenced by `pHash` contains the hash value of the input data.

**Notes**

**Function Wraps do_sha_hash_message**

```c
do_sha_hash_msg_to_bfr( pBlock, pHash, pBitCount ) performs the same function as
{
    sha_context Context;
    memset( &Context, S, sizeof(Context) );
    do_sha_hash_message(pBlock, pHash, pBitCount, & Context, only);
}
```

**Return Codes**

This function has no return codes.
**hw_sha_hash_message - Compute SHA-1 Hash in Hardware**

**Note:** This function is available only on the coprocessor.

hw_sha_hash_message computes a SHA-1 hash of the requested data on the hashing hardware in the coprocessor.

**Function Prototype**

```c
void hw_sha_hash_message( UCHAR *pText,
                         long text_length,
                         sha_context *chain_vector,
                         long RequestId,
                         ULONG options,
                         UCHAR *final_data,
                         long *pMsg);
```

**Input**

On entry to this routine:

- `pText` is a pointer to the data which is to be hashed.
- `text_length` is the number of bytes of data at `pText`.
- `chain_vector` is the context of this request. If this is the first block of data, or the only block, this variable should be initialized to zeros. On subsequent calls, this variable should be returned to the next call without change.
- `RequestId` is the ID of the caller.
- `options` may include `SHA_MSGPART_FIRST`, `SHA_MSGPART_MIDDLE`, `SHA_MSGPART_FINAL`, or `SHA_MSGPART_ONLY`. These constants are defined in the `xc_types.h` header file.
- `final_data` is a pointer to a buffer at least 20 bytes long, to hold the result of the hash.
- `pMsg` is a pointer to a 4-byte block.

**Output**

On successful exit from this routine:

- `chain_vector` contains the chaining information for the next call to the function, unless this call was with the final or only block of data. This variable should be passed unchanged to the next call.
- `final_data` contains the final hash value, if this call was with the final or only data block.
- `pMsg` contains the return code for the function.
Return Codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>SHA1_DATA64_ERROR</td>
<td>The options argument specified SHA_MSGPART_FIRST or SHA_MSGPART_MIDDLE but the length of the data to process is not a multiple of 64.</td>
</tr>
<tr>
<td>SHA1_FINAL_ERROR</td>
<td>An error occurred while attempting to pad the input data as directed by the SHA-1 algorithm or while attempting to hash the pad bytes.</td>
</tr>
</tbody>
</table>
**sha_hash_message - SHA-1 Hash with Chaining**

*Note:* This function is available on the workstation host and the coprocessor.

sha_hash_message computes the hash of a block of data using the Secure Hash Algorithm (SHA-1) and optionally incorporates the result into an initial hash value. This function calculates the hash in software.

**Function Prototype**

```
ULONG sha_hash_message ( UCHAR *pBlock,
                         UCHAR *pHash,
                         uint64_t *pBitCount,
                         sha_context *pContext,
                         owi_sequence  MsgPart);
```

**Input**

On entry to this routine:

- **MsgPart** controls the operation of the function and must be one of the following constants:
  - **only** The input data constitutes the entire block of data to be hashed. The hash value is computed and returned.
  - **first** The input data constitutes the first portion of a block of data to be hashed. See "Chained Operations" on page 7-1 for details.
  - **middle** The input data constitutes an additional portion of a block of data to be hashed. See "Chained Operations" on page 7-1 for details.
  - **final** The input data constitutes the final portion of a block of data to be hashed. See "Chained Operations" on page 7-1 for details.

- **pBlock** must contain the address of the block of data that is to be incorporated into the hash.

- **pHash** must contain the address of a buffer to which the hash value may be written. The buffer must be at least 20 bytes long. **pHash** is used only if **MsgPart** specifies **only** or **final**.

- **pBitCount** must contain the address of a buffer that contains the length in bits of the block of data referenced by **pBlock**.

  If **MsgPart** specifies **first** or **middle**, **pBitCount** must be a multiple of 512, or data will be lost.

- **pContext** must contain the address of a context buffer from which the function may initialize its internal state and to which the function may write its final internal state. See "Chained Operations" on page 7-1 for details.

  If **MsgPart** specifies **only** or **first**, the initial value of **pContext** is ignored.
Output
On successful exit from this routine:

The buffer referenced by `pHash` contains the hash value of the input data if `MsgPart` specifies `only` or `final`. In the latter case, the hash value incorporates the initial hash value provided in `*pContext`.

`*pContext` has been updated to incorporate changes to the function’s internal state caused by incorporating `*pBlock` into the hash.

Return Codes
Common return codes generated by this routine are:

- **sh_NO_ERROR (i.e., 0)** The operation was successful.
- **sh_MSG_PART_INVALID** The `MsgPart` argument was not `only`, `first`, `middle`, or `last`. 
sha_hash_msg_to_bfr - SHA-1 Hash

**Note:** This function is available on the workstation host and the coprocessor.

sha_hash_msg_to_bfr is a wrapper for sha_hash_message that simplifies the interface when chained operations (see page 7-1) are not necessary.

**Function Prototype**

```c
void sha_hash_msg_to_bfr( UCHAR *pBlock, UCHAR *pHash, uint64_t *pBitCount ) ;
```

**Input**

On entry to this routine:

- `pBlock` must contain the address of the block of data that is to be hashed.

- `pHash` must contain the address of a buffer to which the hash value may be written. The buffer must be at least 20 bytes long.

- `pBitCount` must contain the address of a buffer that contains the length in bits of the block of data referenced by `pBlock`.

**Output**

On successful exit from this routine:

The buffer referenced by `pHash` contains the hash value of the input data.

**Notes**

**Function Wraps sha_hash_message**

`sha_hash_msg_to_bfr(pBlock, pHash, pBitCount)` performs the same function as

```c
{
    sha_context Context;
    memset( &Context, 0x00, sizeof(Context) ) ;
    sha_hash_message(pBlock, pHash, pBitCount, & Context, only) ;
}
```

**Return Codes**

This function has no return codes.
This chapter describes functions to assist in the use of key tokens and other cryptographic structures.

You should understand the use of the CCA control-vector before using the functions in this chapter. Control vectors are explained and described in Appendix C of the CCA Basic Services Reference and Guide. Three bits in the basic control vector have been reserved for UDX developers. Setting Bit 61 will prevent a key token from being used in any CCA standard verb except the import and export verbs. Bits 4 and 5 of the control vector will not be checked by any standard CCA code. This allows developers to use these three bits to indicate their own, UDX specific keys, which can be used only by UDX verbs. (These verbs must be written to test the required bits.)

**Note:** All functions within this chapter are available only on the coprocessor.

### Header Files for DES Utility Functions

When using these functions, your program must include the following header files.

```c
#include "cmncryt2.h"    /* T2 definitions */
#include "castyped.h"    /* Adapter structures */
#include "cassub.h"      /* DES 96 prototypes */
#include "casfunct.h"    /* TDES prototypes */
```

### Summary of Functions

DES utility routines include the following functions.

- **cas_adjust_parity** Adjusts the parity of a DES key token. See page 8-4.
- **cas_build_default_token** Builds a default DES key token using the default master key set. See page 8-6.
- **cas_build_default_cv** Builds a default control vector. See page 8-5.
- **cas_current_mkvp** Returns the current master key verification pattern from the default key set. See page 8-8.
- **cas_des_key_token_check** Verifies the integrity of a DES key token. See page 8-9.
- **cas_get_key_type** Returns the type of DES key token. See page 8-10.
- **cas_key_length** Determines the length of a DES key from its control vector. See page 8-11.
- **cas_key_tokentv_check** Verifies a DES key token validation value. See page 8-12.
- **cas_master_key_check** Performs a master key version check using the default master key set. See page 8-13.
- **cas_old_mkvp** Returns the old master key verification pattern from the default key set. See page 8-14.
cas_parity_odd
Determines whether a DES key has odd parity. See page 8-15.

CasBuildCv
Builds a default control vector. See page 8-5.

CasBuildToken
Builds a default DES key token. See page 8-6.

CasCurrentMkvp
Returns the current master key verification pattern. See page 8-8.

CasMasterKeyCheck
Performs a master key version check. See page 8-13.

CasOldMkvp
Returns the old master key verification pattern. See page 8-14.

des_engine_ede2_triple_decrypt
TDES decrypt, using a double length key. See page 8-16.

des_engine_ede2_triple_encrypt
TDES encrypt, using a double length key. See page 8-17.

RecoverDESDataKey
Recovers the cleartext form of a DES data key with the default master key set. See page 8-18.

RecoverDesDataKeyWithMK
Recovers the cleartext form of a DES data key. See page 8-18.

RecoverDesKekImporter
Recovers the cleartext form of a DES key encrypting key (KEK). See page 8-20.

RecoverDesKekImporterWithMK
Recovers the cleartext form of a DES key encrypting key (KEK). See page 8-20.

S390KeyLength
Determines the key length (SINGLE, DOUBLE, or TRIPLE) of a key token. See page 8-22.

Overview
The routines described in this chapter are used to analyze, modify, and validate CCA DES key tokens.

Refer to the IBM 4764 PCI-X Cryptographic Coprocessor CCA Basic Services Reference and Guide for more information. Chapter 5, “Basic CCA DES Key Management” includes an in-depth discussion of DES key token management within CCA. You can also refer to Appendix B, “Data Structures” for a description of the DES key tokens structures and Appendix C, “CCA Control Vector Definitions and Key Encryption” for a discussion of control vectors.

Keys used in these functions are one of the following KEY_TYPES:

DATA_KEY
For the encryption and decryption of data.

DATAXLATE_KEY
To re-encipher data from one key to another.

CIPHER_KEY
A symmetric key to encipher and decipher data.

ENCIPHER_KEY
A non-symmetric key, which only enciphers data.

DECIPHER_KEY
A non-symmetric key, which only deciphers data.

MAC_KEY
For generating and verify Message Authentication Codes.

MACVER_KEY
For verifying Message Authentication Codes.

IMPORTER_KEY
For decoding keys imported from other engines, or translating keys from one encoding to another.

EXPORTER_KEY
For encoding keys for export (to other engines), or translating keys from one encoding to another.
IKEYXLATE_KEY  For inputting a key translation.
OKEYXLATE_KEY  For outputting a key translation.
PINGEN_KEY     For generating PINs.
PINVER_KEY      For verifying PINs.
IPINENC_KEY     For importing PINs.
OPINENC_KEY     For exporting PINs.
KEYGEN_KEY      Used for key generation.
DKYGEN_KEY      Used for key diversification.
CVARENC_KEY     For encrypting cryptographic variables.
SECMMSG_KEY     For encrypting messages containing keys or PINs.
KEY_TYPE_TOKEN  A key token, rather than a key.
**cas_adjust_parity - Adjust Parity**

cas_adjust_parity adjusts each byte of the passed string, as necessary, so that every byte has odd parity. This is useful when adjusting DES keys for correct parity.

**Function Prototype**

```c
void cas_adjust_parity( UCHAR *DataBytes, 
                        unsigned int Length)
```

**Input**

On entry to this routine:

- **DataBytes** is a pointer to the string that is to be parity-adjusted.
- **Length** is the number of bytes in the string at location **DataBytes**.

**Output**

On successful exit from this routine:

- **DataBytes** is a pointer to the string that has odd parity.

**Return Codes**

This function has no return codes.
**CasBuildCv - Build a Default Control Vector**

CasBuildCv builds a default control vector for the specified key type.

**Function Prototype**

```c
boolean CasBuildCv ( KEY_TYPES   KeyType,
                     CV_LENGTHS   CV_Length,
                     UCHAR        *pCV )

void cas_build_default_cv ( KEY_TYPES               KeyType,
                            UCHAR             *pCV )
```

`cas_build_default_cv` has the same effect as calling `CasBuildCv` after setting the `CV_Length` parameter to `CV_DEFAULT`.

**Input**

On entry to this routine:

- **KeyType** is the type of key your control vector is used with. Possible key types on both workstation and zSeries builds include:

  - IMPORter_KEY
  - PINGEN_KEY
  - DOUBLE_DATA_KEY
  - ENCIPHER_KEY
  - CVARENC_KEY
  - EXPORTER_KEY
  - PINVER_KEY
  - DATA_KEY
  - DECIPHER_KEY
  - OKEYXLAT_KEY
  - IPINENC_KEY
  - DATAVALID_KEY
  - MAC_KEY
  - IKKEYXLAT_KEY
  - OPINENC_KEY
  - CIPHER_KEY
  - MACVER_KEY
  - DATAC_KEY
  - DATAMV_KEY
  - MACVER_MACVER_KEY
  - CVARPINE_KEY
  - CVARXCVR_KEY
  - DATAM_KEY
  - MAC_MAC_KEY
  - CVARDEC_KEY
  - CVARXCVL_KEY
  - IMP_PKA_KEY

- **pCv** is a pointer to a 16-byte location which will hold the new control vector.

- **CV_Length** is one of `CV_DOUBLE`, `CV_SINGLE`, or `CV_DEFAULT`, depending on what length of key you are building a control vector for.

**Output**

On successful exit from this routine:

- `CasBuildCv` returns true if the build was successful, and false otherwise. (For example, if the length requested was not legal for the key type.)

- `pCV` contains the new control vector.

**Return Codes**

This function has no return codes.
CasBuildToken - Build a Default Token
CasBuildToken builds a default key token, of the type specified by parameter KeyType for the mk_set being processed.

Function Prototype

```c
boolean CasBuildToken ( UCHAR  TokenFlag,
                        KEY_TYPES  KeyType,
                        CV_LENGTHS  CV_Length,
                        mk_selectors *pMKSelector,
                        des_key_token_structure *pKeyToken )
```

```c
void cas_build_default_token ( UCHAR  TokenFlag,
                               KEY_TYPES  KeyType,
                               des_key_token_structure *pKeyToken )
```

cas_build_default_token has the same effect as calling CasBuildToken with CV_Length set to CV_DEFAULT and MKSelector set to (MK_SET_DEFAULT, current_mk, SYM_MK).

Input
On entry to this routine:

**TokenFlag** is the token flag used in constructing the new key token. Legal values for this field are:

- **INTERNAL_TOKEN_FLAG** An operational key, which will be encrypted by the master key.
- **EXTERNAL_TOKEN_FLAG** A key which will be encrypted by another key.

**KeyType** is the type of key token to be generated. Examples include data key, exporter key, and MAC key. Possible key types on both workstation and zSeries builds include:

- IMPORTER_KEY
- PINGEN_KEY
- DOUBLE_DATA_KEY
- ENCIPHER_KEY
- CVARENC_KEY
- EXPORTER_KEY
- PINVER_KEY
- DATA_KEY
- DECIPHER_KEY
- OKEYXLAT_KEY
- IPINENC_KEY
- DATAXLAT_KEY
- MAC_KEY
- IKEYXLAT_KEY
- OPINENC_KEY
- CIPHER_KEY
- MACVER_KEY

Additional key types supported only on zSeries builds include:

- DATAC_KEY
- DATAMV_KEY
- MACVER_MACVER_KEY
- CVARPINE_KEY
- CVARXCVR_KEY
- DATAM_KEY
- MAC_MAC_KEY
- CVARDEC_KEY
- CVARXCVL_KEY
- IMP_PKA_KEY

**CV_Length** is one of the following:

- **CV_DOUBLE** A double length (16 byte) key.
- **CV_SINGLE** A single length (8 byte) key.
- **CV_DEFAULT** A key the default length for the designated KeyType.
MKSelector is a parameter of type `mk_selectors`, indicating which set of master keys to use in this function. This variable must be initialized as follows:

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
- `mk_register` must be set to `current_mk`.
- `type_mks` should be set to `SYM_MK`.

`pKeyToken` is a pointer to a 64-byte buffer which can store a key token.

**Output**
On successful exit from this routine:

`pKeyToken` contains the token constructed by the function.

**Return Codes**
This function has no return codes.
**CasCurrentMkvp - Current Master Key Verification Pattern**

CasCurrentMkvp returns the 20-byte master key verification pattern (MKVP) for the current master key for the `mk_set` being processed. The MKVP is a cryptographically calculated checksum on the master key value. It is used in all internal (master-key encrypted) DES key tokens, to indicate which master key was used to encrypt the key.

**Function Prototype**

```c
boolean CasCurrentMkvp ( mk_selectors *pMKSelector,
                        UCHAR *pMKVP)
```

```c
boolean cas_current_mkvp( UCHAR *pMKVP )
```

cas_current_mkvp has the same effect as calling CasCurrentMkvp with the `pMKSelector` parameter set to `{MK_SET_DEFAULT, current_mk, SYM_MK}`.

**Input**

On entry to this routine:

- `pMKSelector` is a parameter of type `mk_selectors`, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` must be set to `current_mk`.
  - `type_mks` should be set to `SYM_MK`.

- `pMKVP` must contain the address of a variable in which a 20-byte master key verification pattern can be stored.

**Output**

On successful exit from this routine:

- `pMKVP` contains the current master key verification pattern.

**CasCurrentMkvp** returns **TRUE** if the verification pattern was found, and **FALSE** otherwise.

**Return Codes**

This function has no return codes.
**cas_des_key_token_check - Verify the DES Key Token**

cas_des_key_token_check performs the following checks to verify the integrity of an internal DES key token.

- Check that all reserved fields are zero.
- Check the token flag.
- Check the version number.
- Check the flags.

If no errors are found, the function returns TRUE. If there is an error, the function returns FALSE and parameter pMessageFlag indicates the cause of the error. Either version 0 or version 1 DES key tokens may be validated.

**Function Prototype**

```c
boolean cas_des_key_token_check( des_key_token_structure *pKeyToken,
                                DES_TOKEN_CHECK *pMessageFlag )
```

**Input**

On entry to this routine:

- pKeyToken is a pointer to the internal DES key token that is to be checked.
- pMessageFlag is a pointer to a location where the return code can be stored.

**Output**

On successful exit from this routine:

The buffer at pMessageFlag contains one of the values defined in “Return Codes” below.

**cas_des_key_token_check** returns a boolean value of TRUE, if the value returned in pMessageFlag is DES_TOKEN_CHECK_VALID, or FALSE otherwise.

**Return Codes**

Common return codes generated by this routine are:

- **DES_TOKEN_CHECK_VALID** - The token is valid.
- **DES_TOKEN_CHECK_TOKENFLAG** - The token is not an internal DES key token.
- **DES_TOKEN_CHECK_RESERVEDi** - Reserved field $i$ (1 <= i <= 6) is incorrectly set.
- **DES_TOKEN_CHECK_VERSION** - The version number is incorrect.
- **DES_TOKEN_CHECK_FLAG_BYTE** - The token flag is incorrect.
- **DES_TOKEN_CHECK_FLAG_NOCV** - The token has no control vector set.
- **DES_TOKEN_CHECK_NOKEY** - The token does not contain a key.
- **DES_TOKEN_CHECK_KEY_LENGTH** - The key length is invalid.
- **DES_TOKEN_CHECK_VERSION_ONE** - The key is a zero CV key, a double or triple length DATA key for S390.
- **DES_TOKEN_CHECK_INVALID_ZEROCV** - The key is a zero CV key but the control vector is a non-zero.
cas_get_key_type - Return Key Type

cas_get_key_type returns the key type corresponding to the specified key token.

Function Prototype
KEY_TYPES cas_get_key_type( des_key_token_structure *pKeyToken )

Input
On entry to this routine:

pKeyToken is a pointer to the key token which is to be examined.

Output
On successful exit from this routine:

cas_get_key_type returns the key type corresponding to the specified key token.

Return Codes
This function has no return codes.
**cas_key_length - Return Key Length**

`cas_key_length` determines the length of a key, based on the Control Vector. The key length is returned as the function result.

**Note:** This function does not properly deal with external DATA tokens, or CCF DATA tokens, which have zero-value control vectors. Functions which may be testing external DATA keys or keys with zero-value control vectors, should use the `S390KeyLength` function instead.

**Function Prototype**

```
LENGTH_KEYWORD cas_key_length(     eightbyte CvBase,
                    eightbyte CvExtension )
```

**Input**

On entry to this routine:

- `CvBase` is the control vector base.
- `CvExtension` is the control vector extension.

**Output**

On successful exit from this routine:

- `cas_key_length` returns `SINGLE` or `DOUBLE`, depending on whether the specified key is single or double length.

**Examples**

To determine the length of the key stored in `DataKey`:

```
switch( cas_key_length( DataKey, cvBase, DataKey. cvExten ) )
{
    case SINGLE:
        /* deal with a single length key */
        break;
    case DOUBLE:
        /* deal with a double length key */
        break;
    default :
        /* return with an error */
}
```

**Return Codes**

This function has no return codes.
cas_key_tokentvv_check - Verify the Token Validation Value

cas_key_tokentvv_check verifies the Token Validation Value (TVV) in the specified internal DES key token. The TVV is an integrity check value used to detect corruption of the token.

The function returns TRUE if the TVV verifies, and FALSE if not.

**Function Prototype**

```c
boolean cas_key_tokentvv_check( des_key_token_structure *pKeyToken )
```

**Input**

On entry to this routine:

- `pKeyToken` is a pointer to the internal DES key token that you want to check.

**Output**

On successful exit from this routine:

- `cas_key_token_tvv_check` returns a boolean value of **TRUE** if the TVV verifies, and **FALSE** if not.

**Return Codes**

This function has no return codes.
**CasMasterKeyCheck - Master Key Version Check**

CasMasterKeyCheck determines which version of the master key was used to encrypt the specified key token for the mk_set being processed. The response indicates whether the key token is encrypted using the current master key, the old master key, or a master key that is no longer available.

**Function Prototype**

```
UNDER_MASTER_KEY CasMasterKeyCheck (  mk_selectors  *pMKSelector,
                                      des_key_token_structure  *pKeyToken )

UNDER_MASTER_KEY cas_master_key_check( des_key_token_structure  *pKeyToken )
```

cas_master_key_check has the same effect as calling CasMasterKeyCheck with the pMKSelector parameter set to {MK_SET_DEFAULT, current_mk, SYM_MK}.

**Input**

On entry to this routine:

- pMKSelector is a parameter of type mk_selectors, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - mk_set is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
  - mk_register is ignored.
  - type_mks should be set to SYM_MK.

- pKeyToken is a pointer to the key token which is to be examined.

**Output**

On successful exit from this routine:

CasMasterKeyCheck returns either OLD, CURRENT, or OUT_OF_DATE which identifies which master key (old, current, or no longer available) the key token is encrypted under.

If CasMasterKeyCheck did not return OUT_OF_DATE, the mk_registers field of the pMKSelectors parameter has been updated to the appropriate value (old_mk or current_mk).

**Notes**

In CCA, an “operational key” is a key that has been multiply-enciphered with the master key. In order to use an operational key, it must first be deciphered using the master key.

When the user (security officers, and so on) updates the master key, CCA maintains a copy of the old master key. This routine determines which version of the master key was used to encipher the specified key token (CCA does this by maintaining a hash value of the master key called the master key verification pattern which is stored in the DES key token). Refer to Appendix B of the *IBM 4764 PCI-X Cryptographic Coprocessor CCA Basic Services Reference and Guide* for more information.

Since CCA only stores 2 versions of the master key (current and old), upon encountering a key token enciphered with the old master key, the UDX developer may opt to re-encipher the key token using the current master key.

**Return Codes**

This function has no return codes.
**CasOldMkvp - Old Master Key Verification Pattern**

CasOldMkvp returns the 20-byte master key verification pattern (MKVP) for the old master key for the mk_set being processed. The MKVP is a cryptographically calculated checksum on the master key value. It is used in all internal (master-key encrypted) DES key tokens, to indicate which master key was used to encrypt the key.

**Function Prototype**

```c
boolean CasOldMkvp ( mk_selectors *pMKSelector,
                    UCHAR *pMKVP)

boolean cas_old_mkvp( UCHAR *pMKVP )
```

`cas_old_mkvp` has the same effect as calling `CasOldMkvp` with the `pMKSelector` parameter set to `{MK_SET_DEFAULT, current_mk, SYM_MK}`.

**Input**

On entry to this routine:

- `MKSelector` is a parameter of type `mk_selectors`, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` must be set to `old_mk`.
  - `type_mks` should be set to `SYM_MK`.

- `pMKVP` must contain the address of a variable in which a 20-byte master key verification pattern can be stored.

**Output**

On successful exit from this routine:

- `pMKVP` contains the current master key verification pattern.

**CasOldMkvp** returns **TRUE** if the verification pattern was found, and **FALSE** otherwise.

**Return Codes**

This function has no return codes.
**cas_parity_odd - Verify Parity**
cas_parity_odd determines whether the specified byte has odd or even parity.

**Function Prototype**
boolean cas_parity_odd( UCHAR DataByte )

**Input**
On entry to this routine:

*DataByte* is the byte that is to be checked.

**Output**
On successful exit from this routine:

*cas_parity_odd* returns **TRUE** if the specified byte has odd parity, or **FALSE** if it has even parity.

**Return Codes**
This function has no return codes.
Des_engine_ede2_triple_decrypt - DES Engine EDE2 Triple Decrypt

Des_engine_ede2_triple_decrypt will triple decrypt data using the clear key specified. In a zSeries processor, this function will also perform a RAS test to ensure that the processor is working correctly.

Function Prototype

```c
unsigned long des_engine_ede2_triple_decrypt ( double_length_key *pKey,
                           char *pCipherData,
                           unsigned long DataLength,
                           char *pClearData )
```

Input

On entry to this routine:

- **pKey** is a pointer to a 16-byte string which is the clear DES key to be used for decryption.
- **pCipherData** is a pointer to the data which is to be decrypted.
- **DataLength** is the length of the data to be encrypted, in bytes. This must be a multiple of 8.
- **pCipherText** is a pointer to a buffer in which the result will be stored. This buffer must be at least **DataLength** bytes long.

Output

On successful exit from this routine:

- **pClearData** contains the decrypted data.

Return Codes

Common return codes generated by this function are:

- **OK** The operation was successful.
- **CCA_ERR_RETRY** The RAS test failed. The card has been stopped.
- **Other codes** Other error codes may be returned by the xCrypto function **xc_EDE3_3DES**.
des_engine_ede2_triple_encrypt - DES Engine EDE2 Triple Encrypt

des_engine_ede2_triple_encrypt will triple encrypt data using the clear key specified. In a zSeries processor, this function will also perform a RAS test to ensure that the processor is working correctly.

Function Prototype

unsigned long des_engine_ede2_triple_encrypt ( double_length_key *pKey,
               char *pClearData,
               unsigned long DataLength,
               char *pCipherText )

Input

On entry to this routine:

pKey is a pointer to a 16-byte string which is the clear DES key to be used for encryption.

pClearData is a pointer to the data which is to be encrypted.

DataLength is the length of the data to be encrypted, in bytes. This must be a multiple of 8.

pCipherText is a pointer to a buffer in which the result will be stored. This buffer must be at least DataLength bytes long.

Output

On exit from this routine:

pCipherText contains the encrypted data.

Return Codes

Common return codes generated by this function are:

OK The operation was successful.

CCA_ERR_RETRY The RAS test failed. The card has been stopped.

Other codes Other error codes may be returned by the xCrypto function xc_EDE3_3DES.
**RecoverDesDataKeyWithMK - Recover DES Data Key**

RecoverDesDataKeyWithMK recovers the cleartext form of a DES data key that has been enciphered with the master key for the `mk_set` being processed. The input token is checked to ensure it is valid.

**Note:** This function assumes the key to be double length, but also tests for triple length keys.

**Function Prototype**

```c
long RecoverDesDataKeyWithMK ( des_key_token_structure *pDesToken,
                                mk_selectors *pMKSelector,
                                UCHAR *pClearKey,
                                long *pMsg)

long RecoverDesDataKey ( des_key_token_structure *pDesToken,
                         UCHAR *pClearKey,
                         long *pMsg)
```

*RecoverDesDataKey* has the same effect as calling *RecoverDesDataKeyWithMK* with the `pMKSelector` parameter set to `{MK_SET_DEFAULT, current_mk, SYM_MK}`.

**Input**

On entry to this routine:

- `pDesToken` is a pointer to the input key token.
- `pMKSelector` is a parameter of type `mk_selectors`, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is ignored.
  - `type_mks` should be set to `SYM_MK`.
- `pClearKey` is a pointer to a buffer where up to 24 bytes of key data can be stored.
- `pMsg` is a pointer to a buffer where a 4 byte return code can be stored.

**Output**

On successful exit from this routine:

The buffer pointed to by `pClearKey` is a pointer to the location where the function stores the recovered, cleartext key.

The buffer pointed to by `pMsg` contains the error code.

*RecoverDesDataKeyWithMK* and *RecoverDesDataKey* return `ERROR` if the value in `pMsg` is non-zero, or `OK` otherwise.
Return Codes
Common return codes generated by this routine are:

OK
The operation was successful.

ERROR
pMsg contains one of the following error codes:

- **RT_OMK_TOKEN_USED**: The key was encrypted with the Old master key (warning).
- **E_INTRN_TOKEN_TVV**: The token is not valid.
- **mk_SRDI_OPEN_ERROR**: Could not open the master key SRDI.
- **mk_SEM_CLAIM_FAILED**: Unable to access the master key SRDI.
- **RT_KEY_INV_MKVN**: The key was encrypted using an out-of-date master key.
**RecoverDesKekImporterWithMK - Recover DES Importer KEK**

RecoverDesKekImporterWithMK recovers the cleartext form of a DES importer key encrypting key (KEK) that has been enciphered with the master key for the \texttt{mk\_set} being processed. The token validation value is then confirmed.

**Note:** This function assumes that the key is double length.

**Function Prototype**

```c
long RecoverDesKekImporterWithMK ( des_key_token_structure *pDesToken, mk_selectors *pMKSelector, UCHAR *pClearKey, long *pMsg )

long RecoverDesKekImporter ( des_key_token_structure *pDesToken, UCHAR *pClearKey, long *pMsg )
```

**RecoverDesKekImporter** has the same effect as calling **RecoverDesKekImporterWithMK** with the \texttt{MKSelector} parameter set to \{\texttt{MK\_SET\_DEFAULT}, \texttt{current\_mk}, \texttt{SYM\_MK}\}.

**Input**

On entry to this routine:

- \texttt{pDesToken} is a pointer to the input key token.
- \texttt{MKSelector} is a parameter of type \texttt{mk\_selectors}, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - \texttt{mk\_set} is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to \texttt{MK\_SET\_DEFAULT}.
  - \texttt{mk\_register} is ignored.
  - \texttt{type\_mks} should be set to \texttt{SYM\_MK}.
- \texttt{pClearKey} is a pointer to a buffer where 16 bytes of key data can be stored.
- \texttt{pMsg} is a pointer to a buffer where a 4 byte return code can be stored.

**Output**

On successful exit from this routine:

- \texttt{pClearKey} contains the recovered, cleartext key.
- \texttt{pMsg} is the error code.
**Return Codes**

Common return codes generated by this routine are:

- **OK**  
  The operation was successful.

- **ERROR**  
  pMsg contains the error code.

- **RT_OMK_TOKEN_USED**  
  The key was encrypted with the Old master key (warning).

- **E_INTRN_TOKEN_TVV**  
  The token is not valid.

- **mk_SRDI_OPEN_ERROR**  
  Could not open the master key SRDI.

- **mk_SEM_CLAIM_FAILED**  
  Unable to access the master key SRDI.

- **RT_KEY_INV_MKVN**  
  The key was encrypted using an out-of-date master key.
**S390KeyLength - Find the Key Length of a DES Key Token**

S390KeyLength finds the length (whether **SINGLE**, **DOUBLE**, or **TRIPLE**) of a DES key in a CCA key token.

**Note:** Only S390 DATA tokens are allowed to be triple-length.

**Function Prototype**

```c
LENGTH_KEYWORD S390KeyLength (des_key_token_structure *pKeyToken)
```

**Input**

`pKeyToken` is a pointer to a 64-byte DES key token, either internal or external format key.

**Output**

On successful completion of this function:

S390KeyLength returns either **SINGLE**, **DOUBLE**, or **TRIPLE**, indicating the length of the key contained in `pKeyToken`.

**Return Codes**

This function has no return codes.
Chapter 9.

RSA Functions

This chapter contains functions for dealing with RSA keys and key tokens.

Refer to Appendix B of the IBM 4764 PCI-X Cryptographic Coprocessor CCA Basic Services Reference and Guide for an overview of public and private key token structures.

Note: All functions within this chapter are available only on the coprocessor.

Header Files for RSA Functions

When using these functions, your program must include the following header files.

```c
#include "cmncryt2.h" /* T2 CPRB definitions */
#include "scctypes.h" /* Secure Cryptographic Coprocessor structures */
#include "xc_types.h" /* xCrypto API */
#include "cam_xtrn.h" /* CCA managers */
#include "cadcbln.h" /* public header file */
#include "casfunct.h"
#include "cacmkld.h" /* functions for generating RSA */
/* signatures and registered keys */
```

Summary of Functions

RSA keys and key tokens include the following functions.

- **CalculatenWordLength**
  
  Returns the number of 4-byte words in the modulus. See page 9-7.

- **CreateInternalKeyToken**
  
  Receives a clear key token and creates the operational form with the default master key. See page 9-8.

- **CreateInternalKeyTokenWithMK**
  
  Receives a clear key token and creates the operational form. See page 9-8.

- **CreateRsaInternalSection**
  
  Creates the RSA internal section with the default master key set. See page 9-9.

- **CreateRsaInternalSectionWithMK**
  
  Creates the RSA internal section with the specified master key set. See page 9-9.

- **delete_KeyToken**
  
  Removes a registered public or private key from the SRDI where it is stored. See page 9-10.

- **ExtractRsaPublicSection**
  
  Format a public key token from a private/public key token. See page 9-11.

- **GenerateCcaRsaToken**
  
  Generates a CCA RSA key token from an internal format key token and a CCA PKA skeleton token. See page 9-12.

- **GenerateRsaInternalToken**
  
  Creates an internal RSA token from a CCA RSA key
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate_dSig</td>
<td>Generates a digital signature.</td>
</tr>
<tr>
<td>GetLength</td>
<td>Returns the RSA public exponent byte length.</td>
</tr>
<tr>
<td>getKeyToken</td>
<td>Retrieves a PKA token from the SRDI where it is stored.</td>
</tr>
<tr>
<td>GetModulus</td>
<td>Extracts and copies the RSA modulus.</td>
</tr>
<tr>
<td>GetnBitLength</td>
<td>Returns the bit length of the RSA modulus.</td>
</tr>
<tr>
<td>GetnByteLength</td>
<td>Returns the byte length of the RSA modulus.</td>
</tr>
<tr>
<td>GetPublicExponent</td>
<td>Extracts and copies the RSA key public exponent.</td>
</tr>
<tr>
<td>GetRsaKeyNameSection</td>
<td>Returns a pointer to the key name section which holds the name of the key.</td>
</tr>
<tr>
<td>GetRsaPrivateKeySection</td>
<td>Returns a pointer to the private key section of an RSA key token.</td>
</tr>
<tr>
<td>GetRsaPublicKeySection</td>
<td>Returns a pointer to the public key section of an RSA key token.</td>
</tr>
<tr>
<td>GetRsaPuKeyCertSection</td>
<td>Returns a pointer to the public key certificate section.</td>
</tr>
<tr>
<td>GetTokenLength</td>
<td>Returns the length of the specified token.</td>
</tr>
<tr>
<td>IsPrivateExponentEven</td>
<td>Verifies whether the RSA private exponent is an even valued integer.</td>
</tr>
<tr>
<td>IsPrivateKeyEncrypted</td>
<td>Verifies whether the private key section of the specified key token is encrypted.</td>
</tr>
<tr>
<td>IsPublicExponentEven</td>
<td>Verifies whether the RSA public exponent is an even valued integer.</td>
</tr>
<tr>
<td>IsRsaToken</td>
<td>Verifies whether the key token contains an RSA key.</td>
</tr>
<tr>
<td>IsS390Token</td>
<td>Verifies whether the key token is a valid type for S390.</td>
</tr>
<tr>
<td>IsTokenInternal</td>
<td>Identifies whether the key token is in operational format.</td>
</tr>
<tr>
<td>PkaHashQueryWithMK</td>
<td>Returns the master key version used to encrypt the specified token.</td>
</tr>
<tr>
<td>PkaMkvpQuery</td>
<td>Returns a value indicating which master key in the default set was used to encrypt the specified key token.</td>
</tr>
<tr>
<td>PkaMkvpQueryWithMK</td>
<td>Returns a value indicating which master key was used to encrypt the specified key token.</td>
</tr>
<tr>
<td>Function Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>pka96_tvvgen</td>
<td>Calculates the token validation value (TVV) for the specified key token. See page 9-34.</td>
</tr>
<tr>
<td>RecoverPkaClearKeyTokenUnderMk</td>
<td>Recovers the PKA clear key token under the master key of the default key set. See page 9-36.</td>
</tr>
<tr>
<td>RecoverPkaClearKeyTokenUnderXport</td>
<td>Recovers the PKA clear key token under a DES export key. See page 9-35.</td>
</tr>
<tr>
<td>RecoverPkaClearTokenUnderMkWithMK</td>
<td>Recovers the PKA clear key token under the master key. See page 9-36.</td>
</tr>
<tr>
<td>RecoverPkaClearTokenUnderMkWithMKWithMinModulus</td>
<td>Recovers the PKA clear key token under the specified master key. See page 9-36.</td>
</tr>
<tr>
<td>ReEncipherPkaKeyToken</td>
<td>Re-enciphers an internal PKA token from the old master key to the current master key of the default key set. See page 9-38.</td>
</tr>
<tr>
<td>ReEncipherPkaKeyTokenWithMK</td>
<td>Re-enciphers an internal PKA key token from the old master key to the current master key of a specified key set. See page 9-38.</td>
</tr>
<tr>
<td>RequestRSACrypto</td>
<td>Performs an RSA operation. See page 9-39.</td>
</tr>
<tr>
<td>sccRsaKeyGenerate</td>
<td>Generates an RSA key token in Secure Cryptographic Coprocessor format. See page 6-40.</td>
</tr>
<tr>
<td>store_KeyToken</td>
<td>Saves a public or private key to the SRDI. See page 9-41.</td>
</tr>
<tr>
<td>TokenMkvpmMatchMasterKey</td>
<td>Tests whether the key token was encrypted using a specified version of the master key. See page 9-42.</td>
</tr>
<tr>
<td>ValidatePkaToken</td>
<td>Verifies that the RSA key token is valid for use in the system. See page 9-43.</td>
</tr>
<tr>
<td>ValidatePkaTokenWithMinModulus</td>
<td>Verifies that the RSA key token is valid for use in the system, also checking for S390 minimum modulus lengths. See page 9-43.</td>
</tr>
<tr>
<td>VerifyKeyTokenConsistency</td>
<td>Verifies the consistency of a key token. See page 9-44.</td>
</tr>
<tr>
<td>verify_dSig</td>
<td>Verifies an RSA signature. See page 9-45.</td>
</tr>
</tbody>
</table>
Overview
An RSA key consists of a public modulus (n) which is the product of two large prime numbers, a public exponent (e) which is relatively prime to the modulus, and a private exponent (d). In the coprocessor, keys may be stored in CCA RSA tokens as in the key storage file or in scc (Secure Cryptographic Coprocessor) form, and are used in xCrypto internal tokens. Each form of key has a public and a private version.

The public version xCrypto internal token or SCC token of a key contains the modulus and the public exponent of the key, and the length of each. The private version may be in either modulus exponent or Chinese remainder format, and contains the modulus and public and private exponents for each. The xCrypto version of a key is used in the cryptographic engine for xcRSA requests and is the type returned by xcRSAKeyGenerate. SCC tokens are an intermediate version of the key which is used in the translation of keys from one token format to the other.

CCA RSA tokens consist of a token header, followed by:

1. An optional private key section which holds the decrypting information (the private key and the public modulus), verification data, and key-encryption data
2. A required public key section which holds encryption information (the public exponent, the modulus length, and, if there is no private section, the modulus itself.)
3. An optional public key certificate section may follow, or
4. Depending on the format, a separate section containing private-key blinding information, called the internal section.

Parts of the private key section may be encrypted under the master key (operational keys) or under a transport key (external keys). Only operational keys and public keys may be stored in the key-storage file.

The functions in this chapter can be separated into the following categories:

Informational: All of these functions operate on CCA RSA key tokens.

- `CalculatenWordLength`: Returns the length of the modulus in 16-bit words.
- `GetExponentLength`: Returns the length of the public exponent.
- `GetnBitLength`: Returns the length of the modulus in bits.
- `GetnByteLength`: Returns the length of the modulus in bytes.
- `GetTokenLength`: Returns the length of the CCA RSA key token.

Key checking

- `IsPrivateExponentEven`: Verifies whether the private exponent of the CCA RSA key token is even.
- `IsPrivateKeyEncrypted`: Verifies whether the private key section of the CCA RSA key token is encrypted.
- `IsPublicExponentEven`: Verifies whether the public exponent of the CCA RSA key token is even.
- `IsRsaToken`: Verifies whether the supplied token is an RSA token.
- `IsS390Token`: Verifies whether the key token is a valid type for zOS.
- `IsTokenInternal`: Verifies whether the CCA RSA key token has been encrypted with the master key (that is, an operational token).
- `PkaHashQueryWithMK`: Identifies which master key was used to encrypt the specified internal CCA RSA key token.
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PkaMkvpQuery</td>
<td>Identifies which master key of the default key set was used to encrypt the specified internal CCA RSA key token.</td>
</tr>
<tr>
<td>PkaMkvpQueryWithMK</td>
<td>Identifies which master key was used to encrypt the specified internal CCA RSA key token.</td>
</tr>
<tr>
<td>TokenMkvpMatchMasterKey</td>
<td>Tests whether the operational CCA RSA key token was encrypted with the specified master key.</td>
</tr>
<tr>
<td>ValidatePkaToken</td>
<td>Verifies that a CCA RSA key token is valid for use in the system.</td>
</tr>
<tr>
<td>ValidatePkaTokenWithMinModulus</td>
<td>Verifies that the RSA key token is valid for use in the system, also checking for S390 minimum modulus lengths.</td>
</tr>
<tr>
<td>VerifyKeyTokenConsistency</td>
<td>Tests the length fields of the CCA RSA key token, ensuring that they are consistent.</td>
</tr>
</tbody>
</table>

### Key manipulation

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateInternalKeyToken</td>
<td>Receives a clear CCA RSA key token and encrypts the private key data with the default master key, creating an operational CCA RSA key token.</td>
</tr>
<tr>
<td>CreateInternalKeyTokenWithMK</td>
<td>Receives a clear CCA RSA key token and encrypts the private key data, creating an operational CCA RSA key token.</td>
</tr>
<tr>
<td>CreateRsaInternalSection</td>
<td>Receives an xCrypto complete token and creates an internal CCA RSA section, containing the blinding information, including encrypting with the default master key.</td>
</tr>
<tr>
<td>CreateRsaInternalSectionWithMK</td>
<td>Receives an xCrypto complete token and creates an internal CCA RSA section, containing the blinding information, including encrypting with the master key.</td>
</tr>
<tr>
<td>ExtractRsaPublicSection</td>
<td>Formats a public key token from a private/public key token.</td>
</tr>
<tr>
<td>GenerateCcaRsaToken</td>
<td>Receives an xCrypto complete token and a CCA skeleton token and generates a CCA RSA token.</td>
</tr>
<tr>
<td>GenerateRsaInternalToken</td>
<td>Receives a CCA RSA key token and creates an xCrypto internal token.</td>
</tr>
<tr>
<td>GetModulus</td>
<td>Returns the public modulus of the CCA RSA key token.</td>
</tr>
<tr>
<td>GetPublicExponent</td>
<td>Returns the public exponent of the CCA RSA key token.</td>
</tr>
<tr>
<td>GetRsaKeyNameSection</td>
<td>Returns a pointer to the key name section which holds the name of the key.</td>
</tr>
<tr>
<td>GetRsaPrivateKeySection</td>
<td>Returns a pointer to the private key section of the CCA RSA key token.</td>
</tr>
<tr>
<td>GetRsaPublicKeySection</td>
<td>Returns a pointer to the public key section of the CCA RSA key token.</td>
</tr>
<tr>
<td>GetRsaPuKeyCertSection</td>
<td>Returns a pointer to the public key certificate section.</td>
</tr>
<tr>
<td>pka96_tvvgen</td>
<td>Calculates the token validation value for the CCA RSA key token.</td>
</tr>
<tr>
<td>RecoverPkaClearKeyTokenUnderMk</td>
<td>Decrypts an operational CCA RSA key token with the default master key.</td>
</tr>
<tr>
<td>RecoverPkaClearKeyTokenUnderXport</td>
<td>Decrypts an external CCA RSA key token.</td>
</tr>
<tr>
<td>RecoverPkaClearTokenUnderMkWithMK</td>
<td>Decrypts an operational CCA RSA key token.</td>
</tr>
</tbody>
</table>
RecoverPkaClearTokenUnderMkWithMKWithMinModulus
ReEncipherPkaKeyToken
ReEncipherPkaKeyTokenWithMK

RecoverPkaClearTokenUnderMkWithMKWithMinModulus
ReEncipherPkaKeyToken
ReEncipherPkaKeyTokenWithMK

Key usage

generate_dsig
RequestRSACrypto
verify_dSig

Key usage

generate_dsig
RequestRSACrypto
verify_dSig

generate_dsig
RequestRSACrypto
verify_dSig

RequestRSACrypto
Verifies an RSA signature.

generate_dsig
RequestRSACrypto
verify_dSig

generate_dsig
RequestRSACrypto
verify_dSig

generate_dsig
RequestRSACrypto
verify_dSig

Key storage

delete_KeyToken
ggetKeyToken
store_KeyToken
delete_KeyToken
ggetKeyToken
store_KeyToken

Key storage

delete_KeyToken
ggetKeyToken
store_KeyToken

delete_KeyToken
ggetKeyToken
store_KeyToken

Key storage

delete_KeyToken
ggetKeyToken
store_KeyToken

Key storage

delete_KeyToken
ggetKeyToken
store_KeyToken

Key storage

delete_KeyToken
ggetKeyToken
store_KeyToken

Key storage

delete_KeyToken
ggetKeyToken
store_KeyToken

Key storage

delete_KeyToken
ggetKeyToken
store_KeyToken

Key storage

delete_KeyToken
ggetKeyToken
store_KeyToken

Key storage

delete_KeyToken
ggetKeyToken
store_KeyToken

Key storage

delete_KeyToken
ggetKeyToken
store_KeyToken

Key storage

delete_KeyToken
ggetKeyToken
store_KeyToken
**CalculatenWordLength - Return Word Length of Modulus**

CalculatenWordLength returns the length of the modulus in terms of the number of 16-bit words it occupies.

**Function Prototype**

```c
USHORT CalculatenWordLength ( RsaKeyTokenHeader *pToken )
```

**Input**

On entry to this routine:

- `pToken` is a pointer to the key token.

**Output**

On successful exit from this routine:

- `CalculatenWordLength` returns the length of the modulus in 16-bit words.

**Return Codes**

This function has no return codes.
**CreateInternalKeyTokenWithMK - Create Operational Key Token**

CreateInternalKeyTokenWithMK receives a clear CCA RSA key token and creates the internal form by encrypting the private key areas under the master key for the mk_set being processed.

**Function Prototype**

```c
long CreateInternalKeyTokenWithMK ( RsaKeyTokenHeader *pTokenIn,
                                     mk_selectors *pMKSelector,
                                     RsaKeyTokenHeader *pTokenOut )
```

```c
long CreateInternalKeyToken ( RsaKeyTokenHeader *pTokenIn,
                              RsaKeyTokenHeader *pTokenOut )
```

CreateInternalKeyToken has the same effect as calling CreateInternalKeyTokenWithMK after setting the pMKSelector parameter to (MK_SET_DEFAULT, current_mk, ASYM_MK).

**Input**

On entry to this routine:

pTokenIn is a pointer to the cleartext key token.

pMKSelector is a parameter of type mk_selectors, indicating which set of master keys to use in this function. This variable must be initialized as follows:

- mk_set is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- mk_register is ignored.
- type_mks should be set to ASYM_MK.

**Output**

On successful exit from this routine:

pTokenOut contains the encrypted operational key token.

**Return Codes**

Common return codes generated by this routine are:

- **ERROR** The token is not an RSA token, or does not have a private section.
- **mk_KEY_NOT_VALID** The current master key is not valid.
- **mk_SEM_CLAIM_FAILED** Could not access the master keys.
CreateRsaInternalSectionWithMK - Create RSA Internal Section
CreateRsaInternalSectionWithMK receives an scc (secure cryptographic coprocessor) internal token and creates an internal CCA RSA key token section by copying the blinding values and encrypting under the master key for the mk_set being processed.

Note: This function is only useful for type 2 and 5 key tokens. Other tokens do not have internal sections.

Function Prototype

long CreateRsaInternalSectionWithMK ( RsaKeyTokenHeader *pTokenOut,  
mk_selectors *pMKSelector,  
sccRSAKeyToken_t *pRsaTokenIn )
long CreateRsaInternalSection ( RsaKeyTokenHeader *pTokenOut,  
sccRSAKeyToken_t *pRsaTokenIn )

CreateRsaInternalSection has the same effect as calling CreateRsaInternalSectionWithMK after setting the pMKSelector parameter to (MK_SET_DEFAULT, current_mk, ASYM_MK).

Input
On entry to this routine:

pTokenOut is a pointer to a variable CCA RSA key token whose private section is either type 2 or type 5.

pMKSelector is a parameter of type mk_selectors, indicating which set of master keys to use in this function. This variable must be initialized as follows:

• mk_set is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
• mk_register is ignored.
• type_mks should be set to ASYM_MK.

pRsaTokenIn is a pointer to the internal scc key structure.

Output
This function returns no output. On successful exit from this routine:

The internal section of the CCA RSA key token is created.

pTokenOut contains the extended CCA RSA token.

Return Codes
Common return codes generated by this routine are:

OK The operation was successful.
mk_KEY_NOT_VALID The current master key is not valid.
mk_SEM_CLAIM_FAILED Could not access the master keys.
**delete_KeyToken - Delete a Key From On-Board Storage**
delete_KeyToken permanently removes a registered public key or retained private key from storage in the coprocessor.

**Function Prototype**
long delete_KeyToken ( char *pKeyName )

**Input**
On entry to this routine:

pKeyName is a pointer to a 64 byte array containing the name of the key to be deleted.

**Output**
This function returns no output. On successful exit from this routine:

The key referenced by pKeyName is no longer in storage, and the key storage SRDI has been resized.

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

OK The operation was successful.

**KEY_NAME_NOT_FOUND** The key was not found in the list.

**CP_MEMORY_NAVAIL** Out of memory error.

**PKEY_SRDI_ERROR** Unable to access the key storage SRDI.
**ExtractRsaPublicSection - Format public key token from private/public key token**

ExtractRsaPublicSection takes a CCA RSA key token and extracts the public section, building a CCA RSA public key from the provided token.

**Note:** In a CCA RSA key token with a private key, the public modulus is stored in the private key section. This function copies the modulus into the public key section of the new key token, along with the public exponent and appropriate length fields.

**Function Prototype**

```c
long ExtractRsaPublicSection ( RsaKeyTokenHeader *pTarget,
                               RsaKeyTokenHeader *pSource)
```

**Input**

On entry to this routine:

- `pSource` is a pointer to a buffer containing a CCA RSA key token.
- `pTarget` is a pointer to a buffer large enough to contain the public token.

**Output**

On successful exit from this routine:

- `pTarget` contains a CCA public key token.

**Return Codes**

Common return codes generated by this routine are:

- `ERROR` Unable to access the public key section of the key.
GenerateCcaRsaToken - Generate CCA RSA Key Token

GenerateCcaRsaToken generates a CCA RSA key token from an internal scc (secure cryptographic coprocessor) key token and a CCA PKA skeleton token. The skeleton token must be initialized to indicate the required format of the final token.

Note: The toolkit does not have a function to generate a CCA RSA token from an xCrypto format key token. To use this function, you must generate the initial key token using sccRSAKeyGenerate.

Function Prototype

long GenerateCcaRsaToken ( RsaKeyTokenHeader *pPkaToken, 
sccRSAKeyToken_t *pRsaKeyToken) 
short int internal

Input

On entry to this routine:

pPkaToken must be a pointer to a CCA RSA key token header whose nextSection field contains one of the following:

- RSA_PRIVATE_SECTION_NOPT - for a version 0 modulus exponent key with a fixed modulus.
- RSA_PRIVATE_SECTION_CR - for a version 0 Chinese Remainder key.
- RSA_PRIVATE_SECTION_NOPT_VAR - for a version 0 key with a variable length modulus field.
- RSA_PRIVATE_SECTION_NOPT_NEW - for a version 1 modulus exponent key.
- RSA_PRIVATE_SECTION_CR_NEW - for a version 1 Chinese Remainder key.

pRsaKeyToken must be a pointer to a valid internal scc RSA key token.

internal must be initialized to TRUE if the key is to be an operational key, or FALSE otherwise.

Output

On successful exit from this routine:

pPkaToken contains a valid CCA RSA token of the type desired.

Return Codes

Common return codes generated by this routine are:

OK The operation was successful.
ERROR The skeleton token was not initialized.
**GenerateRsaInternalToken - Generate RSA Key Token**

GenerateRsaInternalToken receives a CCA RSA key token and creates an scc (secure cryptographic coprocessor) token, which may be translated to an xCrypto token for use in RSA computations.

**Function Prototype**

```c
long GenerateRsaInternalToken ( RsaKeyTokenHeader *pPkaTokenIn,
                                sccRSAKeyToken_t  *pRsaKeyTokenOut )
```

**Input**

On entry to this routine:

- `pPkaTokenIn` is a pointer to the CCA RSA key token.

**Output**

On successful exit from this routine:

- `pRsaKeyTokenOut` is a pointer to the location where the function stores the internal scc complete key token it creates from the specified CCA RSA token.

**Return Codes**

Common return codes generated by this routine are:

- **OK** The operation was successful.
- **ERROR** The input key token is not an RSA key token.
**generate_dSig - Generate Digital Signature**

generate_dSig receives an RSA key token in operational format and a buffer of data, with the length of the data and the expected length of the digital signature. The key token is deciphered and the input data is hashed with SHA-1, then the data is formatted according to the requested Type before signing with the clear key. The format may be one of ISO-9796, PKCS #1 block type 0 or 1, or zero-padded.

**Function Prototype**

```c
long generate_dSig ( RsaKeyTokenHeader *pTokenIn,
                      UCHAR *pDataIn,
                      long DataLength,
                      UCHAR *pSignatureOut,
                      USHORT *pSignatureBitLength,
                      UCHAR SignatureType )
```

**Input**

On entry to this routine:

- `pTokenIn` is a pointer to the operational key token.
- `pDataIn` is a pointer to the data which is to be signed.
- `DataLength` is the length of the data to be signed, in bytes.
- `pSignatureOut` is a pointer to a buffer which is to hold the returned signature.
- `pSignatureBitLength` is a pointer to the length of the buffer `pSignatureOut`, in bits.
- `SignatureType` is one of the following:
  - `M_ISO9796` if the data is to be formatted according to the ISO-9796 standard before signing.
  - `M_PKCS1` if the data is to be formatted as specified in the RSA Data Security, Inc., Public Key Cryptography Standards #1 block type 00 before signing.
  - `M_PKCS11` if the data is to be formatted as specified in the RSA Data Security, Inc., Public Key Cryptography Standards #1 block type 01 before signing.
  - `M_ZEROPAD` if the Data is to be placed in the low-order bits of a bit-string of the same length as the modulus with all other bit-positions set to zero before signing.

**Output**

On successful exit from this routine:

- `pSignatureBitLength` contains the length (in bits) of the calculated digital signature.
- `pSignatureOut` contains the digital signature.

**Return Codes**

Common return Codes generated by this routine are:

- **OK** The operation was successful.
- **E_SIZE** The provided buffer was not large enough to contain the signature.
- **PKABadAddr** The key token is not valid.
**GeteLength - Return RSA Public Exponent Byte Length**

GeteLength returns the byte length of the RSA public exponent field, as contained in the member field of the key token.

**Function Prototype**

```c
USHORT GeteLength ( RsaKeyTokenHeader *pToken )
```

**Input**

On entry to this routine:

- `pToken` is a pointer to the key token.

**Output**

On successful exit from this routine:

- `GeteLength` returns the byte length of the RSA public exponent field.

**Return Codes**

This function has no return codes.
**getKeyToken - Get a PKA Token From On-Board Storage**

getKeyToken retrieves a PKA retained private key or registered public key from the SRDI where it is stored.

**Function Prototype**

```c
long getKeyToken ( char *pLabel,
                  char *pKey,
                  USHORT *pFlags )
```

**Input**

On entry to this routine:

- `pLabel` is a pointer to a string containing the label associated with the requested key.

- `pKey` is a pointer to a buffer in which the key token can be written. The maximum length required is 2500 bytes.

- `pFlags` is a pointer to a 2-byte buffer which can hold returned flags from the key token.

**Output**

On successful exit from this routine:

- `pKey` contains the clear key token associated with the label at `pLabel`.

- `pFlags` contains the flags associated with the key.

**Return Codes**

Common return codes generated for this function are:

- `srdiv_NO_ERROR` The command completed successfully.
- `PKEY_NOT_REGISTER` The key was not found.
- `PKEY_SRDI_ERROR` The registered key manager could not be accessed.
**GetModulus - Extract and Copy RSA Modulus**

GetModulus extracts the RSA key modulus from the specified key token, and copies it to the buffer provided.

**Function Prototype**

```c
void GetModulus ( RsaKeyTokenHeader *pToken,
                  UCHAR *pModulus )
```

**Input**

On entry to this routine:

- `pToken` is a pointer to the key token.
- `pModulus` is a pointer to a buffer for the modulus.

**Output**

On successful exit from this routine:

- `pModulus` is a pointer to the provided buffer where the RSA key modulus is stored.

**Return Codes**

This function has no return codes.
GetnBitLength - Return RSA Modulus Bit Length
GetnBitLength returns the bit length of the RSA modulus as contained in the member field of the key token.

Function Prototype
USHORT GetnBitLength ( RsaKeyTokenHeader *pToken )

Input
On entry to this routine:

pToken is a pointer to the key token.

Output
On successful exit from this routine:

GetnBitLength returns the bit length of the RSA modulus.

Return Codes
This function has no return codes.
GetnByteLength - Return RSA Modulus Byte Length
GetnByteLength returns the length of the RSA modulus, in bytes.

Note: The key token contains a member field which indicates the modulus byte length. This field may not be the actual byte length, but is an indication of the length of the field containing the modulus. This function returns the actual byte length of the modulus by calculating it from the bit length. It does not use the byte length member field from the key token.

Function Prototype
USHORT GetnByteLength ( RsaKeyTokenHeader *pToken )

Input
On entry to this routine:

pToken is a pointer to the key token.

Output
On successful exit from this routine:

GetnByteLength returns the length of the RSA modulus, in bytes.

Return Codes
This function has no return codes.
**GetPublicExponent - Extract and Copy Public Exponent**

GetPublicExponent extracts the RSA key public exponent from the specified key token, and copies it to the provided buffer.

**Function Prototype**

```c
USHORT GetPublicExponent ( RsaKeyTokenHeader *pToken, UCHAR *pDest )
```

**Input**

On entry to this routine:

- `pToken` is a pointer to the key token.
- `pDest` is a pointer to the buffer provided for the exponent.

**Output**

On successful exit from this routine:

- `pDest` contains the RSA key public exponent.
- `GetPublicExponent` returns the length of the exponent.

**Return Codes**

This function has no return codes.
GetRSAKeyNameSection - Get the Section Containing the Name of the Key
GetRSAKeyNameSection returns a pointer to the section of the token containing the key name.

Function Prototype
void * GetRsaKeyNameSection ( RsaKeyTokenHeader *pRsaKeyToken )

Input
On entry to this routine:

pRsaKeyToken is a pointer to a valid CCA RSA key token.

Output
On successful exit from this routine:

GetRsaKeyNameSection returns a pointer to the key name section, or NULL if there is no such section.

Return Codes
This function has no return codes.
GetRsaPrivateKeySection - Return Private Key
GetRsaPrivateKeySection returns a pointer to the private key section of an RSA key token, if it is present. Otherwise, the function returns a null pointer.

Function Prototype
void GetRsaPrivateKeySection ( RsaKeyTokenHeader   *pToken )

Input
On entry to this routine:

pToken is a pointer to the CCA RSA key token.

Output
This function returns no output. On successful exit from this routine:

GetRsaPrivateKeySection returns a pointer to the private key section of a CCA key token.

Notes
Refer to Appendix B of the *IBM 4764 PCI-X Cryptographic Coprocessor CCA Basic Services Reference and Guide* for a diagram of the key token structure.

A typical RSA key token looks similar to the following:

```
+-----------------+              +-----------------+
| Header          |              | Header          |
|                 |              |                 |
| Private Section | or            | Public Section  |
|                 |              |                 |
| Public Section  |              |                 |
```

Return Codes
This function has no return codes.
GetRsaPublicKeySection - Return Public Key
GetRsaPublicKeySection returns a pointer to the public key section of an RSA key token, if it is present. If not, the function returns a null pointer.

Note: If no public key section is present an internal error has occurred, since all RSA tokens should contain a public key section.

Function Prototype
void GetRsaPublicKeySection ( RsaKeyTokenHeader *pToken )

Input
On entry to this routine:

pToken is a pointer to the RSA key token.

Output
On successful exit from this routine:

GetRsaPublicKeySection returns a pointer to the public key section of an RSA key token.

Return Codes
This function has no return codes.
GetRsaPuKeyCertSection - Return Public Key Certificate
GetRsaPuKeyCertSection returns a pointer to the public key section of an RSA key token, if it is present. If not, the function returns a null pointer.

Function Prototype
void GetRsaPuKeyCertSection ( RsaKeyTokenHeader *pToken )

Input
On entry to this routine:

pToken is a pointer to the RSA key token.

Output
On successful exit from this routine:

GetRsaPuKeyCertSection returns a pointer to the public key section of an RSA key token.

Return Codes
This function has no return codes.
**GetTokenLength - Return Key Token Length**
GetTokenLength returns the length of the specified token, as contained in the member field of the header.

**Function Prototype**
USHORT GetTokenLength ( RsaKeyTokenHeader *pToken )

**Input**
On entry to this routine:

*pToken is a pointer to the key token.

**Output**
On successful exit from this routine:

GetTokenLength returns the length of the specified token.

**Return Codes**
This function has no return codes.
IsPrivateExponentEven - Verify RSA Private Exponent

IsPrivateExponentEven returns TRUE if the private exponent in the specified key token is an even valued integer; otherwise, it returns FALSE.

Function Prototype

boolean IsPrivateExponentEven ( RsaKeyTokenHeader *pToken )

Input
On entry to this routine:

pToken is a pointer to the key token.

Output
On successful exit from this routine:

IsPrivateExponentEven returns TRUE if the private exponent in the specified key token is an even valued integer, and FALSE if it is not.

Return Codes
This function has no return codes.
**IsPrivateKeyEncrypted - Verify Private Key Encryption**

IsPrivateKeyEncrypted returns TRUE if the private key section of the specified PKA key token is in encrypted form, or FALSE if not.

**Function Prototype**

```c
boolean IsPrivateKeyEncrypted ( RsaKeyTokenHeader *pToken )
```

**Input**

On entry to this routine:

- `pToken` is a pointer to the key token.

**Output**

On successful exit from this routine:

- `IsPrivateKeyEncrypted` returns TRUE if the private key section of the specified PKA key token is in encrypted form, and FALSE if it is not.

**Return Codes**

This function has no return codes.
**IsPublicExponentEven - Verify RSA Public Exponent**

IsPublicExponentEven returns TRUE if the public exponent in the specified key token is an even valued integer; otherwise, it returns FALSE.

**Function Prototype**

```c
boolean IsPublicExponentEven ( RsaKeyTokenHeader *pToken )
```

**Input**

On entry to this routine:

- `pToken` is a pointer to the key token.

**Output**

On successful exit from this routine:

- `IsPublicExponentEven` returns TRUE if the public exponent in the specified key token is an even valued integer, and FALSE if it is not.

**Return Codes**

This function has no return codes.
**IsRsaToken - Verify RSA Key**

IsRsaToken returns TRUE if the specified key token contains an RSA key, or FALSE if it does not.

**Function Prototype**

boolean IsRsaToken ( RsaKeyTokenHeader *pToken )

**Input**

On entry to this routine:

pToken is a pointer to the key token.

**Output**

On successful exit from this routine:

IsRsaToken returns TRUE if the specified key token contains an RSA key, and FALSE if it is not an RSA key token.

**Return Codes**

This function has no return codes.
**IsS390Token - Verify zOS Compatibility**
IsS390Token verifies that the key provided is of a type which may be used in a zOS system.

**Note:** This function may only be used on a coprocessor with the zOS load.

**Function Prototype**
boolean IsS390Token ( RsaKeyTokenHeader *pToken )

**Input**
On entry to this routine:

*pToken is a pointer to the key token.

**Output**
On successful exit from this routine:

IsS390Token returns TRUE if the specified key token contains an RSA key, and FALSE if it is not an RSA key token.

**Return Codes**
This function has no return codes.
IsTokenInternal - Key Token Format

IsTokenInternal returns TRUE if the specified key token is in operational format, or FALSE if it is in external format.

Function Prototype

boolean IsTokenInternal ( RsaKeyTokenHeader *pToken )

Input

On entry to this routine:

pToken is a pointer to the key token.

Output

On successful exit from this routine:

IsTokenInternal returns TRUE if the specified key token is in operational format, or FALSE if it is in external format.

Notes

Operational key tokens contain private key information that has been multiply-enciphered with the master key. RecoverPkaClearTokenUnderMkWithMK is used to decipher an internal key token so that it may be used.

Return Codes

This function has no return codes.
**PkaHashQueryWithMK - Return Master Key Version**
PkaHashQueryWithMK returns a value indicating which master key was used to encrypt the specified key token for the mk_set being processed.

**Function Prototype**

```c
MK_VERSION PkaHashQueryWithMK ( RsaKeyTokenHeader *pToken,
                                mk_selectors *pMKSelector )
```

**Input**

On entry to this routine:

- `pToken` is a pointer to a variable which will hold the new CCA RSA key token.
- `pMKSelector` is a parameter of type `mk_selectors`, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is ignored.
  - `type_mks` should be set to `ASYM_MK`.

**Output**

On successful exit from this routine:

- `PkaHashQueryWithMK` returns the version of the master key (`MK_CURRENT`, `MK_OLD`, or `MK_OUT_OF_DATE`) that was used to encrypt the specified key token. If the value returned was not `MK_OUT_OF_DATE`, `pMKSelector->mk_register` is set to either `old_mk` or `current_mk`, whichever is appropriate.

**Return Codes**

Common return codes generated by this routine are:

- **OK** The operation was successful.
- **mk_KEY_NOT_VALID** The current master key is not valid.
- **mk_SEM_CLAIM_FAILED** Could not access the master keys.
**PkaMkvpQueryWithMK - Return Master Key Version**

PkaMkvpQueryWithMK returns a value indicating which master key was used to encrypt the specified key token for the mk_set being processed.

**Note:** This function works only with version 0 tokens.

**Function Prototype**

\[
\begin{align*}
\text{MK_VERSION PkaMkvpQueryWithMK ( RsaKeyTokenHeader *pToken, } \\
\text{mk_selectors *pMKSelector )}
\end{align*}
\]

\[
\begin{align*}
\text{MK_VERSION PkaMkvpQuery ( RsaKeyTokenHeader *pToken )}
\end{align*}
\]

PkaMkvpQuery has the same effect as calling PkaMkvpQueryWithMK after setting the pMKSelector parameter to \{MK_SET_DEFAULT, current_mk, ASYM_MK\}.

**Input**

On entry to this routine:

- **pToken** is a pointer to the key token that is checked.

- **pMKSelector** is a parameter of type mk_selectors, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  
  - **mk_set** is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
  
  - **mk_register** is ignored.
  
  - **type_mks** should be set to ASYM_MK.

**Output**

On successful exit from this routine:

- **PkaMkvpQueryWithMK** returns the version of the master key (MK_CURRENT, MK_OLD, or MK_OUT_OF_DATE) that was used to encrypt the specified key token. If the value returned was not MK_OUT_OF_DATE, pMKSelector->mk_register is set to either old_mk or current_mk, whichever is appropriate.

**Return Codes**

This function has no return codes.
**pka96_tvvgen - Calculate Token Validation Value**
pka96_tvvgen calculates the token validation value (TVV) for the specified key token.

**Function Prototype**
```c
void pka96_tvvgen ( USHORT  token_len,
                    UCHAR   *key_token_ptr,
                    ULONG   *tvv )
```

**Input**
On entry to this routine:

- `token_len` is the length of the token specified with parameter `key_token_ptr`.
- `key_token_ptr` is a pointer to the key token whose TVV is calculated.

**Output**
On successful exit from this routine:

- `tvv` contains the calculated TVV.

**Return Codes**
This function has no return codes.
RecoverPkaClearKeyTokenUnderXport - Recovers the PKA Clear Key Token Under a DES Export Key

RecoverPkaClearKeyTokenUnderXport receives a PKA key token which is encrypted under a DES transport key, and recovers the clear form by decrypting the private key areas and then verifying the SHA-1 hashes contained in those areas.

Function Prototype

```c
long RecoverPkaClearKeyTokenUnderXport(RsaKeyTokenHeader *pTokenIn,
                                        double_length_key *desKey,
                                        RsaKeyTokenHeader *pTokenOut)
```

Input

On entry to this routine:

- `pTokenIn` is a pointer to the encrypted key token.
- `desKey` is a pointer to 16 bytes of clear key information, which constitute the double length key used to encipher the RSA key.
- `pTokenOut` is a pointer to a location which can store a key token.

Output

On successful exit from this routine:

- `pTokenOut` contains the cleartext key token that it recovers.

Return Codes

Common return codes generated by this routine are:

- **OK**  The operation was successful.
- **ERROR**  The operation failed.
RecoverPkaClearTokenUnderMkWithMKWithMinModulus - Recovers the PKA clear key token

RecoverPkaClearKeyTokenUnderMkWithMKWithMinModulus receives a PKA key token which is encrypted under the master key for the mk_set which is currently in use. If the key is in the on-board cache of decrypted keys, this key is returned to the calling function. Otherwise, the clear form of the key is recovered by decrypting the private areas of the key and verifying the SHA-1 hashes of those sections. The allowable minimum modulus length is also verified. The clear key is then added to the on-board cache before being returned to the calling function.

Function Prototype

```c
long RecoverPkaClearTokenUnderMkWithMKWithMinModulus( RsaKeyTokenHeader *pTokenIn, RsaKeyTokenHeader *pTokenOut, mk_selectors *pMKSelector, long *pMsg, long signal )

long RecoverPkaClearTokenUnderMkWithMK( RsaKeyTokenHeader *pTokenIn, RsaKeyTokenHeader *pTokenOut, mk_selectors *pMKSelector, long *pMsg )

long RecoverPkaClearKeyTokenUnderMk ( RsaKeyTokenHeader *pTokenIn, RsaKeyTokenHeader *pTokenOut, long *pMsg )
```

RecoverPkaClearKeyTokenUnderMk has the same effect as calling RecoverPkaClearTokenUnderMkWithMKWithMinModulus after setting the signal parameter to ORIGINAL_MIN.

RecoverPkaClearKeyTokenUnderMk has the same effect as calling RecoverPkaClearTokenUnderMkWithMK after setting the pMKSelector parameter to {MK_SET_DEFAULT, current_mk, ASYM_MK}.

**Input**

On entry to this routine:

- pTokenIn is a pointer to the encrypted key token.
- pMKSelector is a parameter of type mk_selectors, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - mk_set is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
  - mk_register is ignored.
  - type_mks should be set to ASYM_MK.

**Output**

On successful exit from this routine:

- pTokenOut is a pointer to the location which contains the decrypted key token.
- pMsg is the error code.
- signal is a constant indicating which type of function the key will be used in:
• **ORIGINAL_MIN** - Minimum bit length of 512 bits
• **PKE_DSV_MIN** - Minimum bit length of 128 bits
• **PKD_MIN** - Minimum bit length of 128 bits for external keys; otherwise, 512 bits is minimum.

**Notes**

RecoverPkaClearTokenUnderMkWithMkWithMinModulus determines which master key was used to encipher the PKA key.

This function does not change the value of byte 28 of the private key, the Key Format and Security byte. If you are planning to store this key in clear form, you should change this byte to the appropriate value before storing. Refer to Appendix B of the *CCA Basic Services Reference and Guide* for the appropriate values for different RSA key token formats.

**Return Codes**

Common return codes generated by this routine are:

- **OK** The operation was successful.
- **ERROR** Returns an xCrypto error indicating the cause of the error.
- **RT_TKN_UNUSEABLE** The token was not an RSA token.
- **RT_KEY_INV_MKVN** The key was encrypted with an invalid master key.
- **mk_SRDI_OPEN_ERROR** Could not open the master key.
- **mk_SEM_CLAIM_FAILED** Unable to access the SRDI Manager.
ReEncipherPkaKeyTokenWithMK - Re-Encipher PKA Key Token
ReEncipherPkaKeyTokenWithMK re-enciphers an internal PKA key token from the old master key to the current master key for the mk_set being processed.

Function Prototype

```c
long ReEncipherPkaKeyTokenWithMK ( RsaKeyTokenHeader *pToken,
     mk_selectors *pMKSelector,
     UCHAR *pWorkArea )
```

```c
long ReEncipherPkaKeyToken ( RsaKeyTokenHeader *pToken,
     UCHAR *pWorkArea )
```

ReEncipherPkaKeyToken has the same effect as calling ReEncipherPkaKeyTokenWithMK after setting the pMKSelector parameter to (MK_SET_DEFAULT, current_mk, ASYM_MK).

Input
On entry to this routine:

pToken is a pointer to the input key token, enciphered under the old master key.

pMKSelector is a parameter of type mk_selectors, indicating which set of master keys to use in this function. This variable must be initialized as follows:

- mk_set is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- mk_register must be set to new_mk or current_mk.
- type_mks should be set to ASYM_MK.

pWorkArea is a pointer to a variable which can hold a private key. This is used as a work area when decrypting.

Output
On successful exit from this routine:

pToken contains the key token, which has been enciphered under the master key specified by the pMKSelector->mk_register.

Return Codes
Common return codes generated by this routine are:

- OK  The operation was successful.
- ERROR  The input token is not an RSA token.
- FALSE  Unable to verify the current master key.
RequestRSACrypto - Perform an RSA Operation

RequestRSACrypto converts the specified CCA RSA key token to the RSA internal key token format that the RSA engine requires, and then requests that the RSA engine perform the specified RSA function.

Note: Prior to using this routine, ensure that you've deciphered the private key (if you're using it) using the routine RecoverPkaClearTokenUnderMkWithMk().

Function Prototype

long RequestRSACrypto ( void *pInput,
                        RsaKeyTokenHeader *pKeyToken,
                        void *pOutput,
                        ULONG  DataBitLength,
                        ULONG  RsaOperation )

Input

On entry to this routine:

pInput is a pointer to the input data for the RSA operation.

pKeyToken is a pointer to the key token for the RSA key. This is a clear CCA format RSA key token.

DataBitLength is the length of the input data, in bits. This number is presumed to be equal to the length of the output data buffer, in bits. If this number is larger than the modulus length in bits, the data which will be operated on is in the rightmost modulus length bits of the input data buffer, and the result will be placed in the rightmost modulus length bits of the output data buffer.

RsaOperation is the requested RSA operation, such as RSA_ENCRYPT (public key operation) or RSA_DECRYPT (private key operation).

Output

On successful exit from this routine:

pOutput is a pointer to a buffer that receives the results of the requested RSA operation.

Return Codes

Common return codes generated by this routine are:

OK The operation was successful.

ERROR Could not create a buffer to receive the RSA key token.

E_SIZE The data is larger than the modulus.

PKABadAddr The key token is not valid.

PKANoSpace Unable to allocate sufficient memory.
sccRSAKeyGenerate - Generate an SCC Key Token
sccRSAKeyGenerate will generate an scc format RSA key token, which may then be used to build a CCA RSA key token.

Note: There are no functions provided to build a CCA RSA Key Token from an xCrypto format key token. This function calls the xcRSAKeyGenerate function and translates the resulting key to sccRSAKeyToken_t format for use in the other functions in this chapter.

Function Prototype
long sccRSAKeyGenerateAsync ( sccRSAKeyGen_RB_t  *psccRB,
                               unsigned long    *pmsg_id
)

Input
On entry to this routine:

psccRB is a pointer to an SCC key generate request block, whose fields must be formatted as follows:

- regen_data is a pointer to regeneration data provided to allow the generation of the same key multiple times for testing.
- key_token is a pointer to a buffer large enough to hold the generated key (2500 bytes maximum).
- regen_size is the length of the provided regeneration data. Zero is a valid length.
- key_size is a pointer to a buffer holding the length in bytes of the buffer which is pointed to by key_token.
- mod_size is the length in bits of the requested modulus.
- public_exp is one of the following:
  - RSA_EXPONENT_RANDOM - The public exponent will be randomly generated.
  - RSA_EXPONENT_3 - The public exponent will be given a value of 0x03.
  - RSA_EXPONENT_65537 - The public exponent will have a value of 0x010001.
  - RSA_EXPONENT_2 - The public exponent will have a value of 0x02.
  - RSA_EXPONENT_FIXED - The public exponent is defined in the buffer pointed to by key_token.
- key_type is one of the following:
  - RSA_PRIVATE_CHINESE_REMAINDER - Chinese Remainder format.
  - RSA_PRIVATE_MODULUS_EXPONENT - Modulus exponent format with a private key.
  - RSA_X931_PRIVATE_CHINESE_REMAINDER - Restricted Chinese Remainder format key.
  - RSA_X931_PRIVATE_MODULUS_EXPONENT - Restricted modulus exponent format key.
  - RSA_PKCS_PRIVATE_CHINESE_REMAINDER
  - RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER

Output
On successful exit from this routine:

psccRB->key_token contains a new RSA key token with the exponent requested and in the format requested.

Return Codes
Common return codes generated by this routine are:

E_ALLOCATE_MEM Unable to allocate working memory.

Other return codes may be returned from xcRSAKeyGenerate.
store_KeyToken - Store Registered or Retained Key

store_KeyToken saves a registered public key or retained private key to the key retain SRDI on the coprocessor. Once stored in this area, a key may not be changed except by deleting with delete_KeyToken.

Function Prototype

long store_KeyToken ( KEY_register_data_t *pKey )

Input

On entry to this routine:

pKey is a pointer to a KEY_register_data_t, whose fields must be initialized as follows:

- version - The version of the key token stored in this record. Legal values are 0 and 1.
- reservd - This short variable must be initialized to 0.
- length - The length of this record, in big-endian format.
- label - Contains a 64-byte key name.
- flags - Valued to CCA_CLONE if this key is allowed to participate in master key cloning operations, or 0 otherwise.
- keydata - The beginning of the actual key token.

Output

This function returns no output. On successful exit from this routine:

The KeyRetain SRDI has been expanded to include the data stored in pKey.

Return Codes

Common return codes generated by this routine are:

- srdi_NO_ERROR The operation was successful.
- CP_MEMORY_NAVAIL Out of memory error.
- PK_SRDI_ERROR Unable to access the key storage SRDI.
- DUPLICATE_NAME A key with the same name is already registered.
**TokenMkvpMatchMasterKey - Test Encryption of RSA Key**

TokenMkvpMatchMasterKey tests whether the specified key token was encrypted using a specified version of the master key. The Master Key Verification Pattern (MKVP) of the specified key token is compared to the MKVP for the specified master key. If the two are equal, the function returns TRUE; if not, it returns FALSE.

**Function Prototype**

```c
boolean TokenMkvpMatchMasterKey( mk_selectors *mk_selector,
                                 RsaKeyTokenHeader   *pToken )
```

**Input**

On entry to this routine:

- `mk_selector` is a pointer to a variable which must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is set to either `old_mk`, `current_mk`, or `new_mk`, representing the key which should be tested.
  - `type_mks` should be set to `ASYM_MK`.

- `pToken` is a pointer to the key token that you want to test.

**Output**

On successful exit from this routine:

- `TokenMkvpMatchMasterKey` returns TRUE if the MKVP of the specified key token is equal to the MKVP for the specified master key, and FALSE if it is not.

**Return Codes**

This function has no return codes.
**ValidatePkaTokenWithMinModulus - Validate RSA Key Token**

ValidatePkaTokenWithMinModulus accepts a cleartext RSA key token, and verifies that the token is valid for use in the system.

**Function Prototype**

```c
long ValidatePkaTokenWithMinModulus( RsaKeyTokenHeader *pToken,
                                         long *pErrorCode )
```

**Input**

On entry to this routine:

- `pToken` is a pointer to the RSA key token.
- `pErrorCode` is a pointer to the location where the function stores an error code, if a critical error occurs.

`signal` is a value indicating the type of function the RSA key length is to be validated for:

- **ORIGINAL_MIN** - Minimum bit length of 512 bits
- **PKE_DSV_MIN** - Minimum bit length of 128 bits
- **PKD_MIN** - Minimum bit length of 128 bits for external keys; otherwise, 512 bits is minimum

**Output**

This function returns no output.

**Return Codes**

Common return codes generated by this routine are:

- **OK** The operation was successful.
- **ERROR** The input token is not an RSA key token.
- **RSA_KEY_INVALID** The input token is not an internal or external RSA key token.
- **RT_TKN_UNUSEABLE** The input token is not an RSA key token.
- **E_KEY_TKNVER** Incorrect version data in input token.
- **E_PKA_KEYINVALID** An error was found in the token.
VerifyKeyTokenConsistency - Verify Key Token Consistency
VerifyKeyTokenConsistency verifies that the length specified in the input matches the length of the RSA key token, and that the length contained in the token is consistent with the lengths of all of the parts of the token.

Function Prototype
long VerifyKeyTokenConsistency( RsaKeyTokenHeader *pToken, USHORT tokenLengthIn )

Input
On entry to this routine:

pToken is a pointer to the key token.

tokenLengthIn is the length of the token specified by pToken.

Output
On successful exit from this routine:

VerifyKeyTokenConsistency returns OK if the key token was consistent, and FALSE otherwise.

Return Codes
This function has no return codes.
verify_dSig - Verify RSA Key Token Signature

verify_dSig receives an RSA key token in operational form, a buffer of data (with the length of the data), a
digital signature and the length of the digital signature (in bytes), as well as the format of the digital
signature. The data is hashed with SHA-1 and formatted according to the Type variable before being
compared with the encrypted signature. The return code indicates whether the signature was verified.

Function Prototype

long verify_dSig ( RsaKeyTokenHeader *pTokenIn,
                 UCHAR *pDataIn,
                 long DataLength,
                 UCHAR *pDigitalSignature,
                 USHORT SignatureLength,
                 UCHAR Type)

Input

On entry to this routine:

pTokenIn is a pointer to the operational key token.

pDataIn is a pointer to the data which is to be hashed and compared with the encrypted signature.

DataLength is the length of the data to be signed, in bytes.

pDigitalSignatureOut is a pointer to a buffer which contains the signature to be verified.

SignatureLength is the length of the buffer pSignatureOut, in bytes.

Type is one of the following:

• M_ISO9796 if the data is to be formatted according to the ISO-9796 standard before signing.
• M_PKCS1 if the data is to be formatted as specified in the RSA Data Security, Inc., Public Key Cryptog-
  raphy Standards #1 block type 00 before signing.
• M_PKCS11 if the data is to be formatted as specified in the RSA Data Security, Inc., Public Key Cryptog-
  raphy Standards #1 block type 01 before signing.
• M_ZEROPAD if the Data is to be placed in the low-order bits of a bit-string of the same length as the mod-
  ulus with all other bit-positions set to zero before signing.

Output

This function returns no output.

Return Codes

Common return Codes generated by this routine are:

OK The operation was successful.

DSIG_NOT_VERIFIED The digital signature was not verified.

E_SIZE The provided buffer was not large enough to contain the signature.

PKABadAddr The key token is not valid.
Chapter 10. 

CCA SRDI Manager Functions

This chapter describes the CCA SRDI Manager, which manages the storage and retrieval of persistent data in the coprocessor.

Note: All functions within this chapter are available only on the coprocessor.

Header Files for SRDI Manager Functions

When using these functions, your program must include the following header files.

```c
#include "cmncrypt.h" /* Cryptographic definitions */
#include "cam_xtrn.h" /* SRDI manager definitions */
```

Overview

The security relevant data item (SRDI)\(^1\) Manager is the interface through which CCA-related functions access security related data. Only the SRDI Manager interacts with the Flash EPROM on which the SRDI data is stored. The CCA verbs and any other CCA code read and write SRDI information through the SRDI Manager interface. In turn, the SRDI Manager accesses the physical SRDI storage through the Linux JFFS2 Filesystem Manager, which controls the flash EPROM memory. This relationship is shown in Figure 10-1 on page 10-2.

---

\(^1\) SRDIs are the sensitive data elements owned by the cryptographic application, and requiring protection. Examples include cryptographic keys and access control profiles.
Encapsulation of the SRDI physical storage mechanism makes it possible to change that mechanism without any effect on the CCA application code.

Each SRDI is identified by a name, much like a file name. The SRDI name is an eight character ASCII string, with no null terminator. SRDIs created by a UDX must have names which begin with the letters “UDX”, followed by a 4-digit number, for example “UDX0001”. The number must be unique and must not be “0000”. Names that are less than eight characters should be left-justified, and padded on the right with ASCII spaces.
CCA SRDI Manager Operation

The Linux JFFS2 Filesystem Manager stores SRDI data in the flash EPROM memory, encrypted with a secret key.

Flash EPROM

The flash memory is very large, but very slow to write. In addition, it has a limited lifetime in terms of write cycles; after 100,000 writes to any single memory cell, that cell may fail.

The flash memory can only be written in segments of 64K bytes. Thus, when any SRDI is written to flash, the Manager will usually have to rewrite a large amount of data that is not associated with that particular SRDI, but happens to lie in the same 64K byte page. This means that the 100,000 cycle lifetime may be reached much more quickly than expected, if a calculation is made based only on the number of times a specific SRDI is stored.

These characteristics make flash the appropriate location for SRDI data that is large, and infrequently changed. Examples include access control profiles, and stable cryptographic keys.

BBRAM

BBRAM is small, but fast, and it has no limitations on the number of times it can be written. This makes it the appropriate choice for storage of data that is small, and frequently updated. Examples include session keys, sequence counters, and state information. The BBRAM medium is mounted as a filesystem at /bbram. In order to use the BBRAM memory, you must open, write, and read files in this directory. You will also be responsible for controlling concurrent access to your data.

CCA applications do not have direct access to the SRDI information stored in Flash EPROM. When an SRDI is opened, the SRDI Manager creates a cleartext copy in RAM, in the CCA application address space. The caller receives a pointer to this location in RAM, and uses that space for all read and write references to the SRDI.

Only one working copy of an SRDI exists in RAM at any time, regardless of how many different callers open that same SRDI. The SRDI Manager maintains an open count for each open SRDI, indicating how many callers are using it. This count is initialized to one when the first caller opens the SRDI, and incremented for each additional open request on the same SRDI. When a caller closes the SRDI, the count is decremented. If the count reaches zero, indicating that no callers are using the SRDI, the working copy is deleted from memory.

When the caller asks to store the SRDI data, the SRDI Manager copies it to the persistent memory. Since there is only one physical working copy of the data at any one time, each caller’s changes are made to the same SRDI data area, and all are saved when any of the callers requests that the SRDI be stored.

---

2. Persistent memories are those that preserve their contents even when power is turned off. In the coprocessor, the flash EPROM and the BBRAM are persistent. The main system RAM used for executing programs and their data is not persistent.
An Example: Opening an SRDI

Figures 10-2, 10-3, and 10-4 describe the steps when an SRDI is opened. The following text explains the sequence of events, using reference numbers that match those on the figures.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>A CCA command processor sends a request to the CCA SRDI Manager, asking for access to an SRDI named <em>UDX0025</em>, which resides in flash EPROM. At this time, SRDI <em>UDX0025</em> is not open. No copy of the SRDI data exists in the CCA application RAM address space.</td>
</tr>
<tr>
<td>2.</td>
<td>The CCA Manager uses fstat() to determine the length of SRDI <em>UDX0025</em>. It needs to know the length, so it can allocate the required buffer in RAM.</td>
</tr>
<tr>
<td>3.</td>
<td>The CCA SRDI Manager allocates a buffer to hold <em>UDX0025</em>. This buffer is in RAM addressable by the CCA application.</td>
</tr>
</tbody>
</table>

Figure 10-2. Master SRDI Read Illustration, Part 1
Step | Description
--- | ---
4. | The CCA SRDI Manager sends a request to the Linux Filesystem Manager, asking it to read *UDX0025* into the buffer allocated in step 3.
5. | The SRDI is read from flash EPROM, decrypted, and deposited in the specified buffer.

![Diagram](image)

Figure 10-3. Master SRDI Read Illustration, Part 2

Step | Description
--- | ---
6. | The CCA SRDI Manager returns the buffer address to the CCA command processor. The command processor then uses the RAM copy of the SRDI whenever it needs to read or alter *UDX0025*.

![Diagram](image)

Figure 10-4. Master SRDI Read Illustration, Part 3 Controlling Concurrent Access to an SRDI
Since the CCA application is multi-threaded, different callers may access an SRDI at the same time. If one caller is altering data in the SRDI while a different caller is either reading or writing that same data, corruption results.

Serialization semaphores are used to prevent this from occurring. Each time the SRDI Manager retrieves an SRDI from flash EPROM or BBRAM, it allocates a semaphore for that SRDI. The SVid which identifies this semaphore is first locked, and then passed back to the caller whenever an SRDI is opened. For UDXes, the semaphore identifier is the numeric value of the 4 digits following “UDX” in the SRDI name added to X’21240000’. For example, the semaphore for “UDX0002” would be X’55440002’.

Every SRDI user in the CCA application is required to gain ownership of the semaphore before either reading or writing to the SRDI. This guarantees that no other caller is simultaneously accessing that same SRDI. As soon as the SRDI is no longer needed, the semaphore is released so that others can use the SRDI.

The semaphore is controlled by use of the Linux system calls sem_wait and sem_post, or by the UDX functions LIBR_RequestSemaphore and LIBR_ReleaseSemaphore. The CCA application should never, under any circumstances, destroy the semaphore; this is done by the SRDI Manager when the last user closes the SRDI.

**Summary of Functions**

These functions are used by the CCA command processor to read and write SRDI data.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>close_cca_srdi</td>
<td>Closes the open copy of an SRDI.</td>
</tr>
<tr>
<td>create_cca_srdi</td>
<td>Creates an SRDI.</td>
</tr>
<tr>
<td>delete_cca_srdi</td>
<td>Deletes an SRDI from memory.</td>
</tr>
<tr>
<td>get_cca_srdi_length</td>
<td>Obtains the length of an SRDI, in bytes.</td>
</tr>
<tr>
<td>open_cca_srdi</td>
<td>Opens and gains access to an SRDI.</td>
</tr>
<tr>
<td>resize_cca_srdi</td>
<td>Increases or decreases the length of an SRDI, in bytes.</td>
</tr>
<tr>
<td>save_cca_srdi</td>
<td>Stores SRDI data.</td>
</tr>
</tbody>
</table>
close_cca_srdi - Close CCA SRDI

close_cca_srdi deactivates the open copy of the SRDI, which is managed by the SRDI Manager. If no other applications are using the SRDI, the RAM which held the working copy of the SRDI is released.

Notes:

• If the working copy of the SRDI has been changed, the application must issue the save_cca_srdi() function in order to have the SRDI saved. SRDI data is not automatically saved when the SRDI is closed.
• The close_cca_srdi function does not release the semaphore which was locked by the open_cca_srdi function. This must be done by the user, prior to closing the SRDI.

Function Prototype

long close_cca_srdi( char *srdi_name )

Input

On entry to this routine

srdi_name is a pointer to an eight character ASCII string containing the name of the desired SRDI. This string does not contain a null terminator.

Output

This function returns no output. On successful exit from this routine:

close_cca_srdi deactivates the open copy of the SRDI.

Return Codes

Common return codes generated by this routine are:

srdi_NO_ERROR The operation was successful.
srdi_GENERAL_ERROR Cannot access the SRDI Manager, the operation cannot be completed.
srdi_NOT_OPEN The SRDI item is not in the open state.
**create_cca_srdi - Create CCA SRDI**

create_cca_srdi creates an SRDI in flash EPROM using the specified name.

**Note:** This function currently only works in EPROM. It does not work in BBRAM.

**Function Prototype**

```c
long create_cca_srdi( char *srdi_name, ULONG srdi_options, char *srdi_addr, ULONG srdi_length )
```

**Input**

On entry to this routine:

- `srdi_name` is a pointer to an eight character ASCII string containing the name of the desired SRDI. This string does not contain a null terminator, and should be padded on the right with blanks. It should begin with “UDX”, followed by 4 decimal digits.

- `srdi_options` holds bit-significant options which are passed on to the Linux Filesystem Manager. This parameter should be set to binary zeroes.

- `srdi_addr` is a pointer to the address of the SRDI data. This data is written to the newly created SRDI.

- `srdi_length` is the length of the SRDI data, in bytes.

**Output**

This function returns no output. On successful exit from this routine:

- `create_cca_srdi` creates an SRDI in flash EPROM using the specified name.

Data is only encrypted when stored in the flash EPROM; it is never encrypted in BBRAM. The BBRAM contents are destroyed on intrusion, so there is no need to protect the data there by way of encryption.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPD_NONE</td>
<td>X'00'</td>
<td>Store in flash, unencrypted.</td>
</tr>
</tbody>
</table>

**Return Codes**

Common return codes generated by this routine are:

- `srdi_NO_ERROR` The operation was successful.
- `srdi_GENERAL_ERROR` Cannot access the SRDI Manager, the operation cannot be completed.
- `srdi_EXISTS` The SRDI item already exists.
create4update_cca_srdi - Create CCA SRDI for Update Only

create4update_cca_srdi creates an SRDI in flash EPROM. The SRDI may not be resized after creation.

Note: This function was intended for updating BBRAM, but currently only writes to flash EPROM. Future releases will affect the BBRAM on the device.

Function Prototype

```c
long create4update_cca_srdi( char *pSrdiName,
                   char *pSrdiAddr,
                   ULONG SrdiLength )
```

Input

On entry to this routine:

- `pSrdiName` is a pointer to an eight character ASCII string containing the name of the desired SRDI. This string does not contain a null terminator, and should be padded on the right with blanks.

- `pSrdiAddr` is a pointer to the data which should be written to the SRDI.

- `SrdiLength` contains the length of the data pointed to by `pSrdiAddr`, and the permanent length of this SRDI.

Output

This function returns no output. On successful exit from this routine, the data has been stored in BBRAM under the requested name.

Return Codes

Common return codes generated by this routine are:

- `srdi_NO_ERROR` The operation was successful.
- `srdi_EXISTS` The SRDI item already exists.
- `srdi_GENERAL_ERROR` The SRDI Manager was unable to create the SRDI.
**delete_cca_srdi - Delete CCA SRDI**
delete_cca_srdi deletes an SRDI from the persistent memory area where it is stored. This is equivalent to erasing a file from a hard disk.

**Function Prototype**

```c
long delete_cca_srdi( char *srdi_name )
```

**Input**

On entry to this routine:

- `srdi_name` is a pointer to an eight character ASCII string containing the name of the desired SRDI. This string does not contain a null terminator, and should be padded on the right with blanks.

**Output**

This function returns no output. On successful exit from this routine:

- `delete_cca_srdi` deletes an SRDI from the persistent memory area where it is stored.

**Return Codes**

Common return codes generated by this routine are:

- `srdi_NO_ERROR` The operation was successful.
- `srdi_GENERAL_ERROR` Cannot access the SRDI Manager, the operation cannot be completed.
- `srdi_NOT_FOUND` SRDI item does not exist.
- `srdi_OPEN` The SRDI item is not in the closed state.

**Note:** An SRDI cannot be deleted if it is in the "open" state, since another application may be using it.
**get_cca_srdi_length - Get CCA SRDI Length**

get_cca_srdi_length obtains the length of the specified SRDI, in bytes.

**Function Prototype**

```c
long get_cca_srdi_length( char *srdi_name,
                          ULONG *srdi_length )
```

**Input**

On entry to this routine:

- `srdi_name` is a pointer to an eight character ASCII string containing the name of the desired SRDI. This string does not contain a null terminator.

- `srdi_length` is a pointer to the ULONG variable in which the length will be stored.

**Output**

On successful exit from this routine:

- `srdi_length` contains the length of the SRDI data, in bytes.

**Return Codes**

Common return codes generated by this routine are:

- **srdi_NO_ERROR** The operation was successful.
- **srdi_GENERAL_ERROR** Cannot access the SRDI Manager, the operation cannot be completed.
- **srdi_READ_ERROR** Unable to read the SRDI item from flash.
open_cca_srdi - Open CCA SRDI

open_cca_srdi opens an SRDI, gaining access to its contents. The function returns the address and length of the SRDI data, where the address points to a cleartext working copy of the actual SRDI, which is stored in flash EPROM.

If multiple callers open the same SRDI, they all have access to the same shared copy in RAM. Any modifications to the SRDI are visible immediately to all functions that open that SRDI.

In addition to the SRDI address and length, the function returns a semaphore ID for the selected SRDI, which has been locked by the open_cca_srdi function. The SRDI Manager names the semaphore according to the 4 digits which follow “UDX” in the name of the SRDI item; therefore, these digits should be unique to a given SRDI. This semaphore is used to gain exclusive access to the SRDI, to prevent errors when one thread is writing data, while another is simultaneously either reading or writing that same data. See "Master SRDI Read Illustration, Part 3 Controlling Concurrent Access to an SRDI” on page 10-5 for further details.

Function Prototype

```c
long open_cca_srdi( char     *srdi_name,
                    char **srdi_addr,
                    ULONG *srdi_length,
                    ULONG *semSvid )
```

Input

On entry to this routine:

- `srdi_name` is a pointer to an eight character ASCII string containing the name of the desired SRDI. This string does not contain a null terminator. The name should consist of the ASCII string “UDX” concatenated with a string of 4 decimal digits. The final character may be any character which is valid in a file name, or a space.

Output

On successful exit from this routine:

- `srdi_addr` is a pointer to a pointer variable, in which the SRDI Manager returns the address of the SRDI. This is an address in RAM, where the SRDI Manager places a copy of the SRDI data.

- `srdi_length` is a pointer to a location where the SRDI Manager stores the length of the SRDI data, in bytes.

- `semSvid` is the SVid for the semaphore assigned to the specified SRDI.

Return Codes

Common return codes generated by this routine are:

- `srdi_NO_ERROR` The operation was successful.
- `srdi_NOT_FOUND` The SRDI item could not be found.
- `srdi_READ_ERROR` Unable to read the SRDI item from flash.
- `srdi_ALLOC_ERROR` Unable to allocate memory for the SRDI item.
- `srdi_GENERAL_ERROR` Could not access the SRDI Manager, the operation was not completed.
**resize_cca_srdi - Resize CCA SRDI**

`resize_cca_srdi` increases or decreases the length of the specified CCA SRDI, in bytes.

An SRDI can only be resized if you are the only requestor who has it open. If the SRDI is opened by more than one user concurrently, it cannot be resized; the address of the RAM copy changes when it is resized, and there is no way to notify other callers of this.

**Function Prototype**

```c
long resize_cca_srdi( char   *srdi_name,
        ULONG    srdi_length,
        char **new_srdi_addr )
```

**Input**

On entry to this routine:

- `srdi_name` is a pointer to an eight character ASCII string containing the name of the desired SRDI. This string does not contain a null terminator.

- `srdi_length` is the new length for the SRDI, in bytes.

**Output**

On successful exit from this routine:

- `new_srdi_addr` is a pointer to a location where the function returns the address of the resized SRDI. After resizing, the SRDI buffer is relocated from its previous address.

**Return Codes**

Common return codes generated by this routine are:

- **srdi_NO_ERROR** The operation was successful.
- **srdi_NOT_FOUND** The SRDI item could not be found.
- **srdi_READ_ERROR** Unable to read the SRDI item from BBRAM or flash.
- **srdi_ALLOC_ERROR** Unable to allocate memory for the SRDI item.
- **srdi_GENERAL_ERROR** Could not access the SRDI Manager, the operation was not completed.
save_cca_srdi - Save CCA SRDI

save_cca_srdi stores the SRDI data on a persistent storage medium (flash EPROM).

This function ensures that no thread is updating the SRDI while it is being stored by gaining exclusive access rights using the SRDI semaphore. See "Master SRDI Read Illustration, Part 3 Controlling Concurrent Access to an SRDI" on page 10-5 for details on this semaphore.

Function Prototype

long save_cca_srdi( char *srdi_name )

Input

On entry to this routine:

srdi_name is a pointer to an eight character ASCII string containing the name of the desired SRDI. This string does not contain a null terminator.

Note: No two SRDIs can have the same name, even if one resides in flash EPROM and the other resides in BBRAM. The CCA SRDI Manager enforces this restriction.

Output

This function returns no output. On successful exit from this routine:

save_cca_srdi stores the SRDI data on a persistent storage medium.

Return Codes

Common return codes generated by this routine are:

srdi_NO_ERROR The operation was successful.

srdi_NOT_OPEN The SRDI item is not in the open state.

srdi_WRITE_ERROR Unable to write to flash or BBRAM.

srdi_GENERAL_ERROR Could not access the SRDI Manager, the operation was not completed.
**update_cca_srdi - Update an SRDI Item**

update_cca_srdi updates an SRDI with provided data.

**Function Prototype**

```c
long update_cca_srdi( char *pSrdiName, char *pDatabuffer, unsigned long Datalength, unsigned long Dataoffset )
```

**Input**

On entry to this routine:

- `pSrdiName` is a pointer to an eight character string containing the name of the SRDI to be changed. This string does not contain a null terminator and should be padded on the right with blanks. This SRDI must have been opened using the `create4update_cca_srdi` call.

- `pDatabuffer` is a pointer to a buffer containing the data with which to update the SRDI.

- `Datalength` contains the length of the data to be changed, in bytes.

- `Dataoffset` contains the offset of the first byte of data to change from `pDatabuffer`.

**Output**

This function returns no output. On successful exit from this routine, the SRDI item has been changed.

**Return Codes**

Common return codes generated by this routine are:

- `srdi_NO_ERROR` The operation was successful.
- `srdi_WRITE_ERROR` The operation could not be completed.
Example Code
The following C-language code shows a general structure for the way a CCA application would open, use, and close an SRDI.

```c
#define MY_SRDI_NAME "UDX0001 " /* Name of SRDI we’re using */

USHORT srdi_rc; /* SRDI fcn. return code */
char * my_srdi_addr; /* Pointer to clear SRDI data in RAM */
ULONG my_srdi_length; /* Length of SRDI data, in bytes */
ULONG semaphore_id; /* SVid of SRDI access semaphore */

void srdi_stuff ( void )
{
    /* Open the SRDI */
    srdi_rc = open_cca_srdi ( MY_SRDI_NAME, & my_srdi_addr, & my_srdi_length,
                              & semaphore_id);
    if ( srdi_NO_ERROR == srdi_rc) /* If no errors opening SRDI . . . */
    {
        /* do other stuff as needed. . . */
        /* This is where the code will read and/or write
           to the SRDI data, in the area pointed
           to by my_srdi_addr. */

        memcpy (my_srdi_addr, newdata, my_srdi_length);
        srdi_rc = save_cca_srdi (MY_SRDI_NAME);
        /* Release semaphore, allowing others to access the SRDI */
        LIBR_ReleaseSemaphore ( semaphore_id );

        if ( srdi_rc != srdi_NO_ERROR)
        {
            /* do stuff relating to write error for SRDI */
        }
        /* do other stuff as needed. . . */
        /* suppose we later want to read what was in the srdi */
        if ( OK == LIBR_RequestSemaphore ( semaphore_id ) )
        {
            if ( memcmp (calculated_data, my_srdi_address, my_srdi_length) )
                /* do whatever 'no change' means */
            else
                /* do whatever 'changed' means */
                LIBR_ReleaseSemaphore ( semaphore_id );
        }
        /* handle semaphore request error */
    }
    /* Close the SRDI */
    srdi_rc = close_cca_srdi ( MY_SRDI_NAME );
    if ( srdi_NO_ERROR != srdi_rc)
    {
        /* handle SRDI close error. . . */
    }
}
```

Chapter 11.  Access Control Manager Functions

The functions described in this chapter enable a UDX to initialize or update access control tables, log users on or off of the coprocessor, or check a user’s authority to perform a specified function.

Note: All functions within this chapter are available only on a coprocessor with the workstation load.

Note: Access control on a z/OS coprocessor is handled through the TKE and OA Manager; therefore, none of these functions are available on a coprocessor installed in a z/OS system.

Header Files for Access Control Manager Functions
When using these functions, your program must include the following header files.

```
#include "camxtrn.h" /* CCA managers */
#include "camacm.h"  /* Access Control Manager */
#include "camacm_p.h" /* Access Control Manager */
```

Summary of Functions
Access Control Manager includes the following functions.

- `ac_check_authorization`: Check whether a user is authorized to execute a specified function.
- `ac_chg_prof_auth_data`: Change the authentication data for a user profile.
- `ac_chg_prof_exp_date`: Change the expiration date of a user profile.
- `ac_del_profile`: Delete a user profile.
- `ac_del_role`: Delete a role definition.
- `ac_get_list_sizes`: Find out how many roles and how many profiles exist on the coprocessor.
- `ac_get_profile`: Read the contents of a specified user profile.
- `ac_get_role`: Read the contents of a specified role.
- `ac_init`: Initialize the access control system, including setup of role and profile tables.
- `ac_list_profiles`: Get a list of all the user IDs for which there are profiles.
- `ac_list_roles`: Get a list of all the role IDs.
- `ac_load_profiles`: Load one or more user profiles.
- `ac_load_roles`: Load one or more role definitions.
ac_lu_add_user  Add a user to the table of logged-on users, and generate a session key for that user.
ac_lu_drop_user  Drop a user from the table of logged-on users.
ac_lu_get_ks  Get a copy of a specified user’s session key.
ac_lu_get_num_users  Find out how many users are currently logged on.
ac_lu_get_role  Get the role ID for a specified logged-on user.
ac_lu_ks_dec  Decrypt data with a specified user’s session key.
ac_lu_ks_enc  Encrypt data with a specified user’s session key.
ac_lu_ks_hmacgen  Computes a HMAC on a message using the session key.
ac_lu_ks_hmacver  Verifies an HMAC using the session key.
ac_lu_ks_macgen  Compute a MAC (message authentication code) using a specified user’s session key.
ac_lu_ks_macver  Verify a MAC (message authentication code) using a specified user’s session key.
ac_lu_list_users  Get a list of all the logged-on users.
ac_lu_query_user  Check whether a specified user is currently logged on (authenticated).
ac_query_profile  Verify that a specified profile exists on the coprocessor, and return its length.
ac_query_role  Verify that a specified role exists on the coprocessor.
ac_reinit  Reinitialize the access control system, to the default state.
ac_reset_logon_fail_cnt  Reset the logon failure count in a user profile.
acm_xlt_errorcode  Translates an ACM error code to an SAPI error code.

SRDI Files
The Access Control Manager uses two SRDI files, stored in flash EPROM.

- Profiles - holds all the user profiles that are enrolled on the coprocessor.
- Roles - holds all the roles that have been defined on the coprocessor.

Each of these SRDIs has a similar format. The file begins with a 4-byte header, in which the first two bytes contain an integer specifying how many items are in the file, and the second two bytes are reserved and set to X'0000'. The number of items reflects either the number of profiles or the number of roles, depending on the SRDI in question.

The profiles or roles follow the header. The first one is concatenated to the end of the header, and each successive role or profile is concatenated to its predecessor. Neither roles nor profiles are ordered in any meaningful way.

For quicker access, the Access Control Manager builds indexes in RAM for the roles and the profiles. For each role or profile in the SRDIs, the respective index table holds the name of the role or profile and its offset from the start of the SRDI.
Data Structures
This section contains definitions for common data structures used in the Access Control Manager.

Generic Data Types
The following types are used in various other definitions.

typedef unsigned char two_byte[2]; // "Big-endian" 2-byte integer

It is assumed that all of the "standard" types are available, such as UCHAR, UINT, USHORT, and boolean.

Profile Structures
The Profile ID is an eight character unterminated string, as follows.

typedef UCHAR profile_id_t[8]; // Profile ID

The activation and expiration dates use the following format.

typedef struct {
two_byte year;
UCHAR month;
UCHAR day;
} prof_date_t;

The profile consists of a fixed structure, with the authentication data concatenated to the end. The fixed portion is defined with the structure below.

typedef struct {
two_byte profile_ver; // Profile structure version
two_byte profile_lth; // Length of entire profile
char comment[2]; // Descriptive comment
two_byte cksum; // Checksum for integrity
UCHAR failure_cnt; // Logon failure count
UCHAR reserved_1; // Reserved field
profile_id_t user_id; // User ID
role_id_t role_id; // Role ID
prof_date_t act_date; // Activation date
prof_date_t exp_date; // Expiration date
// The variable-length authentication data appears here
} user_profile_t;

Role Structures
The Role ID is an eight character unterminated string, as follows.

typedef UCHAR role_id_t[8]; // Role ID

The role definition consists of a fixed structure, with the list of permitted access control points appended at the end. The fixed part of the structure is defined by the following type declaration.

typedef struct {
two_byte role_ver; // Role structure version
two_byte role_lth; // Length of role structure
char comment[20]; // Descriptive comment
two_byte cksum; // Checksum for integrity
two_byte reserved_1; // Reserved field
role_id_t role_id; // Role ID
} role_t;
Access Control Manager Functions

User Information Structures

The Access Control Manager holds the following data for each user logged on to the coprocessor.

- The profile ID (user ID)
- The role ID for that user
- The session key $K_S$ for the user

The structures defining this data are as follows.

```c
typedef UCHAR des_key[8];  // DES key
typedef user_session_key_t des_key[3];  // Triple-length DES session key

typedef struct {
    profile_id_t userid;  // User ID
    role_id_t role_id;  // User’s role
    dl_des_key ks;  // User’s session key
} user_info_t;

#define INITIAL_USER_TBL_SIZE 20  // Initial number of spaces in table
```

// There will be a static pointer to an array of user_info_t elements. If the
// array fills, a new and larger array will be allocated and assigned to the
// pointer, and the contents of the old array copied over.
**ac_check_authorization - Check Authorization to Execute Function**

ac_check_authorization determines if a user is permitted to execute a specified function, based on the user's role, access control state information, and other access control parameters.

**Function Prototype**

```c
long ac_check_authorization( role_id_t role,
                            USHORT requested_fcn,
                            boolean *granted);
```

**Input**

- `role_id_t` is a pointer to the role SRDI.
- `role` is the role under which the user is operating.
- `requested_fcn` is an index into the Access Control Points table, indicating which function the user wants to execute.
- `granted` is a pointer to a variable that will receive the response, indicating whether the user is allowed to execute the desired function.

**Output**

On successful exit from this routine:

- `ac_check_authorization` returns a boolean value of TRUE indicating the user is allowed to use the specified function in the CCA application, and FALSE if not.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR** The operation was successful.
- **acm_ROLE_NOT_FOUND** The specified role was not found.
**ac_chg_prof_auth_data - Change Profile Authentication Data**

`ac_chg_prof_auth_data` replaces the authentication data in a user profile, for a specific authentication mechanism. The old data is replaced with the new data, but only if the Replacable flag is set in the attributes of the old authentication data.

Alternatively, if no data exists in the profile for the specified mechanism, the data is appended to the authentication data already present in that profile.

**Function Prototype**

```c
long ac_chg_prof_auth_data( profile_id_t profile_id,
                        void *auth_data);
```

**Input**

- `profile_id` is the 8-byte, non-NULL terminated ID of the profile whose authentication data is to be changed.
- `auth_data` is a pointer to a buffer containing the new authentication data, in the following format:
  - a variable of type `profs_authen_mech_t` which contains:
    - `numbytes` - the number of bytes of data in the buffer.
    - `mech_Id` - an identifier which describes the authentication mechanism - for passphrase authentication, the mechanism ID is X'01'
    - `mech_strength` - an integer which defines the strength of this authentication method. X'00' is reserved for users who have not been authenticated.
    - `exp_date` - a `prof_date_t` variable containing the expiration date for this authentication data. The format is:
      - year is a 2-byte integer in big-endian order
      - month is a one-byte integer (1-12)
      - date is a one-byte integer (1-31)
    - `mech_attr` - a bit-flag to represent any attributes needed to describe the operation and use of this authentication mechanism. The only currently defined value is RENEWABLE, in the most significant bit. The presence of this flag indicates that an authentication method may be updated by the user (for example, a passphrase change).
  - concatenated to the `profs_authen_mech_t` variable, a variable length field containing the authentication data for this user and method. For passphrases, this field contains the 20-byte SHA-1 hash of the user's passphrase. This hash is computed in the host.

**Output**

This function returns no output. On successful exit from this routine:

- The profile specified in `profile_id` has been updated in one of the following two ways:
  1. If the profile previously had an authentication with the same mechanism ID, its authentication method has been changed.
  2. If the profile previously had NO authentication with an identical mechanism ID, the authentication data has been ADDED to the profile.

The PROFILES SRDI has been saved to flash EPROM.
Return Codes
Common return codes generated by this routine are:

- **acm_NO_ERROR**: The operation was successful.
- **acm_INVALID_AUTHENT_DATA**: The authentication data was invalid.
- **acmPROFILE_NOT_FOUND**: The specified profile was not in the profile list.
- **acm_NON_REPLACABLE_DATA**: The REPLACABLE flag was not set in the user’s profile.
- **acm_MEM_ALLOC_ERROR**: There was insufficient memory to store the new data.
### ac_chg_prof_exp_date - Change Profile Expiration Date

ac_chg_prof_exp_date changes the expiration date in a specified user profile. This function is used to re-enable a user whose profile has expired, or to extend the lifetime of a profile that is about to expire.

**Function Prototype**

```c
long ac_chg_prof_exp_date( profile_id_t profile_id,
                          prof_exp_date_t exp_date);
```

**Input**

- `profile_id` is the ID of the profile whose expiration date should be changed.
- `exp_date` is the new expiration date, in the following format:
  - `year` is a 2-byte integer in big-endian order
  - `month` is a one-byte number representing the month (1-12)
  - `date` is a one-byte number representing the date (1-31)

**Output**

This function returns no output. On successful exit from this routine:

> The expiration date of `profile_id` has been changed to `exp_date`, and the PROFILES SRDI has been saved to flash EPROM.

**Return Codes**

Common return codes generated by this routine are:

- `acm_NO_ERROR` The operation was successful.
- `acmPROFILE_NOT_FOUND` The specified profile was not in the profile list.
**ac_del_profile - Delete User Profile**

ac_del_profile deletes a user profile from the coprocessor. The specified profile is deleted from the table of enrolled profiles. In addition, if the specified user is logged on, they are logged off.

**Function Prototype**

```c
long ac_del_profile( profile_id_t profile_id);
```

**Input**

- `profile_id` is the 8-byte, non-NULL terminated ID of the profile to be deleted.

**Output**

This function returns no output. On successful exit from this routine:

The user specified by `profile_id` has been logged off the system (if needed) and the profile has been removed from the profiles list. In addition, the PROFILES SRDI has been saved to flash memory.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR** The operation was successful.
- **acm_PROFILE_NOT_FOUND** The specified profile was not in the profile list.
ac_del_role - Delete Role
ac_del_role deletes a specified role definition from the coprocessor. In addition, if any logged on users have their authority defined by this role, those users are logged off.

If the request is to delete the DEFAULT role, the role is not actually deleted; the default role must always be present on the coprocessor. Instead, it is reset to the default values for that role.

Function Prototype
long ac_del_role( role_id_t role_id);

Input
role_id is the 8-byte, non-NULL terminated ID of the role to be deleted.

Output
This function returns no output. On successful exit from this routine:

The specified role has been deleted, unless the role specified was DEFAULT. If the specified role was DEFAULT, the Default role has been reset to its original contents. The list of roles has been saved to flash memory.

Return Codes
Common return codes generated by this routine are:

acm_NO_ERROR The operation was successful.
acm_ROLE_NOT_FOUND The specified role was not found.
**ac_get_list_sizes - Get Sizes of Role and Profile Lists**
ac_get_list_sizes is used to find out how much data will be returned by the `ac_list_profiles` and `ac_list_roles` functions. By calling this function first, the application can ensure it has a large enough buffer for the data it will receive.

**Function Prototype**
```c
long ac_get_list_sizes( ULONG *num_profiles, ULONG *num_roles );
```

**Input**
- `num_profiles` is a pointer to a variable that will receive the number of profiles on the coprocessor.
- `num_roles` is a pointer to a variable that will receive the number of roles on the coprocessor.

**Output**
On successful exit from this routine:
- `num_profiles` contains the number of profiles returned.
- `num_roles` contains the number of roles returned.

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

- **acm_NO_ERROR** The operation was successful.
**ac_get_profile - Get Profile**

`ac_get_profile` returns the information from a specified user profile. The data begins with the fixed profile contents, defined by data type `user_profile_t`. The user’s authentication data is concatenated to the end of this structure.

**Function Prototype**

```c
long ac_get_profile(    profile_id_t profile_id,
            ULONG *profilesize,
            void *profile_data) ;
```

**Input**

- `profile_id` is an 8-byte character string non-NULL terminated containing the name of the profile to be returned.
- `profile_size` is a pointer to a variable which contains the size of the `profile_data` buffer, in bytes.
- `profile_data` is a pointer to a buffer where the profile will be stored.

**Output**

On successful exit from this routine:

- `profile_size` contains the actual size of the returned profile data.
- `profile_data` contains the profile data.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR** The operation was successful.
- **acmPROFILE_NOT_FOUND** The specified profile was not in the profile list.
- **acm_BFR_TOO_SMALL** The buffer was not large enough to hold the profile data. The buffer size required is in the `profile_size` variable.
**ac_get_role - Get Role**

ac_get_role returns the information from a specified role.

**Function Prototype**

```c
long ac_get_role( role_id_t role_id,
                 ULONG *role_size,
                 void *role_data);
```

**Input**

*role_id* is an 8-byte character string non-NULL terminated containing the name of the role to be returned.

*role_size* is a pointer to a variable which contains the size of the *role_data* buffer, in bytes.

*role_data* is a pointer to a buffer where the role data will be stored.

**Output**

On successful exit from this routine:

*role_size* contains the actual size of the returned role data.

*role_data* contains the role data.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR** - The operation was successful.
- **acm_ROLE_NOT_FOUND** - The specified role was not found.
- **acm_BFR_TOO_SMALL** - The buffer was not large enough to hold the role data. The buffer size required is in the *role_size* variable.
ac_init - Initialize the Access Control Manager

ac_init initializes the Access Control Manager including all data areas and state information. In addition, creates and initializes the role and profile SRDIs if they do not already exist, including creation of a DEFAULT role.

Function Prototype
long ac_init(void);

Input
This function has no input.

Output
This function has no output. On successful exit from this routine:

ac_init creates and initializes the Access Control Manager.

Return Codes
Common return codes generated by this routine are:

acm_NO_ERROR The operation was successful.
**ac_list_profiles - List User Profiles**

ac_list_profiles returns a list of all the user IDs for the profiles enrolled on the coprocessor. This is a series of unordered 8-byte user IDs.

**Function Prototype**

```c
long ac_list_profiles( profile_id_t profile_list [],
                      ULONG *num_profiles);
```

**Input**

- `profile_list` is a pointer to an array of 8-byte elements to hold the user IDs.
- `num_profiles` is a pointer to the maximum number of 8-byte values which the array can hold.

**Output**

On successful exit from this routine:

- `profile_list` contains the list of IDs (non-NULL terminated 8-byte strings).
- `num_profiles` contains the number of profile IDs which were returned.

**Return Codes**

Common return codes generated by this routine are:

- `acm_NO_ERROR` The operation was successful.
- `acm_BFR_TOO_SMALL` The array is not large enough to hold all the profile IDs.
**ac_list_roles - List Roles**

ac_list_roles returns a list of the role IDs for each role defined on the coprocessor. This is a series of unordered 8-byte role IDs.

**Function Prototype**

```c
long ac_list_roles( role_id_t role_list[],
                    ULONG *num_roles);
```

**Input**

role_list is a pointer to an array to hold the list of role IDs.

num_roles is a pointer to a variable containing the maximum number of 8-byte IDs which the array can hold.

**Output**

On successful exit from this routine:

role_list contains the list of role IDs.

num_roles contains the number of role IDs returned.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR** The operation was successful.
- **acm_BFR_TOO_SMALL** The array was not large enough to hold all of the role IDs.
ac_load_profiles - Load User Profiles
ac_load_profiles loads one or more user profiles into the coprocessor. Existing profiles can be replaced, if new ones have the same name.

The user profiles are an aggregate structure, where any number of profiles can be concatenated into a single message.

Function Prototype
long ac_load_profiles( void *profile_list,
                        boolean replace_profiles);

Input
On entry to this routine:

profile_list is a pointer to a variable length buffer of the following form:

- a 4-byte variable containing the number of profiles in this list followed by
- a 4-byte variable containing X'0000' followed by
- the first profile followed by
- any profiles in between followed by
- the last profile

Each of the profiles in the profile list is a user_profile_t variable initialized to the following values:

- profile_vers - the profile structure version, currently X'01'.
- profile_lth - the length of this profile, in bytes.
- comment - a descriptive comment, of 19 bytes or less in length, NULL terminated.
- cksum - a checksum value to ensure the integrity of the data.
- failure_cnt - the number of logon failures, initialized to 0.
- user_id - the user ID associated with this profile. An 8- byte, non-NULL terminated string.
- role_id - an existing Role from the ROLES SRDI. An 8- byte, non-NULL terminated string.
- act_date - the activation date of this user, a prof_date_t variable with the following values:
  - year - a 2-byte variable with the year (4 digits) in big-endian format
  - month - a one-byte character representing the month (1-12)
  - day - a one-byte character representing the date (1-31)
- exp_date - the expiration date of this user, a prof_date_t variable with the values as the act_date structure
- authentication data is included at this point. The variable length nature of authentication data is the reason for the profile_lth field.

replace_profiles is a boolean variable with TRUE indicating that existing profiles should be changed, FALSE indicating that they should not.

Output
This function returns no output. On successful exit from this routine:

The PROFILES SRDI has been updated and saved to flash memory.
Return Codes
Common return codes generated by this routine are:

- acm_NO_ERROR: The operation was successful.
- acm_FLASH_SPACE_FULL: There is insufficient memory in the flash EPROM to store the new profile list.
- acm_MEMALLOC_ERROR: The function was unable to allocate sufficient memory to install the new profiles.
- acm_PROFILE_EXISTS: The user_id is already in the PROFILES SRDI, and replace_profiles was FALSE. No profiles have been added.
**ac_load_roles - Load Roles**

ac_load_roles loads one or more roles into the coprocessor. The roles are passed in an aggregate structure, which can contain any number of concatenated role definitions.

**Function Prototype**

```c
long ac_load_roles( void *role_list);
```

**Input**

On entry to this routine:

- `role_list` is a pointer to a variable length buffer of the following form:

  - a 4-byte variable containing the number of roles in this list followed by
  - a 4-byte variable containing 0 followed by
  - the first role followed by
  - any roles in between followed by
  - the last role

Each role is a variable length role initialized to the following:

- A `role_t` variable containing:
  - `role_vers` - the role structure version, X'01'
  - `role_lth` - the number of bytes in this role
  - `comment` - a 20-byte comment, NULL-terminated, describing this role
  - `cksum` - checksum, for determining role integrity
  - `role_id` - an 8-byte, non-NULL terminated string, the ID for this role.
  - `reqd_auth_str` - the required authentication strength for this role. Each method of authentication is assigned a strength value, with 0 being no authentication. A role may be restricted to users who have logged on with a more stringent method (that is, passphrase rather than PIN) if more than one method has been supplied.
  - `lower_time_limit`
  - `upper_time_limit`
  - `valid_dow`

- Followed by an access control point list for the role.

**Output**

This function has no output. On successful exit from this routine:

The new role is added to the existing set of roles. If the role exists, it is replaced with the new role.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR** - The operation was successful.
**ac_lu_add_user - Add a User to the List of Logged on Users**

ac_lu_add_user informs the Access Control Manager that a specified user has successfully authenticated, and should be added to the list of logged-on users.

The Access Control Manager adds a new entry to the table of logged on users, inserting the specified profile ID as the search key used to identify the entry. It finds the profile and looks up the role, and also puts that in the table entry. This saves the cost of finding the profile each time the user’s role is required for future access control operations. Finally, the Access Control Manager generates a session key for the user, and inserts that in the table with the ID and the role.

If the specified profile ID is already in the table of logged on users, the request is rejected.

**Function Prototype**

```c
long ac_lu_add_user( profile_id_t profile_id) ;
```

**Input**

`profile_id` is an 8-byte, non-NULL terminated string representing the user to be logged on.

**Output**

This function has no output. On successful exit from this routine:

The user has been added to the list of logged-on users.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR**  
  The operation was successful.

- **acm_ALREADY_LOGGED_ON**  
  The user was already logged on.

- **acm_MEM_ALLOC_ERROR**  
  The function was unable to allocate sufficient memory to add the role.

- **acm_ROLE_NOT_FOUND**  
  The specified role was not found.

- **acm_PROFILE_NOT_FOUND**  
  The specified profile was not in the profile list.
**ac_lu_drop_user - Remove a User from the Logon List**

ac_lu_drop user informs the Access Control Manager that a user should be logged off, dropping that user from the logged-on users table.

**Function Prototype**

```c
long ac_lu_drop_user( profile_id_t profile_id);
```

**Input**

profile_id is an 8-byte, non-NULL terminated string containing the name of the profile to be removed.

**Output**

This function returns no output. On successful exit from this routine:

ac_lu_drop_user removes the user from the logged-on users table.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR** The operation was successful.
- **acm_NOT_LOGGED_ON** The user was not logged on.
**ac_lu_get_ks - Get a Copy of a Session Key**

ac_lu_get_ks gets a copy of the session key for a specified logged-on user.

**Function Prototype**

```c
long ac_lu_get_ks( profile_id_t profile_id,
                   user_session_key_t *session_key) ;
```

**Input**

- `profile_id` is an 8-byte, non-NULL terminated string containing the name of the profile whose session key you want to retrieve.

- `session_key` is a pointer to the caller's variable which will receive the cleartext session key $K_S$ for the specified user.

**Output**

On successful exit from this routine:

- `session_key` contains the cleartext session key $K_S$ for the specified user.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR**
  The operation was successful.
**ac_lu_get_num_users - Get the Number of Logged On Users**

`ac_lu_get_num_users` returns the number of users who are currently logged on to the coprocessor. This can be used to determine how much data will be returned by the `ac_lu_list_users` function.

**Function Prototype**

```c
long ac_lu_get_num_users( ULONG *num_users);
```

**Input**

`num_users` is a pointer to a variable which receives the number of logged-on users.

**Output**

On successful exit from this routine:

`num_users` contains the number of users who are currently logged on to the coprocessor.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR** The operation was successful.
au_lu_get_role - Get Role from the Logon List
au_lu_get_role returns the role ID for a specified logged-on user.

Function Prototype
long ac_lu_get_role( profile_id_t profile_id,
                   role_id_t role) ;

Input
profile_id is an 8-byte, non-NULL terminated ID of the profile that you want to retrieve.

role is a variable which receives the role ID for the specified user.

Output
On successful exit from this routine:

role contains the role ID for the specified user.

Return Codes
Common return codes generated by this routine are:

acm_NO_ERROR The operation was successful.
**ac_lu_ks_dec - Decrypt Data with Session Key**

ac_lu_ks_dec decrypts a string of data with a specified user’s session key. The data length must be a multiple of eight bytes. Decryption is performed using triple-DES CBC mode, with an IV of X'0000000000000000'.

**Function Prototype**

```c
long ac_lu_ks_dec( profile_id_t profile_id,
                  UCHAR *ciphertext,
                  UCHAR *cleartext,
                  ULONG text_length);
```

**Input**

- `profile_id` is an 8-byte, non-NULL terminated string representing the name of the profile whose session key should be used to decrypt the data.

- `ciphertext` is a pointer to the buffer containing the data to be deciphered.

- `cleartext` is a pointer to a buffer where the deciphered data will be stored. This buffer must be at least as large as the `ciphertext` buffer.

- `text_length` is the length of the data to be deciphered, in bytes. This value must be a multiple of eight.

**Output**

On successful exit from this routine:

- `cleartext` contains the deciphered data.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR** The operation was successful.
**acLu_ks_enc - Encrypt Data with Session Key**

Encrypt a string of data with a specified user’s session key. The data length must be a multiple of eight bytes. Encryption is performed using triple-DES CBC mode, with an IV of X’0000000000000000’.

**Function Prototype**

```c
long acLu_ks_enc( profile_id_t profile_id,
                   UCHAR *cleartext,
                   UCHAR *ciphertext,
                   ULONG text_length);
```

**Input**

- **profile_id** is an 8-byte, non-NULL terminated string representing the name of the profile whose session key should be used to encrypt the data.

- **cleartext** is a pointer to the buffer containing the data to be enciphered.

- **ciphertext** is a pointer to a buffer where the enciphered data will be stored. This buffer must be at least as large as the cleartext buffer.

- **text_length** is the length of the data to be enciphered, in bytes. This value must be a multiple of eight.

**Output**

On successful exit from this routine:

- **ciphertext** contains the enciphered data.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR** The operation was successful.
**ac_lu_ks_hmacgen - Compute an HMAC Using Session Key**

Computes the HMAC on a message, using the supplied session key.

**Function Prototype**

```c
long ac_lu_ks_hmacgen( profile_id_t profile_id,
                        UCHAR *messagetext,
                        ULONG msg_length,
                        mac_t mac )
```

**Input**

On entry to this routine:

- `profile_id` must contain the ID of the profile of the logged-on user.
- `messagetext` is a pointer to a buffer containing the message.
- `msg_length` is the number of bytes of data in `messagetext`.
- `mac` is an 8-byte buffer.

**Output**

On successful exit from this routine:

- `mac` contains the 8-byte HMAC of the `messagetext`, computed with the session key associated with `profile_id`.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR** — The operation was successful.
**ac_lu_ks_hmacver - Verify an HMAC Using Session Key**

Verifies a supplied MAC on a supplied message, using the user’s session key.

**Function Prototype**

```c
long ac_lu_ks_hmacver( profile_id_t profile_id,
                       UCHAR *messagetext,
                       ULONG msg_length,
                       mac_t mac,
                       USHORT mac_length,
                       boolean *verified ) ;
```

**Input**

On entry to this routine:

- **profile_id** should contain the user’s profile ID.
- **messagetext** is a pointer to the message that is to be verified.
- **msg_length** is the number of bytes in the message.
- **mac** is the 8-byte HMAC which is to be verified.
- **verified** is a pointer to a boolean variable, in which the answer will be supplied.

**Output**

On successful exit from this routine:

- **verified** contains true if the MAC was verified, and false if it was not.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR** The operation was successful.
**ac_lu_ks_macgen - Compute a MAC using Session Key**

ac_lu_ks_macgen computes a triple-DES MAC (message authentication code) using the specified user’s session key. The full 8-byte MAC result is returned.

The data length must be a multiple of eight bytes. The MAC is computed using an IV of X’0000000000000000’.

**Function Prototype**

```c
long ac_lu_ks_macgen( profile_id_t  profile_id,
                      UCHAR *messagetext,
                      ULONG  msg_length,
                      mac_t   *mac) ;
```

**Input**

- **profile_id** is an 8-byte, non-NULL terminated string representing the name of the profile whose session key should be used to compute the MAC.

- **messagetext** is a pointer to a buffer containing the data for which you want to calculate the MAC.

- **msg_length** is the number of bytes of data in buffer messagetext. This must be a multiple of eight.

- **mac** is a pointer to a buffer which will receive the 8-byte MAC.

**Output**

On successful exit from this routine:

- **mac** contains the 8-byte MAC.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR** The operation was successful.
ac_lu_ks_macver - Verify a MAC using Session Key

ac_lu_ks_macver verifies a triple-DES MAC (message authentication code) using the specified user’s session key. The MAC can be any length up to the full eight bytes. The result is a boolean value indicating whether the supplied MAC matched the computed value.

The MAC is computed using an IV of X'0000000000000000'.

Function Prototype

```c
long ac_lu_ks_macver( profile_id_t profile_id,
                      UCHAR *messagetext,
                      ULONG msg_length,
                      mac_t mac,
                      USHORT mac_length,
                      boolean *verified) ;
```

Input

- **profile_id** is an 8-byte, non-NULL terminated string representing the name of the profile whose session key should be used to compute the MAC.
- **messagetext** is a pointer to a buffer containing the data on which you want to verify the MAC.
- **msg_length** is the number of bytes of data in buffer *messagetext*. This must be a multiple of eight.
- **mac** is the input MAC, which is to be compared with the calculated MAC.
- **mac_length** is the length of the *mac* parameter, in bytes. It can range from 1 to 8.

Output

On successful exit from this routine:

- **verified** is a pointer to a boolean variable which receives the verification result. The value is TRUE if the MAC verifies, and FALSE if it does not.

Return Codes

Common return codes generated by this routine are:

- **acm_NO_ERROR** The operation was successful.
**ac_lu_list_users - List the IDs of the Logged On Users**

ac_lu_list_users returns a list of the user IDs for each logged-on user.

**Function Prototype**

```c
long ac_lu_list_users( profile_id_t user_list[], ULONG *num_users);
```

**Input**

- `user_list` is an array which receives the list of logged-on users.

- `num_users` is a pointer to a variable containing the number of elements in `user_list`.

**Output**

On successful exit from this routine:

- `user_list` contains the (unordered) list of users.

- `num_users` receives the number of elements returned in the `user_list` array.

**Return Codes**

Common return codes generated by this routine are:

- `acm_NO_ERROR` The operation was successful.

- `acm_BFR_TOO_SMALL` The array is not large enough to hold the number of users.
ac_lu_query_user - Check if a User is Logged On

ac_lu_query_user determines whether a specified user is currently logged on to the coprocessor. This is an indication that the user has successfully authenticated to the coprocessor according to one of the enrolled user profiles.

Function Prototype

```c
long ac_lu_query_user( profile_id_t   profile_id,
                        boolean *logged_on);
```

Input

- `profile_id` is an 8-byte, non-NULL terminated string representing the name of the user being checked.
- `logged_on` is a pointer to a boolean variable which receives the response to your query.

Output

On successful exit from this routine:

- `ac_lu_query_user` returns a boolean value of TRUE if the user is logged on, and FALSE if not.

Return Codes

Common return codes generated by this routine are:

- `acm_NO_ERROR` The operation was successful.
**ac_query_profile - Return the Length of a User Profile**

ac_query_profile returns the length of a specified user profile, or zero if the profile does not exist on the coprocessor.

**Function Prototype**

```c
long ac_query_profile( profile_id_t profile_id, ULONG *profile_length);
```

**Input**

- `profile_id` is an 8-byte, non-NULL terminated string representing the name of the profile being checked.

- `profile_length` is a pointer to variable that will receive the profile structure length, in bytes. The length will be zero if the profile does not exist.

**Output**

On successful exit from this routine:

- `profile_length` contains the length, in bytes. If the profile does not exist, a length of zero is returned.

**Return Codes**

Common return codes generated by this routine are:

- `acm_NO_ERROR` The operation was successful.
- `acm_PROFILE_NOT_FOUND` The specified profile was not in the profile list.
**ac_query_role - Return the Length of a Role**

ac_query_role returns the length of a specified role, or zero if the role does not exist on the coprocessor.

**Function Prototype**

```c
long ac_query_role( role_id_t role_id, ULONG *role_length);
```

**Input**

- `role_id` is an 8-byte, non-NULL terminated ID of the role to be queried.
- `role_length` is a pointer to a variable which receives the length of the specified role, in bytes.

**Output**

On successful exit from this routine:

- `role_length` contains the length, in bytes. If the role does not exist, a length of zero is returned.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR**  
  The operation was successful.
- **acm_ROLE_NOT_FOUND**  
  The role does not exist in the SRDI.
**ac_reinit - Reinitialize the Access Control Manager**

*ac_reinit* reinitializes the Access Control Manager, deletes all existing roles, profiles, and other data, and reinitializes to the access control state of a new coprocessor.

**Function Prototype**

```c
long ac_reinit( void );
```

**Input**

This function has no input.

**Output**

This function has no output. On successful exit from this routine:

*ac_reinit* reinitializes the Access Control Manager.

**Return Codes**

Common return codes generated by this routine are:

- **acm_NO_ERROR**
  
  The operation was successful.
**ac_reset_logon_fail_cnt - Reset Logon Failure Count**

ac_reset_logon_fail_cnt resets the logon failure count in the specified user profile. This is the count of the number of consecutive times the user has tried unsuccessfully to authenticate themselves to the coprocessor. The count is reset to zero.

**Function Prototype**

```c
long ac_reset_logon_fail_cnt( profile_id_t profile_id);
```

**Input**

`profile_id` is an 8-byte, non-NULL terminated character string ID for the profile to be reset.

**Output**

This function returns no output. On successful exit from this routine:

- The logon failure count of the specified profile has been set to zero.

**Return Codes**

Common return codes generated by this routine are:

- `acm_NO_ERROR` — The operation was successful.
- `acmPROFILE_NOT_FOUND` — The specified profile was not in the profile list.
**acm_xlt_errorcode - Translate an ACM Error Code to a SAPI Error Code**
Translates an ACM error code (defined in *camacm.h*) to an SAPI error code (defined in *cmnerrcd.h*).

**Function Prototype**

```c
int acm_xlt_errorcode( long acm_code,
                        long sapi_code )
```

**Input**

On entry to this routine:

*acm_code* contains the error code returned by the Access Control Manager.

*sapi_code* is a pointer to a 4-byte buffer, which will hold the translated code.

**Output**

On successful exit from this routine:

acm_xlt_errorcode returns true if the provided ACM code was found, and false otherwise.

If acm_xlt_errorcode returned true, sapi_code contains the corresponding SAPI error code.

**Return Codes**

This function has no return codes.
Chapter 12.

Cache Management Functions

This section describes functions used to maintain an on-board cache of security relevant data. For example, the CCA API currently uses these functions to maintain a cache of decrypted private keys.

Note: All functions within this chapter are available on the workstation host and the coprocessor.

Header Files for Caching Functions

When using these functions, your program must include the following header file.

```c
#include "cache.h" /* Cache management functions */
```

Overview of Cache Management Functions

The cache management functions allow the operation of a cache of data in the DRAM of the coprocessor. Each entry in the cache consists of a data item and its unique identifier. The cache is indexed on two bytes of data supplied by the user, for example a predefined label or two chosen bytes of a hash of the item or two bytes of the unique identifier. The cache uses a Least Recently Used (LRU) replacement system, eliminating the item which has been unused the longest when more space is needed in the cache.

Data in the cache is stored and accessed using a two-level lookup process. An item is referenced using a "tag", which is any arbitrary-length byte string that uniquely identifies an item. In addition, every access to an item passes a 2-byte "short" tag, which should also be as unique as possible to the item being accessed. It is up to the user to decide how to create the short tags (examples might include a hash of the full tag, the first two bytes of the full tag, or two fixed-position bytes of ciphertext for items that contain encrypted data).

The short tag is broken into two separate one-byte tags, referred to as tag1 and tag2. Tag1 is used as an index into a 256-entry array, where each element either contains a pointer to a list of items that are further addressed using tag2, or NULL if the cache does not contain any items with the indexed tag1 value.

The tag2 lists are linked-lists, where each element contains the data of a cached item. The linked-list, once addressed through the tag1 array, is searched linearly for items with tag2 entries matching the passed tag2 value. For each tag2 match, the entire full tag is compared with that stored in the item, to find the one that is the desired object. (Note that it is possible for multiple items to share identical tag1 and tag2 values, although it is unlikely for good choices of tag1/ tag2 computation methods, unless the cache holds a very large number of items.)
Summary of Functions
These functions are used by the cache manager to manage the cache.

- **cache_clear**  Remove all entries from a cache.
- **cache_delete**  Remove an existing cache.
- **cache_delete_item**  Remove a specific item from the cache.
- **cache_get_item**  Retrieve an item from the cache, copying it into a user-supplied buffer.
- **cache_get_item_b**  Retrieve an item from the cache after allocating memory to hold it.
- **cache_init**  Open a new cache.
- **cache_status**  Check the status of an existing cache.
- **cache_write_item**  Store an item in the cache.
cache_clear - Remove All Data from Cache

cache_clear clears all data from the specified cache.

Function Prototype

cacheError cache_clear( cacheHandle handle )

Input

On entry to this routine:

handle must be set to a valid cacheHandle.

Output

This function returns no output. On successful exit, all data has been cleared from the specified cache.

Return Codes

Common return codes generated by this routine are:

- ce_OK                The function completed successfully.
- ce_BAD_HANDLE      The specified cache does not exist.
**cache_delete - Delete Cache**
cache_delete will remove an existing cache, recovering all memory.

**Function Prototype**
cacheError cache_delete( cacheHandle handle )

**Input**
On entry to this routine:

handle must be set to a valid cacheHandle.

**Output**
This function returns no output. On successful exit, all data has been cleared from the specified cache, the cache has been removed, and all associated memory objects have been freed.

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

- **ce_OK** The function completed successfully.
- **ce_BAD_HANDLE** The specified cache does not exist.
**cache_delete_item - Delete One Item from Cache**

Cache_delete_item deletes the specified item from the cache.

**Function Prototype**

```c
cacheError cache_delete_item( cacheHandle handle,
                               short_tag_t short_tag,
                               ULONG full_tag_length,
                               UCHAR *full_tag )
```

**Input**

On entry to this routine:

- **handle** contains the handle of the specified cache.
- **short_tag** is the two-byte tag used for indexing the cache items.
- **full_tag_length** is the length of the unique identifier of the item to be deleted.
- **full_tag** is a buffer which contains the unique identifier of the item to be deleted.

**Output**

This function returns no output. On successful completion, the data item referenced by the `short_tag` and the `full_tag` has been removed from the cache.

**Return Codes**

Common return codes generated by this routine are:

- **ce_OK** The function completed successfully.
- **ce_BAD_HANDLE** The specified cache does not exist.
- **ce_ITEM_NOT_FOUND** The specified item was not found in the cache.
Cache Management Functions

Cache Management Functions

**cache_get_item - Get an Item from Cache**
cache_get_item retrieves a specific item from the cache and copies it into a buffer supplied by the user.

**Function Prototype**

cacheError cache_get_item( cacheHandle handle,
ULONG *bfrSize,
UCHAR *bfr,
short_tag_t short_tag,
ULONG full_tag_length,
UCHAR *full_tag )

**Input**

On entry to this routine:

handle contains the handle for the specified cache.

bfrSize is a pointer to the length of the available space in the supplied buffer.

bfr is a pointer to the buffer in which the retrieved item will be returned.

short_tag is the two-byte tag which is used to index the item in the cache.

full_tag_length is the length of the unique identifier of the item in the cache.

full_tag is the unique identifier of the item to be retrieved from the cache.

**Output**

On successful exit to this routine:

bfrSize is a pointer to the length of the item which was retrieved from the cache.

bfr contains the item which was retrieved.

**Return Codes**

Common return codes generated by this routine are:

- **ce_OK**  
The function completed successfully.

- **ce_BAD_HANDLE**  
The specified cache does not exist.

- **ce_ITEM_NOT_FOUND**  
The specified item was not found in the cache.

- **ce_BFR_TOO_SMALL**  
The provided buffer was not large enough to hold the item.
**cache_get_item_b - Get an Item in a New Buffer**

`cache_get_item_b` retrieves a specific item from the cache and copies it into a buffer which is allocated by the function. This function is useful when the cache holds items with very different sizes.

**Note:** The calling function is responsible for freeing the memory allocated for the item when the item is no longer needed. However, if the function fails to complete successfully, the buffer is not allocated.

**Function Prototype**

```c
cacheError cache_get_item_b(
    cacheHandle handle,
    ULONG *bfrSize,
    UCHAR *bfr,
    short_tag_t short_tag,
    ULONG full_tag_length,
    UCHAR *full_tag )
```

**Input**
On entry to this routine:

- `handle` is the handle which specifies the required cache.
- `short_tag` is the two-byte tag used to index the item within the cache.
- `full_tag_length` is the length of the unique identifier for the item.
- `full_tag` is the unique identifier of the item.

**Output**
On successful exit from this routine:

- `bfrSize` is a pointer to the size of the buffer returned.
- `bfr` is a pointer to a buffer containing the recovered item. This buffer has been allocated by the function.

**Return Codes**

Common return codes generated by this routine are:

- **ce_OK** The function completed successfully.
- **ce_BAD_HANDLE** The specified cache does not exist.
- **ce_ITEM_NOT_FOUND** The specified item was not found in the cache.
- **ce_MEM_ALLOC_ERROR** The function was unable to allocate memory for the buffer to return the item.
cache_init - Initialize a Cache

cache_init opens a new cache, returning a handle with which to access the cache. The caller specifies the maximum size of the cache and the maximum size of the items within the cache.

Function Prototype

cacheError cache_init( ULONG maxBytes, 
ULONG maxItemBytes, 
ULONG options, 
cacheHandle *handle )

Input

On entry to this routine:

maxBytes contains the maximum number of bytes to be used in the cache. This number must be large enough to hold at least one item of maxItemBytes, along with extra room for control structures. This number determines at which point the LRU algorithm will come into play, replacing old items with new ones.

maxItemBytes contains the maximum number of bytes which an item will occupy. This value should include both the maximum size of the item and the maximum size of the unique identifier.

options must be set to one of the following values:

• 0x00000001 to enable an LRC check on the data
• 0x00000000 to disable the LRC check

handle is a pointer to an item of cacheHandle type, in which the function will return the handle for accessing the cache.

Output

On successful exit from this routine:

handle contains the cacheHandle used to access the new cache.

Return Codes

Common return codes generated by this routine are:

ce_OK The function completed successfully.

ce_ITEM_TOO_LARGE The maxItemBytes parameter is too large for the maxBytes.

ce_MEM_ALLOC_ERROR The function was unable to allocate memory for the cache overhead.

Note: If options is 0x00000001 on entry to this routine, the LRC check will be enabled for each access to each item stored in the cache. If the LRC check fails, ce_ITEM_NOT_FOUND will be returned on any attempt to access an item.
cache_status - Determine Cache Status

cache_status returns the status of the specified cache, that is bytes used by each item (including overhead), bytes used altogether, number of items currently in the cache.

Function Prototype

```c
#include "cache_sim.h"

cacheError cache_status( cacheHandle handle,
        ULONG *maxBytes,
        ULONG *maxItemBytes,
        ULONG *bytesUsed,
        ULONG *itemsStored )
```

Input

On entry to this routine:

- `handle` is the handle of the cache to be queried.
- `maxBytes` is a pointer to a long data item.
- `maxItemBytes` is a pointer to a long data item.
- `bytesUsed` is a pointer to a long data item.
- `itemsStored` is a pointer to a long data item.

Output

On successful exit from this routine:

- `maxBytes` is a pointer to the maximum number of bytes to be used in the cache, including overhead.
- `maxItemBytes` is a pointer to the maximum number of bytes each entry will use, including the data item, the unique identifier, and the overhead.
- `bytesUsed` is a pointer to the number of bytes currently used in the cache, including data and overhead.
- `itemsStored` is a pointer to the number of items which are currently stored in the cache.

Return Codes

Common return codes generated by this routine are:

- `ce_OK` The function completed successfully.
- `ce_NULL_POINTER` One of the parameters was a null pointer.
- `ce_BAD_HANDLE` The specified cache does not exist.
cache_write_item - Add an Item to Cache

cache_write_item writes an item to the specified cache. If the cache is full, the function will delete the least recently used item from the cache before storing the new item.

Function Prototype

cacheError cache_write_item(  
    cacheHandle  handle,  
    ULONG  item_length,  
    UCHAR *item,  
    short_tag_t  short_tag,  
    ULONG  full_tag_length,  
    UCHAR *full_tag )

Input

On entry to this routine:

handle is the handle which identifies the cache.

item_length is the length of the item to be stored. This length plus the full_tag_length must be smaller than the maxItemBytes specified when the cache was initialized.

item is a pointer to a buffer containing the item to be stored.

short_tag is the two-byte tag to be used to index this item within the cache.

full_tag_length is the length of the unique identifier for the item. This length plus item_length must be less than the maxItemBytes specified when the cache was initialized.

full_tag is the unique identifier for this data item.

Output

This function returns no output. On successful completion, the item has been added to the cache.

Return Codes

Common return codes generated by this routine are:

- **ce_OK**: The function completed successfully.
- **ce_NULL_POINTER**: One of the parameters was a null pointer.
- **ce_BAD_HANDLE**: The specified cache does not exist.
- **ce_ITEM_TOO_LARGE**: The sum of the full_tag_length and the item_length is greater than the maximum item size for the cache.
Chapter 13.  Miscellaneous Functions

This chapter describes functions that do not fit into any of the previously described categories.

Header Files for Miscellaneous Functions
When using these functions, your program must include the following header files.

```
#include "cassub.h"
#include "camacm.h"
#include "cmnfunct.h"
#include "cmnlibr.h"
```

Summary of Functions

- **CasGetFileDescriptor** Returns the file descriptor which has been assigned by CCA to the hardware device required for an xCrypto function.
- **CHECK_ACCESS_AUTH** Verifies the user’s authority.
- **check_access_auth_fcn** Verifies the user’s authority.
- **GetKeyLength** Returns the length of a specified key token.
- **intel_long_reverse** Byte-reverses a 4-byte block of data.
- **intel_word_reverse** Byte-reverses a 2-byte block of data.
- **LIBR_AllocStorage** Allocate memory.
- **LIBR_FreeStorage** Free unneeded memory.
- **LIBR_ReleaseSemaphore** Release a semaphore, to allow others to access the item
- **LIBR_RequestSemaphore** Wait for a semaphore to be free.
- **TOKEN_IS_A_LABEL** Determines whether a key identifier is a token or a label.
- **TOKEN_LABEL_CHECK** Determines whether a key identifier is a token or a label.
**CasGetFileDescriptor**

**Note:** This function is available on the coprocessor.

CasGetFileDescriptor returns the file descriptor which has been assigned by CCA to the hardware device required for an xCrypto function.

**Function Prototype**

```c
int CasGetFileDescriptor ( int hardware_type )
```

**Input**

On entry to this routine:

- `hardware_type` is a constant indicating which file descriptor to return. The following constants are defined:
  - `FD_XCRYPTO` is for accessing the low-battery and intrusion reset.
  - `FD_PKA` is for RSA, DSA, and modular math functions.
  - `FD_DES` is for DES and TES functions.
  - `FD_SHA` is for SHA-1 functions.
  - `FD_HWRNG` is for the Hardware random number generator.
  - `FD_OA` is for the Outbound Authentication Manager.
  - `FD_RTC` is for the Real-Time Clock.

**Output**

The return value of this function is the file descriptor assigned to the requested piece of hardware, or `NO_FILE_DESCRIPTOR` if the hardware is not available or invalid.

**Return Codes**

This function has no return codes.
**check_access_auth_fcn - Verify User Authority**

**Note:** This function is available on the coprocessor.

check_access_auth_fcn performs operations that are necessary before executing a requested CCA command.

1. It checks to see if the user who sent the request is authorized to perform the requested function. This is done by passing a function code, known as an *Access Control Point*. A user’s role contains a list of the Access Control Points corresponding to functions that the user is permitted to execute.

2. If the user is authorized to execute the command, the reply CPRB and parameter block are initialized.

The function returns a boolean value in *pGranted* to indicate whether the specified function was authorized.

**Function Prototype**

```c
ULONG check_access_auth_fcn(  CPRB_ptr  pRequestCprb,
                             CPRB_ptr  pReplyCprb,
                             role_id_t  roleID,
                             USHORT  requested_fcn_code,
                             boolean *pGranted)
```

```c
#define CHECK_ACCESS_AUTH( Rqc, Rpc, r, c, g) check_access_auth_fcn( Rqc, Rpc, r, c, g)
```

**Input**

On entry to this routine:

- *pRequestCprb* is a pointer to the request (input) CPRB structure.

- *pReplyCprb* is a pointer to a buffer which receives the initialized reply (output) CPRB structure.

- *roleID* is the eight-character Role ID defining the access control role for the user who sent this request. The Role ID is an input parameter, passed to every CCA command processor when it is called.

- *requested_fcn_code* is the Access Control Point corresponding to the CCA verb you are executing. The Access Control Manager determines if the user is allowed to execute this verb, based on whether the Access Control Point is enabled in the user’s role.

**Output**

On successful exit from this routine:

- *pGranted* is a pointer to a location where the boolean result is returned. The value stored in *pGranted* is TRUE if the user has authorization, and FALSE if not.
Notes
In a zOS system, access is granted to role IDs only through the TKE utility. UDXs built with the libcsullib.so file may have access granted to role IDs using the csulcnm utility, if new access control points are added to the file csuap.def.

This function may also be called using the macro CHECK_ACCESS_AUTH, with the same parameters as previously described.

Return Codes
Common return codes generated by this routine are:

OK The operation was successful.
acm_ROLE_NOT_FOUND The role is not in the SRDI.
GetKeyLength - Get Length of Key Token

Note: This function is available on the workstation host.

GetKeyLength returns the length of a specified key token.

Function Prototype

USHORT GetKeyLength ( UCHAR *keyid_ptr,
                       long   *key_parm_length_ptr,
                       long   *message_ptr )

Input

On entry to this routine:

keyid_ptr is a pointer to the start of the input key data.

key_parm_length_ptr is a pointer to the expected key length, if this is an RSA key token, or NULL if not an RSA token. (RSA tokens are passed to the host with a parameter length, because they are variable sized. This function returns an error if the RSA token is larger than this expected size.)

message_ptr is a pointer to an address which stores the return code.

Output

On successful exit to this routine:

GetKeyLength returns the length of the token, or -1 if an error occurred.

message_ptr contains the return code. If there is no error, this is set to S_OK (0).

Return Codes

Common return codes generated by this routine are:

ERROR The value pointed to by message_ptr is a SAPI error code, see below.
E_KEY_LEN The key has a length less than 1 byte.
E_SIZE The key is longer than the expected length in key_parm_length_ptr.
E_KEY_TOKEN keyid_ptr was not pointing at a valid key token.
**intel_long_reverse - Convert Long Values**

**Note:** This function is available on both the host and the coprocessor.

`intel_long_reverse` reverses the order of the bytes in a long (4-byte) integer. This is used to convert long values between big-endian and little-endian formats.

### Function Prototype

```c
ULONG intel_long_reverse( ULONG long_val)
```

For portability reasons, the following macros have been conditionally defined for integer translation.

```c
#ifndef HOST_BIG_ENDIAN
    #define atohs(d) intel_word_reverse((USHORT)d)
    #define htoas(d) intel_word_reverse((USHORT)d)
    #define atohl(d) intel_long_reverse((ULONG)d)
    #define htoal(d) intel_long_reverse((ULONG)d)
#else
    #define xtohl(d) ((ULONG)d)
    #define htoxl(d) ((ULONG)d)
    #define atohl(d) ((ULONG)d)
    #define htoal(d) ((ULONG)d)
#endif

#ifdef ADAPTER_BIG_ENDIAN
    #define xtoal(d) intel_long_reverse((ULONG)d)
    #define atoxl(d) intel_long_reverse((ULONG)d)
    #define ltoas(d) ((USHORT)d)
    #define ltoal(d) ((ULONG)d)
#else
    #define xtoal(d) intel_long_reverse((ULONG)d)
    #define atoxl(d) intel_long_reverse((ULONG)d)
    #define ltoal(d) intel_long_reverse((ULONG)d)
    #define stoal(d) ((ULONG)d)
#endif
```

### Input

On entry to this routine:

- `long_val` is the input value. It is reversed in byte order, and returned as the function result.

### Output

This function returns no output. On successful exit to this routine:

- `intel_long_reverse` returns the bytes from `long_val` in reverse order.

### Return Codes

This function has no return codes.
**intel_word_reverse - Convert 2-Byte Values**

*Note:* This function is available on both the host and the coprocessor.

`intel_word_reverse` reverses the order of the bytes in a word (2-bytes) of data. This is used to convert 2-byte values between big-endian and little-endian formats.

**Function Prototype**

```c
USHORT intel_word_reverse( USHORT intel_int)
```

For portability reasons, the following macros have been conditionally defined for integer translation.

```c
#ifndef HOST_BIG_ENDIAN
    #define xtohs(d) intel_word_reverse((USHORT)d)
    #define htoxs(d) intel_word_reverse((USHORT)d)
    #define atohs(d) intel_word_reverse((USHORT)d)
    #define htoas(d) intel_word_reverse((USHORT)d)
#else
    #define xtohs(d) ((USHORT)d)
    #define htoxs(d) ((USHORT)d)
    #define atohs(d) ((USHORT)d)
    #define htoas(d) ((USHORT)d)
#endif

#ifndef ADAPTER_BIG_ENDIAN
    #define xtoas(d) ((USHORT)d)
    #define atoxs(d) ((USHORT)d)
    #define ltoas(d) ((USHORT)d)
    #define stoas(d) intel_word_reverse((USHORT)d)
#else
    #define xtoas(d) intel_word_reverse((USHORT)d)
    #define atoxs(d) intel_word_reverse((USHORT)d)
    #define ltoas(d) intel_word_reverse((USHORT)d)
    #define stoas(d) ((USHORT)d)
#endif
```

**Input**

On entry to this routine:

`intel_int` is the input word. It is reversed in byte order, and returned as the function result.

**Output**

On successful exit from this routine:

`intel_word_reverse` returns the bytes from `intel_int` in reverse order.

**Return Codes**

This function has no return codes.
**LIBR_AllocStorage - Allocate Memory**

LIBR_AllocStorage allocates memory from the heap, returning a pointer to the memory.

**Note:** This function is available on the workstation host and the coprocessor.

**Function Prototype**

```c
void * LIBR_AllocStorage( ULONG   size  /* Size of storage to allocate. */)
```

**Input**

`size` is the amount of memory to be allocated.

**Output**

`LIBR_AllocStorage` returns a pointer to the allocated memory, or NULL if no memory was allocated.

**Return Codes**

This function has no return codes.
**LIBR_FreeStorage - Free Memory**

LIBR_FreeStorage de-allocates memory which was allocated by **LIBR_AllocStorage**.

**Note:** This function is available on the workstation host and the coprocessor.

**Function Prototype**

```c
void LIBR_FreeStorage( void *Address)
```

**Input**

*Address* is the address of the memory to be freed.

**Output**

This function has no output.

**Return Codes**

This function has no return codes.
**LIBR_ReleaseSemaphore**
LIBR_ReleaseSemaphore releases a semaphore, so that other processes may use the resource it protects.

**Note:** This function is available on the workstation host and the coprocessor.

**Function Prototype**

```c
void LIBR_ReleaseSemaphore( ULONG  SemHandle)
```

**Input**

`SemHandle` is the semaphore handle to be released.

**Output**

This function has no output. On successful exit, the semaphore has been released.

**Return Codes**

This function has no return codes.
**LIBR_RequestSemaphore**

LIBR_RequestSemaphore waits until a semaphore is available, to ensure that no other processes may use the resource it protects.

**Note:** This function is available on the workstation host and the coprocessor.

**Function Prototype**

```c
void LIBR_RequestSemaphore( ULONG  SemHandle)
```

**Input**

*SemHandle* is the semaphore handle which is requested.

**Output**

This function has no output. On successful exit, the semaphore has been obtained.

**Return Codes**

This function has no return codes.
**TOKEN_IS_A_LABEL - Identifies the Token as a Label**

This macro has a value of TRUE when the first byte of the key identifier input is valid for a key label. All key labels have a first byte between 0x20 and 0xFE. TOKEN_IS_A_LABEL should be used when a token is available for checking.

```c
#define TOKEN_IS_A_LABEL( keyid) \
( ( keyid [0] >= MIN_FOR_LABEL ) & & ( keyid [0] <= MAX_FOR_LABEL ) )
```
**TOKEN_LABEL_CHECK - Determine if Key Identifier is a Label**

This macro has a value of TRUE when the character input is valid for a key label. All key labels have a first byte between 0x20 and 0xFE. TOKEN_LABEL_CHECK should be used when only one byte is available for checking.

```c
#define TOKEN_LABEL_CHECK( keyid) \
( ( keyid >= MIN_FOR_LABEL ) & & ( keyid <= MAX_FOR_LABEL ) )
```
**Appendix A.**

**UDX Sample Code - Host Piece - Service**

This appendix contains a listing of the sample file `zudxsvc.bal`. This function is a skeleton for the design of the host piece of a CCA extension.

```
TITLE 'ZUDXSVC: Sample UDX - PIN Block Processing Service'
ZUDXSVC CSECT
ZUDXSVC AMODE 31
ZUDXSVC RMODE ANY

* *************************************************************************** *
*                                                                            *
*   Sample callable service for UDX. This service is for a z990.           *
*                                                                            *
*   (C) Copyright IBM Corp. 2003                                           *
*                                                                            *
*   Function: This program will process an encrypted PIN block             *
*               (assume a proprietary block form) and return the           *
*               block encrypted under the original or a second key.        *
*                                                                            *
*   Inputs:   Rule Array Count   Number of keywords passed in rule        *
*              array                                                      *
*                                                                            *
*   Rule Array     Keywords (0, 1 or 2 keywords may be passed)             *
*                                                                            *
*   Input Pin Key Id  Input PIN encrypting key identifier. The input PIN   *
*                      block is enciphered under this key. The identifier may  *
*                      be a token or label.                                  *
*                                                                            *
*   Input Pin Block  Enciphered PIN block to be processed.                 *
*                                                                            *
*   Output Pin Key Id  Output PIN encrypting key identifier or a null     *
*                      token. The identifier may be a token or label. If the  *
*                      key is not used, a null token is supplied.            *
*                                                                            *
*   Extra Data       Extra data to be used in processing the PIN block    *
*                      (always 8 bytes).                                     *
*                                                                            *
*   Processing: 1. Copy the parameter address block to the ICSF            *
*                 address space.                                           *
*                                                                            *
*   2. Copy all input parameters to the ICSF address space.               *
*                                                                            *
*   3. Validate the rule array count.                                     *
*                                                                            *
*   4. Check that the caller is authorized to use this protected resource *
*       (RACF check).                                                     *
*                                                                            *
*   5. Process the InputPinKeyId. If the identifier is a label, retrieve  *
*       the token from the CKDS.                                          *
*                                                                            *
*   6. Process the OutputPinKeyId. If the identifier is a label, retrieve  *
*       the token from the CKDS.                                          *
*                                                                            *
```

**UDX Sample Code - Host Piece - Service** page A-1
Set a flag if the identifier is a null token.

7. Call CSFGAPSI to perform the following:
   - Create the request CPRB for the function.
   - Submit the request CPRB to the PCIXCC
   - Validate the reply CPRB.

8. Parse the PIN block from the verb unique data block and copy it to the output parameter.

9. If the return/reason code indicates that a token was enciphered under the old master key, parse the key block and copy the reenciphered keys to the appropriate identifiers. Labels are not updated.

10. Return to caller.

Outputs: Return Code  Return code from processing
        Reason Code  Reason code from processing
        Output Pin Block  Processed enciphered PIN block
        Input Pin Key Id  Reenciphered token if the key was enciphered under the old master key.
        Output Pin Key Id  Reenciphered token if the key was enciphered under the old master key.

External routines:
CSFACKDS - Retrieve a DES key from the CKDS
CSFGAPSI - Invoke PCIXCC
CSFASEC - Check authorization to a RACF-protected or security-exit-protected resource

External executable macros:
CSFAGET - Obtain dynamic storage
CSFAFREE - Release dynamic storage

Register usage:
R0 = Work register
R1 = Address of parameter lists
R2-R6 = Work registers
R7 = CCVT address
R8 = SPB address
R9 = GSVT address and work register
R10 = CCVE address and work register
R11 = Dynamic data area address
R12 = Module base register
R13 = unused
R14-R15 = Linkage registers

Change History
Date    Programmer    Description
--------    ----------    -----------
09/26/99    kbb           Created
01/31/01    mce           Converted to Assembler language
08/01/03    mce           Converted for operation on z990
* ***************************************************************************** *
* EJECT
* MACRO
&LABEL   COPYPARMS  &SOURCE=,&SRCALET=,&COPYLEN=,&TARGET=,&TGTALET=,   *
&COPYDIR=
* ***************************************************************************** *
.* Macro to invoke the COPY subroutine
.* Parameters to the COPY subroutine, specified on the COPYPARMS
.* macro, are the following:
.* SOURCE  - address of the parameter from which the copy is to
.*          be performed
.* SRCALET - the ALET of the SOURCE parameter
.* COPYLEN - the length of the parameter to be copied
.* TARGET  - address of the location to where the SOURCE parameter
.*          should be copied
.* TGTALET - the ALET of the TARGET location
.* COPYDIR - the direction in which the copy should be performed
.* TO_ICSF means copy from the application address
.* space to ICSF's address space
.* TO_CALLER means copy from ICSF's address space to
.* the application address space
* ***************************************************************************** *
L     R10,&SOURCE
ST    R10,SOURCE_ADDR
LA    R10,&SRCALET
ST    R10,SOURCE_ALET
L     R10,&COPYLEN
ST    R10,COPY_LENGTH
L     R10,&TARGET
ST    R10,TARGET_ADDR
LA    R10,&TGTALET
ST    R10,TARGET_ALET
L     R10,&COPYDIR
ST    R10,COPY_DIRECTION
BAS   R14,COPY
MEND
EJECT
*
* ***************************************************************************** *
*     Main entry for module
* ***************************************************************************** *
*
MAINENT  DS    0H
USING *,R15
B     PROLOG
DC    AL1(18)
DC    C'ZUDXSVC   2003.213'
DROP  R15
PROLOG  BSM   R14,0
BAKR  R14,0
LAE   R12,0
PSTART   EQU   ZUDXSVC
USING PSTART,R12
LAE   R10,0,(R1)
L     R0,DYNDATA_SIZE
LA    R15,0
CPYA  AR1,AR12
SAC   512
CSFAGET  OBTAIN,LENGTH=(R0),SP=(R15),LINKAGE=SYSTEM
LAE   R11,0,(R1)
USING DATD,R11
EREG  R14,R1
CPYA  AR1,AR10
MVC   PARAMETER_LIST(PARMLLEN),0(R1)
LR    R8,R0
CPYA  AR8,AR12
USING SPB,R8
* ***************************************************************************** *
*     Save the address of the caller's parameter block                   *
* ***************************************************************************** *
LR    R6,R1
*
* ***************************************************************************** *
*     Clear the processing flags and local variables                    *
* ***************************************************************************** *
*
XC    PROCESSING_FLAGS(1),PROCESSING_FLAGS
SLR   R7,R7
ST    R7,LOCAL_REASON_CODE
ST    R7,LOCAL_RETURN_CODE
ST    R7,SERVICE_RC
ST    R7,SERVICE_RS
*
MAINPROCNDS10H
* ***************************************************************************** *
*     Check the environment and copy the parameters to local storage.   *
* ***************************************************************************** *
*
BAS   R14,VALIDATE_AND_COPY
L     R9,LOCAL_RETURN_CODE
LTR   R9,R9
BP    ENDMAIN
*
* ***************************************************************************** *
*     Process the key identifiers.                                     *
* ***************************************************************************** *
*
BAS   R14,PROCESS_KEYS
L     R10,LOCAL_RETURN_CODE
LTR   R10,R10
BP    ENDMAIN
*
* ***************************************************************************** *
*     Process the request.                                             *
* ***************************************************************************** *
*
BAS   R14,PROCESS_REQUEST
L     R9,LOCAL_RETURN_CODE
LTR   R9,R9
BP    ENDMAIN
*
* ***************************************************************************** *
*     Copy the output PIN block to the output parameter.               *
* ***************************************************************************** *
*
Call the COPY routine with parameters as follows:
Address of returned encrypted PIN block (PIN_BLOCK@)
ICSF's ALET
Length of LOCAL_OUTPUT_PIN_BLOCK
OUTPUT_PIN_BLOCK@
Application ALET
TO_CALLER
*
*
COPYPARMS SOURCE=PIN_BLOCK@,SRCALET=ICSF_ALET,
COPYLEN=PIN_BLOCK_LEN,TARGET=OUTPUT_PIN_BLOCK@,
TGTALET=APPL_ALET,COPYDIR=TO_CALLER
*
* ***************************************************************************** *
*     Check return and reason code to see if a token was enciphered    *
*     under the old master key. (RC=0, RS=10001)                        *
(We assume any necessary reencipherment was performed by the UDX code in the PCIXCC.) If so, copy the reenciphered tokens back to the caller.

CLC LOCAL_REASON_CODE(4),OMK_TOKEN_USED
BNE ENDMAN

Process the input pin key id first. (Only process the key if the input key identifier is not a label.)

TM PROCESSING_FLAGS,INPUT_KEY_IS_LABEL
BNZ CHKOUTKY

Copy the (possibly) reenciphered input PIN key back to the caller's parameter area. Call the COPY routine with parameters as follows:
- Address of LOCAL_INPUT_KEY_TOKEN
- ICSF's ALET
- Length of LOCAL_INPUT_KEY_TOKEN
- INPUT_PIN_KEY_ID@
- Application ALET
- TO_CALLER

LA R10,LOCAL_INPUT_KEY_TOKEN
ST R10,SOURCE_ADDRESS
COPYPARMS SOURCE=SOURCE_ADDRESS,SRCALET=ICSF_ALET,
COPYLEN=TOKEN_LENGTH,TARGET=INPUT_PIN_KEY_ID@
TGTALET=APPL_ALET,COPYDIR=TO_CALLER

Now process the output pin key id. (Only process the key if the key identifier is not a label and the output key is not null.)

CHKOUTKY TM PROCESSING_FLAGS,OUTPUT_KEY_IS_LABEL
BNZ ENDMAN
TM PROCESSING_FLAGS,OUTPUT_KEY_IS_NULL
BNZ ENDMAN

Copy the (possibly) reenciphered output PIN key back to the caller's parameter area. Call the COPY routine with parameters as follows:
- Address of LOCAL_OUTPUT_KEY_TOKEN
- ICSF's ALET
- Length of LOCAL_OUTPUT_KEY_TOKEN
- OUTPUT_PIN_KEY_ID@
- Application ALET
- TO_CALLER

LA R10,LOCAL_OUTPUT_KEY_TOKEN
ST R10,SOURCE_ADDRESS
COPYPARMS SOURCE=SOURCE_ADDRESS,SRCALET=ICSF_ALET,
COPYLEN=TOKEN_LENGTH,TARGET=OUTPUT_PIN_KEY_ID@
TGTALET=APPL_ALET,COPYDIR=TO_CALLER

ENDMAIN EQU *
* Return to the caller with return reason in register 0.
*................................................................. *

Set RETURN_CODE equal to LOCAL_RETURN_CODE
L    R14,LOCAL_RETURN_CODE
LA   R10,APPL_ALET
SAR  AR15,R10
L    R15,RETURN_CODE@
ST   R14,0(,R15)

Set REASON_CODE equal to LOCAL_REASON_CODE
L    R15,LOCAL_REASON_CODE
SAR  AR10,R10
L    R10,REASON_CODE@
ST   R15,0(,R10)

Free dynamic storage and return to caller with return code
in register 15.
LR   R10,R14   Save return code around call
LR   R3,R15   Save reason code around call
CPYA AR3,AR0
L    R0,DYNDATA_SIZE   Size of area to free
LA   R15,0
LR   R1,R11
CPYA AR1,AR11
CSFAFREE RELEASE,LENGTH=(R0),ADDR=(R1),SP=(R15),LINKAGE=SYSTEM
LR   R0,R3   Reason code into register 0
CPYA AR0,AR3
LR   R15,R10   Return code into register 10
PR
EJECT

***************************************************************************** *

* Subroutines *
*              *

***************************************************************************** *

* Validate and Copy *
*                     *
* Explicit Inputs: none *
*                     *
* Implicit Inputs: CCVT   Crypto communication vector table *
*                      *
*                      PARAMETER_LIST *
*                      *
*                      All input parameters *
*                      *
* Process: 1. Validate that ICSF is running on CMOS hardware. *
*                     *
* 2. Check that at least one PCXICC is active. *
*                     *
* 3. Call RACF (or equivalent) to see if the caller is *
*    authorized to run this program. *
*                     *
* 4. Copy the parameter address list into ICSF storage *
*    using the caller's key. *
*                     *
* 5. Copy the input parameters to local storage using *
*    the caller’s key. *
*                     *
* Explicit Outputs: None *
*                     *
* Implicit Outputs: LOCAL_RETURN_CODE *
*                     *
* LOCAL_REASON_CODE

***************************************************************************** *
LOCAL_RULE_ARRAY_COUNT
LOCAL_RULE_ARRAY
LOCAL_INPUT_KEY_ID
LOCAL_INPUT_PIN_BLOCK
LOCAL_EXTRA_DATA
LOCAL_OUTPUT_KEY_ID

VALIDATE_AND_COPY EQU *
    STM   R14,R12,SAVEAREA
    STAM  AR14,AR12,SAVEAREA+60

* Check the environment.
* Check that ICSF is active (Bit CCVTMK in the CCVT is equal to 1)

L     R7,SPBCCVT
    USING CCVT,R7
    CPYA  AR7,AR12
    TM    CCVTSFG1,CCVTMK
    BO    CHKCCP
    * If ICSF is not active, set LOCAL_RETURN_CODE to 12 and
    * LOCAL_REASON_CODE to 0
    LA    R10,RC_CSF_ERROR
    ST    R10,LOCAL_RETURN_CODE
    SLR   R10,R10
    ST    R10,LOCAL_REASON_CODE
    B     ENDVAL

* Check that at least one CCP is available for the service.
* (Bit CCVTCCP in the CCVT is equal to 1)

CHKCCP EQU *
    TM    CCVTFLAG,CCVTCCP
    BO    CHKAUTH
    * If no CCP is available, set LOCAL_RETURN_CODE to 12 and
    * LOCAL_REASON_CODE to 11060
    LA    R10,RC_CSF_ERROR
    ST    R10,LOCAL_RETURN_CODE
    MVC   LOCAL_REASON_CODE(4),RS_12_CCP_NOT_AVAILABLE
    B     ENDVAL

* Call CSFASEC to see if the caller is authorized to use this program.
* Call CSFASEC with parameters as follows:
* LOCAL_RETURN_CODE
* LOCAL_REASON_CODE
* RESOURCE_NAME
* RESOURCE_LENGTH
* ASECS_CSFSERV
* SPB
* ***************************************************************************** *
* CHKAUTH EQU *
* XC PARMS_FOR_CALL,PARMS_FOR_CALL
LA R7,LOCAL_RETURN_CODE
ST R7,PARMS_FOR_CALL
* LA R9,LOCAL_REASON_CODE
ST R9,PARMS_FOR_CALL+4
LA R10,RESNAME
ST R10,PARMS_FOR_CALL+8
LA R7,RESLEN
ST R7,PARMS_FOR_CALL+12
LA R9,RESCLASS
ST R9,PARMS_FOR_CALL+16
ST R8,PARMS_FOR_CALL+20           SPB pointer
L R7,SPBCCVT
L R10,CCVTCCVE
USING CCVE,R10
SLR R9,R9
SAR AR10,R9
L R9,CCVEGGSVT
CPYA AR9,AR10
USING GSVT,R9
L R15,GSVT_ASEC
LAE R1,PARMS_FOR_CALL
BALR R14,R15                 Branch to CSFASEC
* If CSFASEC returns a return code greater than zero,
* return to caller with CSFASEC's return code.
L R7,LOCAL_RETURN_CODE
LTR R7,R7
BP ENDVL
*
* ***************************************************************************** *
* Copy the parameter address list into ICSF's address space. *
* ***************************************************************************** *

SLR R1,R1
IC R1,SPBPSWKY    PSW key into register 1
LA R2,PARAMETER_LIST  Target address
SLR R3,R3
SAR AR2,R3    Target space
LR R4,R6    Source address (address of
caller's parameter block)
LA R5,APPL_ALET  Caller's ALET
SAR AR4,R5  Target space
LA R0,PARMLLEN Length in register 0
BCTR R0,0
OI SPBF1,SPBTERM Set recovery flag
MVCSPK 0(R2),0(R4)
NI SPBF1,X'FF'--SPBTERM Reset recovery flag
*
* ***************************************************************************** *
* Copy the parameters to local storage. *
* ***************************************************************************** *
* Call the COPY routine with parameters as follows: *
* Address of caller's parameter to be copied *
* Application ALET *
* Length of local area to hold copied parameter *
* Address of local area to hold the copied parameter *
* ICSF's ALET *
* Direction to copy: TO_ICSF *
* ***************************************************************************** *
*
* Copy the Rule Array Count

```
LA    R10,LOCAL_RULE_ARRAY_COUNT
ST    R10,TARGET_ADDRESS
COPYPARMS  SOURCE=RULE_ARRAY_COUNT@,SRCALET=APPL_ALET, *
          COPYLEN=WORDSIZE,TARGET=TARGET_ADDRESS, *
          TGTALET=ICSF_ALET,COPYDIR=TO_ICSF

```

* Validate the rule array count.

```
L     R10,LOCAL_RULE_ARRAY_COUNT
LTR   R10,R10
BM    BADCOUNT
LA    R9,MAX_RULE_COUNT
CR    R10,R9
BNH   COPYRA
BADCOUNT DS    0H

* Return to caller with return code = 8, reason code = 2012
LA    R10,RC_APPLICATION_ERROR
ST    R10,LOCAL_RETURN_CODE
LA    R10,RS_8_IV_RA_COUNT
ST    R10,LOCAL_REASON_CODE
B     ENDVAL

```

* Copy the Rule Array

```
COPYRA   EQU   *
L     R9,LOCAL_RULE_ARRAY_COUNT
LTR   R9,R9
BZ    COPYIPIN
      No rules
SLA   R9,3
      Multiply LOCAL_RULE_ARRAY_COUNT
      by 8
ST    R9,RULES_LENGTH
LA    R10,LOCAL_RULE_ARRAY
ST    R10,TARGET_ADDRESS
COPYPARMS  SOURCE=RULE_ARRAY@,SRCALET=APPL_ALET, *
          COPYLEN=RULES_LENGTH,TARGET=TARGET_ADDRESS, *
          TGTALET=ICSF_ALET,COPYDIR=TO_ICSF

```

* Input PIN Key Id

```
COPYIPIN EQU   *
LA    R10,LOCAL_INPUT_PIN_ID
ST    R10,TARGET_ADDRESS
COPYPARMS  SOURCE=INPUT_PIN_KEY_ID@,SRCALET=APPL_ALET, *
          COPYLEN=TOKEN_LENGTH,TARGET=TARGET_ADDRESS, *
          TGTALET=ICSF_ALET,COPYDIR=TO_ICSF

```

* Input PIN Block

```
COPYIPBK EQU   *
LA    R10,LOCAL_INPUT_PIN_BLOCK

```

UDX Sample Code - Host Piece - Service  page A-9
ST R10,TARGET_ADDRESS
COPYPARMS SOURCE=INPUT_PIN_BLOCK@,SRCALET=APPL_ALET,*
    COPYLEN=PIN_BLOCK_LEN,TARGET=TARGET_ADDRESS,*
    TGTALET=ICSF_ALET,COPYDIR=TO_ICSF
*
*     Extra Data
*     ***************************************************************************** *
*
LA  R10,LOCAL_EXTRA_DATA
ST  R10,TARGET_ADDRESS
COPYPARMS SOURCE=EXTRA_DATA@,SRCALET=APPL_ALET,*
    COPYLEN=EXTRA_DATA_LEN,TARGET=TARGET_ADDRESS,*
    TGTALET=ICSF_ALET,COPYDIR=TO_ICSF
*
*     Output PIN Key Id
*     ***************************************************************************** *
*
COPY_PIN EQU *
LA  R10,LOCAL_OUTPUT_KEY_ID
ST  R10,TARGET_ADDRESS
COPYPARMS SOURCE=OUTPUT_PIN_KEY_ID@,SRCALET=APPL_ALET,*
    COPYLEN=TOKEN_LENGTH,TARGET=TARGET_ADDRESS,*
    TGTALET=ICSF_ALET,COPYDIR=TO_ICSF
MVC LOCAL_OUTPUT_KEY_TOKEN,LOCAL_OUTPUT_KEY_ID
*
ENDVAL DS 0H
LM  R14,R12,SAVEAREA
LAM AR14,AR12,SAVEAREA+60
BR R14 Return to caller
*
*     Process Keys
*     ***************************************************************************** *
*
  Function: Check the input key identifiers. Retrieve the token
  from the CKDS if the identifier is a label.
*
  Explicit Inputs: None
*
  Implicit Inputs: INPUT_PIN_KEY_IDENTIFIER
  OUTPUT_PIN_KEY_IDENTIFIER
*
  Process: 1. Check the INPUT_PIN_KEY_IDENTIFIER.
  2. Check the OUTPUT_PIN_KEY_IDENTIFIER.
*
  Explicit Outputs: None
*
  Implicit Outputs: LOCAL_RETURN_CODE
  LOCAL_REASON_CODE
  LOCAL_INPUT_KEY_TOKEN
  LOCAL_OUTPUT_KEY_TOKEN
  OUTPUT_KEY_IS_LABEL
  INPUT_KEY_IS_LABEL
  OUTPUT_KEY_IS_NULL
  *
PROCESS_KEYS EQU *

STM R14,R12,SAVEAREA

STAM AR14,AR12,SAVEAREA+60

* Process the input PIN encrypting key. Check to see if the identifier is a label. The first character must be greater than a blank if it is a label.

LA R9,LOCAL_INPUT_KEY_ID
SLR R10,R10
SAR AR9,R10
CLI 0(R9),FIRST_LABEL_CHAR
BNH NOTLABEL
OI PROCESSING_FLAGS,INPUT_KEY_IS_LABEL Set flag

* Identifier is a label, call CSFACKDS to retrieve the key.

* Call CSFACKDS with parameters as follows:
* LOCAL_RETURN_CODE
* LOCAL_REASON_CODE
* NULL_EXIT_LENGTH
* NULL_EXIT_DATA
* ACKDS_ENTRY_TOKEN
* LOCAL_INPUT_KEY_ID
* ACKDS_TYPE_ANY
* LOCAL_INPUT_KEY_TOKEN
* SPB

XC PARMS_FOR_CALL,PARMS_FOR_CALL
LA R10,LOCAL_RETURN_CODE
ST R10,PARMS_FOR_CALL
LA R10,LOCAL_REASON_CODE
ST R10,PARMS_FOR_CALL+4
LA R10,NULL_EXIT_LENGTH
ST R10,PARMS_FOR_CALL+8
LA R10,NULL_EXIT_DATA
ST R10,PARMS_FOR_CALL+12
LA R10,ACKDS_ENTRY_TOKEN
ST R10,PARMS_FOR_CALL+16
ST R9,PARMS_FOR_CALL+20
LA R9,TYPEANY
ST R9,PARMS_FOR_CALL+24
LA R10,LOCAL_INPUT_KEY_TOKEN
ST R10,PARMS_FOR_CALL+28
ST R8,PARMS_FOR_CALL+32
L R7,SPBCCVT
L R10,CCVTCCVE
SLR R9,R9
SAR AR10,R9
L R9,CCVEGSVT
CPYA AR9,AR10
L R15,GSVT_ACKDS
LAE R1,PARMS_FOR_CALL
BALR R14,R15 Call CSFACKDS

* If CSFACKDS returns a return code greater than zero,
* return to caller with CSFACKDS's return code.
NOTLABEL L R9,LOCAL_RETURN_CODE
LTR R9,R9
BP ENDPREFIX

* Process the output PIN encrypting key. Check to see if the
* identifier is a null token. The first character will be '00'x.

* LA R9,LOCAL_OUTPUT_KEY_ID
CLI 0(R9),X'00'
BNE NOTNULL
OI PROCESSING_FLAGS,OUTPUT_KEY_IS_NULL Set flag
B ENDPREFIX

* NOTNULL DS 0H

* Check to see if the identifier is a label. The first
* character must be greater than a blank.

* CLI 0(R9),FIRST_LABEL_CHAR
BNH ENDPREFIX
OI PROCESSING_FLAGS,OUTPUT_KEY_IS_LABEL Set flag

* Identifier is a label, call CSFACKDS to retrieve the key.

* Call CSFACKDS with parameters as follows:
* LOCAL_RETURN_CODE
* LOCAL_REASON_CODE
* NULL_EXIT_LENGTH
* NULL_EXIT_DATA
* ACKDS_ENTRY_TOKEN
* LOCAL_OUTPUT_KEY_ID
* ACKDS_TYPE_ANY
* LOCAL_OUTPUT_KEY_TOKEN
* SPB

* XC PARMS_FOR_CALL,PARMS_FOR_CALL
LA R10,LOCAL_RETURN_CODE
ST R10,PARMS_FOR_CALL
LA R10,LOCAL_REASON_CODE
ST R10,PARMS_FOR_CALL+4
LA R10,NULL_EXIT_LENGTH
ST R10,PARMS_FOR_CALL+8
LA R10,NULL_EXIT_DATA
ST R10,PARMS_FOR_CALL+12
LA R10,ACKDS_ENTRY_TOKEN
ST R10,PARMS_FOR_CALL+16
ST R9,PARMS_FOR_CALL+20
LA R9,TYPENY
ST R9,PARMS_FOR_CALL+24
LA R10,LOCAL_OUTPUT_KEY_TOKEN
ST R10,PARMS_FOR_CALL+28
ST R8,PARMS_FOR_CALL+32
L R7,SPBCCVT
L R10,CCVTCCVE
SLR R9,R9
SAR AR10,R9
L R9,CCVEGSVT
CPYA R9,R10
L R15,GSVT_ACKDS
LAE R1,PARMS_FOR_CALL
BALR R14,R15
ENDPKEY DS 0H
LM R14,R12,SAVEAREA
LAM AR14,AR12,SAVEAREA+60
BR R14
*
***************************************************************************** *
* Process Request                                                *
*                                                                 *
* Explicit Inputs: none                                          *
*                                                                 *
* Implicit Inputs: LOCAL_RULE_ARRAY_COUNT                        *
*                                                                 *
*                      LOCAL_RULE_ARRAY                              *
*                                                                 *
*                      LOCAL_EXTRA_DATA                            *
*                                                                 *
*                      LOCAL_INPUT_PIN_BLOCK                         *
*                                                                 *
*                      LOCAL_INPUT_KEY_TOKEN                         *
*                                                                 *
*                      LOCAL_OUTPUT_KEY_TOKEN                        *
*                                                                 *
* ** Process: Call CSFGAPSI to perform the following:               *
*                                                                 *
*              1. Create the CPRB for the request.                   *
*                                                                 *
*              2. Submit the request to the PCIXCC.                  *
*                                                                 *
*              3. Validate the reply CPRB.                           *
*                                                                 *
*              4. Parse the reply parameter block.                   *
*                                                                 *
* Explicit Outputs: None                                         *
*                                                                 *
* Implicit Outputs: LOCAL_RETURN_CODE                            *
*                                                                 *
*                      LOCAL_REASON_CODE                            *
*                                                                 *
*                      PIN_BLOCK@                                   *
*                                                                 *
*                      LOCAL_INPUT_KEY_TOKEN                        *
*                                                                 *
*                      LOCAL_OUTPUT_KEY_TOKEN                       *
*                                                                 *
***************************************************************************** *
* PROCESS_REQUEST EQU   *
STM R14,R12,SAVEAREA
STAM AR14,AR12,SAVEAREA+60
*
SLR R9,R9
ST R9,PIN_BLOCK@ Clear variable to hold address of
  returned PIN Block
***************************************************************************** *
* Set up parameters for CSFGAPSI invocation                      *
***************************************************************************** *
* Set flags to zero
  XC GAPSI_FLAGS,GAPSI_FLAGS
*  Set PCIXCC_INDEX to -1 (ANY PCIXCC)
  SLR R10,R10
  BCTR R10,0
  ST R10,PCIXCC_INDEX
*     Set PCIXCC_SERIAL_NUMBER to "NOT APPL"
   MVC   PCIXCC_SERIAL_NUMBER(8),SERIAL_NUMBER_NOT_APPLICABLE
*
*     Set PCIXCC_DOMAIN to -1 (Not Applicable)
   SLR   R10,R10
   BCTR  R10,0
   ST    R10,PCIXCC_DOMAIN
*
*     Set ALET to ICSF's ALET
   LA    R10,ICSF_ALET
   ST    R10,SOURCE_ALET
*
*     Set up VUD List
   LA    R10,VUD_ELEMENT_LENGTH
   ST    R10,VUD1_LENGTH
   ST    R10,VUD2_LENGTH
   L     R10,VUD_FLAGS
   ST    R10,VUD1_FLAG
   ST    R10,VUD2_FLAG
   LA    R10,LOCAL_INPUT_PIN_BLOCK
   ST    R10,VUD1_STRING
   LA    R10,LOCAL_EXTRA_DATA
   ST    R10,VUD2_STRING
*
*     Set up KEY List
   L     R10,TOKEN_LENGTH
   ST    R10,KEY1_LENGTH
   ST    R10,KEY2_LENGTH
   L     R10,KEY_FLAGS
   ST    R10,KEY1_FLAG
   ST    R10,KEY2_FLAG
   LA    R10,LOCAL_INPUT_KEY_TOKEN
   ST    R10,KEY1_STRING
   LA    R10,LOCAL_OUTPUT_KEY_TOKEN
   ST    R10,KEY2_STRING
*
   LA    R10,UDX_KEYNUM
   ST    R10,NUMBER_OF_KEYS
   XC    RETURNED_KEY_BLOCK(138),RETURNED_KEY_BLOCK
   LA    R9,RETURNED_KEY_BLOCK
   ST    R9,RETURNED_KEY_BLOCK_ADDRESS
*
   LA    R10,UDX_VUDNUM
   ST    R10,NUMBER_OF_VUDS
   XC    RETURNED_VUD_BLOCK(10),RETURNED_VUD_BLOCK
   LA    R10,RETURNED_VUD_BLOCK
   ST    R10,RETURNED_VUD_BLOCK_ADDRESS
*
* ***************************************************************************** *
*     Submit the request.                                            *
* ***************************************************************************** *
*
   LA    R10,SERVICE_RC
   ST    R10,GAPSI_RC_@
   LA    R10,SERVICE_RS
   ST    R10,GAPSI_RS_@
   LA    R10,GAPSI_FLAGS
   ST    R10,GAPSI_FLAGS_@
   LA    R10,FUNCTION_CODE_UDX390
   ST    R10,GAPSI_SUBFUNC_@
   LA    R10,PCIXCC_INDEX
   ST    R10,GAPSI_CCP_INDEX_@
   LA    R10,PCIXCC_SERIAL_NUMBER
   ST    R10,GAPSI_CCP_SN_@
LA    R10,PCIXCC_DOMAIN
ST    R10,GAPSI_CCP_DOM_
LA    R10,LOCAL_RULE_ARRAY_COUNT
ST    R10,GAPSI_RULES_COUNT_
LA    R10,LOCAL_RULE_ARRAY
ST    R10,GAPSI_RULES_
LA    R10,NUMBER_OF_VUDS
ST    R10,GAPSI_NUM_VUDS_
LA    R10,VUD_LIST
ST    R10,GAPSI_VUD_LIST_PARM_
LA    R10,NUMBER_OFKEYS
ST    R10,GAPSI_NUM_KEYS_
LA    R10,KEY_LIST
ST    R10,GAPSI_KEY_LIST_PARM_
LA    R10,RETURNED_VUD_BLOCK_LENGTH
ST    R10,GAPSI_VUD_BLOCK_LEN_
LA    R10,RETURNED_VUD_BLOCK_ADDRESS
ST    R10,GAPSI_VUD_BLOCK_PTR_
LA    R10,RETURNED_KEY_BLOCK_LENGTH
ST    R10,GAPSI_KEY_BLOCK_LEN_
LA    R10,RETURNED_KEY_BLOCK_ADDRESS
ST    R10,GAPSI_KEY_BLOCK_PTR_
LA    R10,APSI_NULL_LENGTH
ST    R10,GAPSI_REQ_DATA_BLK_LEN_
LA    R10,APSI_NULL_POINTER
ST    R10,GAPSI_REQ_DATA_BLK_PTR_
LA    R10,APSI_DEFAULT_ALET
ST    R10,GAPSI_REQ_DATA_BLK_ALET_
LA    R10,APSI_NULL_LENGTH
ST    R10,GAPSI_REP_DATA_BLK_LEN_
LA    R10,APSI_NULL_POINTER
ST    R10,GAPSI_REP_DATA_BLK_PTR_
LA    R10,APSI_DEFAULT_ALET
ST    R10,GAPSI_REP_DATA_BLK_ALET_
ST    R8,GAPSI_SPB_
L    R7,SPBCCVT
L    R10,CCVTCCVE
SLR  R9,R9
SAR  AR10,R9
L    R9,CCVEGSVT
CPYA R9,R10
L    R15,GSVT_GAPSI
LAEE R1,GAPSI_PARMS
BALR R14,R15
*/
* Check return code from CSFGAPSI
L    R10,SERVICE_RC
LTR  R10,R10
BNZ  GAPSI_ERR
*/

*****************************************************************************
* Process the output if the service completed successfully.
* Parse the output PIN block from the VUD.
* VUD structure
* ------------------------
* | overall | output pin | The PIN block is eight bytes long.
* | length | block | The overall length should be 10.
* ------------------------
* 2 bytes   8 bytes
*****************************************************************************
* Get address of returned encrypted PIN Block from VUD
* LA R10,RET_ENCRYPTED_PIN_BLOCK
  ST R10,PIN_BLOCK@
*
* ***************************************************************************** *
*                                                                    *
*    If either key was enciphered under the old master key,          *
*    the reenciphered key will be returned in the key block,         *
*    with the input PIN key first and the output PIN key second.     *
*    Check to see if the input PIN key was updated. If the token     *
*    returned is a null token, then the key was not updated.         *
*                                                                    *
* ***************************************************************************** *
*
CLI RET_KEY1,X'00'
BE CHKOPKEY
MVC LOCAL_INPUT_KEY_TOKEN(64),RET_KEY1
*
* ***************************************************************************** *
*                                                                    *
*    Check to see if the output PIN key was updated. If the          *
*    token returned is a null token, then the key was not            *
*    updated.                                                        *
* ***************************************************************************** *
*
CHKOPKEY DS 0H
CLI RET_KEY2,X'00'
BE KEYSDONE
MVC LOCAL_OUTPUT_KEY_TOKEN(64),RET_KEY2
*
GAPSI_ERR DS 0H
KEYSDONE DS 0H
* ***************************************************************************** *
*                                                                    *
*    Save the return and reason codes.                               *
*                                                                    *
* ***************************************************************************** *
L R10,SERVICE_RC
ST R10,LOCAL_RETURN_CODE
L R10,SERVICE_RS
ST R10,LOCAL_REASON_CODE
* *
* ***************************************************************************** *
*    Return to caller                                                *
* ***************************************************************************** *
*
ENDPRREQ DS 0H
LM R14,R12,SAVEAREA
LAM AR14,AR12,SAVEAREA+60
BR R14
*
*
*****************************************************************************
* Copy *
* *
* Function: Copy storage between the caller's address space and     *
* ICSF's address space in either direction. This                    *
* routine can copy any length of storage. The 'move with source key  *
* (MVCSK)' and 'move with destination key (MVCDK)' instructions are   *
* used.                                                            *
*
* Notes: When copying back to the caller, the target address is     *
* the original address copied from the parameter block.             *
*
* Explicit Inputs: SOURCE_ADDRESS   Address of the source data   *
* SOURCE_ALET       ALET of the source data              *
* COPY_LENGTH       Length of the data                   *
* TARGET_ADDRESS    Address of the target data           *
* TARGET_ALET      ALET of the target data              *
* COPY_DIRECTION   Flag indicating which direction to copy the data. *
* Implicit Inputs: SPB       Secondary parameter block.   *
* Process: Copy the data from the source ALET to the target ALET   *
* 256 bytes at a time. If copying to the caller's ALET,   *
* the parameters are compared. If they are equal, then   *
* the parameter is NOT moved back to user storage. This   *
* allows the user to pass in write-protected, read-only   *
* storage as a parameter.                                 *
* Explicit Outputs: None                                     *
* Implicit Outputs: None                                     *
***************************************************************************** *

COPY     EQU *
STM   R14,R12,COPYSAVE
STAM  AR14,AR12,COPYSAVE+60
OI    DO_COPY,COPY_YES   Initialize flag to require a   *
* copy. (A copy is always   *
* required if the copy direction   *
* is TO_ICSF.)                    *
* *
L     R9,COPY_LENGTH       Length to copy   *
SLR   R1,R1   
IC    R1,SPBPSWKY          PSW key in register 1   *
L     R2,TARGET_ADDR      *
L     R5,TARGET_ALET   *
SAR   AR2,R5          *
L     R4,SOURCE_ADDR      *
L     R10,SOURCE_ALET     *
SAR   AR4,R10
*   
***************************************************************************** *
*   If copying back to caller ALET, compare the storage            *
*   (There is no need to perform the copy if the area has           *
*    not changed.)                                                  *
***************************************************************************** *
*   See if direction to copy is "TO_CALLER"                        *
*   
L     R10,COPY_DIRECTION   
LTR   R10,R10          
BNZ   ENDCOMP
LR    R3,R9   Length to compare   
LR    R5,R9   Length to compare   
NI    DO_COPY,COPY_NO    Initialize flag   
CLCL  R2,R4    Compare operands   
BC    8,ENDCOMP    No copy necessary if equal   
OI    DO_COPY,COPY_YES   Set copy flag if not equal   
ENDCOMP DS 08
*
* Check if a copy operation is required. A copy must be performed if
* the DO_COPY flag is on. Reset the copy registers, which were changed by the compare.

* TM DO_COPY,COPY_YES
  L R2, TARGET_ADDR
  L R4, SOURCE_ADDR
  BNO ENDCOPY
  B TESTLEN

* MOVELOOP DS 0H

* ***************************************************************************** *
*      Set amount to move. The amount is the lesser of
*      the LOCAL_COPY_LENGTH or the MAXIMUM_MOVE amount (256 bytes).
*      If the LOCAL_COPY_LENGTH is greater than 256 bytes, we will
*      loop doing the move for 256 bytes at a time until the
*      entire amount has been copied.
* ***************************************************************************** *

* LA R0,MAXIMUM_MOVE
  CR R0,R9
  BNH USEMAX

USEMAX   BCTR R0,0

* ***************************************************************************** *
*     MVCxK instruction only moves 'MAXIMUM_MOVE' bytes at a
*     time. The instructions require the following input:                *
*     Register 0 is the length. Register 1 is the storage key         *
*     (xxxxxxKx). Register 2 is the target storage. Register 4       *
*     is the source storage.                                          *
* ***************************************************************************** *

* OI SPBF1,SPBTERM   Set recovery flag
  LA R3,COPY_DIRECTION
  CLC 0(4,R3),TO_ICSF
  BNE TOCALLER

TOICSF   DS 0H     Copy to ICSF's storage
  MVCXK 0(R2),0(R4)
  B MVCDONE

TOCALLER DS 0H     Copy to caller's storage
  MVCDK 0(R2),0(R4)

MVCDONE NI SPBF1,X'FF'-SPBTERM   Reset recovery flag

* Set up to do another MVCxK if necessary

* LA R5,MAXIMUM_MOVE
  ALR R2,R5
  ALR R4,R5
  LA R3,MAXIMUM_MOVE
  LCR R3,R3
  ALR R9,R3

TESTLEN LTR R9,R9     Is there more to move?
  BP MOVELOOP

* ENDCOPY DS 0H
  LM R14,R12,COPYSAVE
  LAM AR14,AR12,COPYSAVE+60
  BR R14

* DS 0H
  EJECT

* ***************************************************************************** *

UDX Sample Code - Host Piece - Service  page A-18
* ***************************************************************************** *
* LOCAL NON-VARIABLE FIELDS                              *
* ***************************************************************************** *
* TO_ICSF                      DC   F'1'
* TO_CALLER                    DC   F'0'
* PIN_BLOCK_LEN                DC   F'8'
* EXTRA_DATA_LEN               DC   F'8'
* TOKEN_LENGTH                 DC   F'64'
* WORDSIZE                     DC   F'4'
* RESNAME                      DC   CL8'ZUDXSVC '
* RESLEN                       DC   F'7'
* RESCLASS                     DC   CL8'CSFSERV '
* RS_12_CPP_NOT_AVAILABLE      DC   F'11060'
* ACKDS_ENTRY_TOKEN            DC   F'1'  CSFACKDS is to return a token
* TYPEANY                      DC   CL8'ANY '
* FUNCTION_CODE_UDX390         DC   XL2'5852'
* OMK_TOKEN_USED               DC   F'10001'
* RETURNED_VUD_BLOCK_LENGTH    DC   F'10'
* RETURNED_KEY_BLOCK_LENGTH    DC   F'138'
* NULL_EXIT_LENGTH             DC   F'0'
* NULL_EXIT_DATA                DC   F'0'
* SERIAL_NUMBER_NOT_APPLICABLE DC   CL8'NOT APPL'
* VUD_FLAGS                    DC   XL4'FFFFFFFF'
* KEY_FLAGS                    DC   XL4'00000000'
* APSI_NULL_LENGTH             DC   F'0'
* APSI_NULL_POINTER            DC   F'0'
* APSI_DEFAULT_ALET            DC   F'0'
* *
* ***************************************************************************** *
* REGISTER EQUATES             *
* ***************************************************************************** *
* R0    EQU      0
* R1    EQU      1
* R2    EQU      2
* R3    EQU      3
* R4    EQU      4
* R5    EQU      5
* R6    EQU      6
* R7    EQU      7
* R8    EQU      8
* R9    EQU      9
* R10   EQU     10
* R11   EQU     11
* R12   EQU     12
* R13   EQU     13
* R14   EQU     14
* R15   EQU     15
* *
* ***************************************************************************** *
* ACCESS REGISTER EQUATES      *
* ***************************************************************************** *
* A0    EQU      0
* A1    EQU      1
* A2    EQU      2
* A3    EQU      3
* A4    EQU      4
AR5   EQU      5  
AR6   EQU      6  
AR7   EQU      7  
AR8   EQU      8  
AR9   EQU      9  
AR10  EQU     10  
AR11  EQU     11  
AR12  EQU     12  
AR13  EQU     13  
AR14  EQU     14  
AR15  EQU     15  
*  
* ***************************************************************************** *
*            CONSTANT EQUATES                                        *
* ***************************************************************************** *
*  
RC_APPLICATION_ERROR    EQU 8  
RC_CSF_ERROR            EQU 12  
RS_12_SERV_NOTAVAIL     EQU 8  
RS_8_IV_RA_COUNT        EQU 2012  
APPL_ALET               EQU 1  
ICSF_ALET               EQU 0  
MAX_RULE_COUNT          EQU 2  
COPY_NO                 EQU B'01111111'  
COPY_YES                EQU B'10000000'  
MAXIMUM_MOVE            EQU 256  
FIRST_LABEL_CHAR        EQU X'40'  
OUTPUT_KEY_IS_NULL      EQU X'80'  
INPUT_KEY_IS_LABEL      EQU X'40'  
OUTPUT_KEY_IS_LABEL     EQU X'20'  
UDX_VUDNUM              EQU 2  
UDX_KEYNUM              EQU 2  
VUD_ELEMENT_LENGTH      EQU 8  
*  
LTORG  
*  
EJECT  
*  
* ***************************************************************************** *
*            Dynamic Data definitions                                *
* ***************************************************************************** *
*  
DATD     DSECT  
DS    0F  
**  
***          Saveareas  
**  
SAVEAREA DS 30F  
COPYSAVE DS 30F  
**  
***          Parameter lists  
**  
*  
COPY_PLIST DS 6F  Parameter list for COPY_PARMS  
ORG   COPY_PLIST  
SOURCE_ADDR DS A  
SOURCE_ALET DS A  
COPY_LENGTH DS A  
TARGET_ADDR DS A  
TARGET_ALET DS A  
COPY_DIRECTION DS A  
*  
PARMS_FOR_CALL DS 17A  Parameter list used by internal calls  
*  
GAPSI_PARMS DS 0F  Parameter list to call CSFGAPSI
RULES_LENGTH      DS F
GAPSI_FLAGS       DS BL32
NUMBER_OF_VUDS   DS F
NUMBER_OF_KEYS   DS F
DO_COPY          DS BL8
RETURNED_KEY_BLOCK_ADDRESS DS A
RETURNED_VUD_BLOCK_ADDRESS DS A
*
RETURNED_VUD_BLOCK DS CL10
   ORG   RETURNED_VUD_BLOCK
RET_VUD_LENGTH DS CL2
RET_ENCRYPTED_PIN_BLOCK DS CL8
   ORG   RETURNED_VUD_BLOCK+10
*
RETURNED_KEY_BLOCK DS CL138
   ORG   RETURNED_KEY_BLOCK
RET_KEY__OVERALL_BLK_LEN DS CL2
RET_KEY1_LENGTH DS CL2
RET_KEY1_FLAG DS CL2
RET_KEY1 DS   CL64
RET_KEY2_LENGTH DS CL2
RET_KEY2_FLAG DS CL2
RET_KEY2 DS   CL64
   ORG   RETURNED_KEY_BLOCK+138
*
PROCESSING_FLAGS DS CL1
SUBFUNCTION_CODE DS CL2
PCIXCC_SERIAL_NUMBER DS CL8
*
VUD_LIST   DS  6F
   ORG   VUD_LIST
VUD1_LENGTH DS F
VUD1_FLAG  DS XL2
VUD1_NO_FLAG DS XL2
VUD1_STRING DS A
VUD2_LENGTH DS F
VUD2_FLAG  DS XL2
VUD2_NO_FLAG DS XL2
VUD2_STRING DS A
*
KEY_LIST    DS  6F
   ORG   KEY_LIST
KEY1_LENGTH DS F
KEY1_FLAG  DS CL2
KEY1_STRING DS A
KEY2_LENGTH DS F
KEY2_FLAG  DS CL2
KEY2_STRING DS A
*
   ORG   *(DATD+1-(*-DATD))/(*-DATD)
ENDDATD    DS 0X
DYNSIZE EQU   ((ENDDATD-DATD+7)/8)*8
* THE FOLLOWING INSTRUCTION WILL CAUSE AN ASSEMBLY ERROR IF THE
* SIZE OF THE AUTOMATIC STORAGE AREA IS GREATER THAN 12288 Bytes.
* (12288 bytes is the maximum that can be obtained/released by
* CSFAGET/CSFAFREE.)
   ORG   DATD+(12288-(*-DATD))
   ORG   ENDDATD
*
* **********************************************************************
*            Mapping macros                                            *
* **********************************************************************
* 
  EJECT
  CSFCCVT
  EJECT
  CSFCCVE
  EJECT
  CSFASPB
  EJECT
  CSFGSVT
  EJECT
  END ZUDXSVC
Appendix B.

UDX Sample Code - Host Piece - Service Stub

This appendix contains a listing of the sample file zudxstub.bal. This function is a skeleton for the design of the host piece of a CCA extension.

*** START OF SPECIFICATIONS ******************************************************

* MODULE NAME = UDXSTUB
* DESCRIPTIVE NAME = UDX Service Stub Sample
* FUNCTION =
* THIS IS A SAMPLE SERVICE STUB. IT IS MEANT TO BE LINKEDITED
* WITH THE APPLICATION AND ENTERED VIA A CALL (CALL UDXSTUB). THIS
* STUB CAUSES THE EXECUTION OF THE SERVICE WITH SERVICE NUMBER = 49
* (DECIMAL).
* THIS IS A SAMPLE SERVICE STUB. IT IS MEANT TO BE LINKEDITED
* WITH THE APPLICATION AND ENTERED VIA A CALL (CALL UDXSTUB). THIS
* STUB CAUSES THE EXECUTION OF THE SERVICE WITH SERVICE NUMBER = 49
* (DECIMAL).
* TO MODIFY THE STUB FOR USE:
* 1. CHANGE THE VALUE OF 'UDXNUM' FROM 49 TO THE DECIMAL NUMBER OF
   YOUR SERVICE.
* 2. ISSUE A GLOBAL CHANGE TO CHANGE THE STUB NAME OF 'UDXSTUB' TO
   THE NAME OF YOUR SERVICE STUB.
* **** END OF SPECIFICATIONS ******************************************************

UDXSTUB START 0
UDXNUM EQU 49
UDXSTUB CSECT
MAINENT DS 0H
USING *,R15
LAE R15,0(R15,0)
L R15,-A(CICSTEST)
BAKR 0,R15
LTR R15,R15
BC 2,NOCICS

* YESCICS DS 0H
SAC 0
STM R14,R12,12(R13)
LR R12,R15
DROP R15
USING MAINENT,R12
LR R3,R0
B NORMAL
*

NOCICS DS 0H
USING MAINENT,R12
BSM R14,0
BAKR R14,0
LAE R12,0
LR R12,R15
SLR R13,R13

** At this point, R0 must contain the service number.
* If we are to call the TRUE, R13 is non-zero
* R1 points to the caller's parameter list.

UDX Sample Code - Host Piece - Service Stub  page B-1
UDX Sample Code - Host Piece - Service Stub

***************************************************************
NORMAL DS 0H
LA R0,UDXSNUM       R0 gets service number
SLR R10_ZERO,R10_ZERO
LR RC,R10_ZERO
L R2,CVTPTR
USING CVT,R2
L R2,CVTABEND
CLR R2,R10_ZERO
BC 8,NOICSF
USING SCVTSECT,R2
L R2,SCVTCCVT
CLR R2,R10_ZERO
BC 8,NOICSF
USING CCVT,R2
TM CCVTSFG1,'B'00110000' IS ICSF ACTIVE
BC 1,YESICSF
BC 1,YESICSF
LA RC,12          Set return code to 12 decimal
L R7,RETURN_CODE_PTR(,R1)
ST RC,RETURN_CODE(,R7)
SLR R0,R0
L R7,REASON_CODE_PTR(,R1)
ST R0,REASON_CODE(,R7)
B FINISHED
YESICSF DS 0H
***************************************************************
* Note that, if we're in CICS, the prolog code pointed R3 at the AFCB
* and R13 at the caller's savearea—they're still pointing. Also, R0
* contains the service number, with the high order bit ON if the TRUE
* has been tried and found wanting. In this last case, CSFAPRPD will
* check the high order bit and not attempt to call the TRUE.
* If R13 is zero, we're using the linkage stack. That means we can
* call CSFAPRPC.
* If R13 is not zero, we're using non-stack linkage. That means the
* caller's savearea will be used. CSFAPRPD uses this kind of linkage.
* But note that CSFAPRPD won't return here. Instead, it will return
* directly to the caller—that is, to the owner of the only save
* area around.
***************************************************************
CLR R13,R10_ZERO
BC 8,EXECPRPC
L R15,CVTFRPDP
BALR R14,R15
LR RC,R15
B FINISHED
EXECPRPC L R15,CVTFRPDP
BALR R14,R15
LR RC,R15
FINISHED DS 0H

***************************************************************
* This routine uses the linkage stack to save the caller's regs
* if this is not a CICS environment. In CICS, it uses the save
* area pointed to by register 13. So the epilog code takes one
* of two forms. If this is CICS (i.e. if R13 is non-zero),
* return is via LM and BR 14. If this is not CICS, return is
* via PR.
* *
* On return, the PR of ESA linkage does not restore registers
* 0, 1, 14 and 15. In the LM of normal BR 14 linkage, however,
* everything but 13 gets restored. Since this routine has no
* autodata, there's no way to pass back return and reason codes
* unless we leave 0 and 15 intact. The solution is to deviate
* slightly from normal BR 14 linkage and restore only registers
* 1 through 12 and 14.
***************************************************************
**********************************************************************  
** CICSTEST: Decides whether this is a CICS environment **
**********************************************************************
**********************************************************************
**********************************************************************
CICSTEST DS 0H
CICSTEST:  Decides whether this is a CICS environment
**********************************************************************
**********************************************************************
**********************************************************************
**********************************************************************
ESTA  R6,R11  00610015
LR    R7,R6                Copy of state & key in R7  00620015
N     R7,M_KEY             Q.  problem key?  00630015
BZ    RETRN                no---return  00640015
*    yes--check state  00650015
N     R6,M_STATE           Q.  problem state?  00660015
BZ    RETRN                no---return  00670015
*    yes--get the CICS eye-catcher  00680015
LA    R6,2                 Set ARs 6 and 8 to home  00690015
SAR   R6,R6  00700015
SAR   R8,R6  00710015
L     R8,TCBEXT2           R8->TCB extension  00720015
USING  TCBXNT2,R8  00730015
ICM   R4,B'1111',TCBCAUF   R4 gets AFCX address  00740015
*    Q.  Address there?  00750015
BZ    RETRN                no---return  00760015
*    yes--check eye-catch  00770015
CLC   0(4,R4),CICS_EYE     Q.  CICS?  00780015
BNE   RETRN                no---return  00790015
*    yes--pass info along  00800015
LR    R0,R4                R0 gets the AFCX pointer  00810015
O     R15,M_CICS           Enable high order bit of R15  00820015
RETRN   DS    0H  00830015
DROP  R12                    Free R12  00840015
PR                         Return from CICSTEST subroutine  00850015
*    00860015
LTORG  00870015
DS    0D  00880015
*    00890015
UDXDATA  DS    0F  00900015
R10_ZERO EQU 10  00910015
RC     EQU   05  00920015
R0     EQU   0  00930015
R1     EQU   1  00940015
R2     EQU   2  00950015
R3     EQU   3  00960015
R4     EQU   4  00970015
R5     EQU   5  00980015
R6     EQU   6  00990015
R7     EQU   7  01000015
R8     EQU   8  01010015
R9     EQU   9  01020015
R10    EQU  10  01030015
R11    EQU  11  01040015
R12    EQU  12  01050015
R13    EQU  13  01060015
R14    EQU  14  01070015
R15    EQU  15  01080015
*    01090015
INPUT_PARM EQU 0,8,C'C'  01100063
RETURN_CODE_PTR EQU INPUT_PARM,4,C'A'  01110015
REASON_CODE_PTR EQU INPUT_PARM+4,4,C'A'  01120015
RETURN_CODE EQU 0,4,C'F'  01130063
REASON_CODE EQU 0,4,C'F'  01140063
*    01141063
SAVAREA EQU 0,72,C'C'  01150015
SAVE14 EQU SAVAREA+12,4,C'A'  01160015
SAVE01 EQU SAVAREA+24,4,C'A'  01170015
SCVTSPTR EQU CVTABEND,4,C'F'  01190015
TCBPTR EQU PSATOLD,4,C'F'  01200015
DS    0D  01210015
*    01220015
M_KEY DC X'00800000'  Problem key mask  01240015
M_STATE DC X'00010000'  Problem state mask  01250015
M_NOCICS DC X'7FFFFFFF' Not-CICS mask 01260015
M_CICS DC X'80000000' Yes-CICS mask 01270015
DS 0D 01280015
CICS_EYE DC CL4'AFCX' CICS eye catcher 01290015
* 01300015
IHAPSA 01310015
TITLE 'DSECT CVT' 01320015
CVT DSECT=YES 01330015
TITLE 'DSECT SCVT' 01340015
IHASCVT DSECT=YES 01350015
TITLE 'DSECT TCB' 01360015
IKJTCB 01370015
TITLE 'DSECT CCVT' 01380015
CSFCCVT 01390015
* 01400015
END 01410015
Appendix C.

UDX Sample Code - Host Piece - CSFPCI Post-Processing Exit

This appendix contains a listing of the sample file `zudxexit.bal`. This file is a skeleton for the design of the host piece of a CCA extension.

**** START OF SPECIFICATIONS ****************************
* MODULE NAME = ZUDXEXIT
* DESCRIPTIVE NAME = Sample Post-Processing exit for CSFPCI
* *
* **** END OF SPECIFICATIONS ****************************

```
ZUDXEXIT CSECT,
ZUDXEXIT AMODE 31
ZUDXEXIT RMODE ANY
MAINENT DS 0H
USING *,R15
B     PROLOG
DC    AL1(42)
DC    C'ZUDXEXIT - UDX CSFPCI POST-PROCESSING EXIT'
DROP  R15
PROLOG   BSM   R14,0
BAKR  R14,0
LAE   R12,0
LR    R12,R15
PSTART   EQU   ZUDXEXIT
USING PSTART,R12
L     R00,DYNDATA_SIZE
LA    R15,0
CPYA  AR01,AR12
SAC   512
CSFAGET  OBTAIN,LENGTH=(0),SP=(15),LINKAGE=SYSTEM
LAE   R11,0(R01)
USING DATAD,R11

*-------------------------------------------------------------------
* SAVE THE ADDRESS OF THE XPB
*-------------------------------------------------------------------
EREG  R00,R01
LR    R03_EXPBPTR,R00
USING XPB,R03
CPYA  R03_EXPBPTR,AR12

*-------------------------------------------------------------------
* If the Rule Array indicates to Query Access Control Points, then
```
* copy the UDX access control points to the Reply Parameter Data Block

*--------------------------------------------------------------------

POSTPROC DS 0H
LA R06,1
SAR AR06,R06
LR R06,R01 GET PARAMETER LIST ADDR
L R06,RA_PTR(R06) GET RULE ARRAY ADDR
MVC LOC_RA(R06),RULE_ARRAY(R06) COPY RULE ARRAY
OC LOC_RA(R06),RA_BLANK FOLD IT TO UPPER CASE
CLC LOC_RA(R06),ACPOINTS IS IT "ACPOINTS" CALL ?
BNE NOT_ACP CALL WAS NOT FOR ACPOINTS

*--------------------------------------------------------------------

GET_ACP DS 0H START OF COPY ACPOINTS CODE
LA R14,1
SAR AR14,R14
LR R14,R01 GET PARAMETER LIST ADDR
L R14,REPLY_PARM_DB_LEN_PTR(R14)
L R08,REPLY_PARM_DATA_BLOCK_LEN(R14)
LR R09,R08 SAVE CURRENT DATA BLOCK LEN
LA R00,ACPTLEN GET ACPT LENGTH
ALR R08,R00 FIND TOTAL ACPT LENGTH
CL R08,XPBWORD IS THERE ENOUGH ROOM ?
BNH BUFLENOK YES, BUFFER LENGTH IS OK

*--------------------------------------------------------------------

BUFLENER DS 0H NO, BUFFER LENGTH ERROR
LA R04_LOCAL_RC,8 APPLICATION ERROR
LA R05_LOCAL_RS,605 BUFFER LENGTH ERROR
OI XPB_USERCRS,B'00100100'
B FINISHED

*--------------------------------------------------------------------

BUFLENOK DS 0H YES, BUFFER LENGTH IS OK
LA R04,0 GET ACPT LENGTH
LA R05,ACPTLEN GET ACPT LENGTH
MVC R06,R04
ST R08,REPLY_PARM_DATA_BLOCK_LEN,(R14)

NOT_ACP DS 0H
RCISZERO DS 0H
LA R04_LOCAL_RC,0
LA R05_LOCAL_RS,0
B FINISHED

FINISHED DS 0H
LA R00,DYNDATA_SIZE
LA R15,0
LR R01,R11
CSFAFREE RELEASE,LENGTH=(0),ADDR=(1),SP=(15),LINKAGE=SYSTEM
EREG R01,R01
LR R00,R05_LOCAL_RS
LR R15,R04_LOCAL_RC
PR

********************************************************************

* Storage Declares
********************************************************************

DYNDATA_SIZE DS 0A

UDX Sample Code - Host Piece - CSFPCI Post-Processing Exit  page C-2
DC    A(DYNSIZE)                                           00920020
DS    0D                                                   00930020
*-------------------------------------------------------------------    00940020
* Register Equates                                                      00950020
*-------------------------------------------------------------------    00960020
R00      EQU   0                                                        00970020
R01      EQU   1                                                        00980020
R02      EQU   2                                                        00990020
R03      EQU   3                                                        01000020
R04      EQU   4                                                        01010020
R05      EQU   5                                                        01020020
R06      EQU   6                                                        01030020
R07      EQU   7                                                        01040020
R08      EQU   8                                                        01050020
R09      EQU   9                                                        01060020
R10      EQU  10                                                       01070020
R11      EQU  11                                                       01080020
R12      EQU  12                                                       01090020
R13      EQU  13                                                       01100020
R14      EQU  14                                                       01110020
R15      EQU  15                                                       01120020
*-------------------------------------------------------------------    01130029
* Access Register Equates                                               01140029
*-------------------------------------------------------------------    01150029
AR00     EQU   0                                                        01160029
AR01     EQU   1                                                        01170029
AR02     EQU   2                                                        01180029
AR03     EQU   3                                                        01190029
AR04     EQU   4                                                        01200029
AR05     EQU   5                                                        01210029
AR06     EQU   6                                                        01220029
AR07     EQU   7                                                        01230029
AR08     EQU   8                                                        01240029
AR09     EQU   9                                                        01250029
AR10     EQU  10                                                       01260029
AR11     EQU  11                                                       01270029
AR12     EQU  12                                                       01280029
AR13     EQU  13                                                       01290029
AR14     EQU  14                                                       01300029
AR15     EQU  15                                                       01310029
*-------------------------------------------------------------------    01311029
* Parameter block Equates                                               01312029
*-------------------------------------------------------------------    01313029
PARMBLOCK             EQU 0,64,C'C'                                     01314030
RA_COUNT_PTR          EQU PARMBLOCK+16,4,C'A'                           01315030
RA_PTR                EQU PARMBLOCK+20,4,C'A'                           01316030
REPLY_PARM_DB_LEN_PTR EQU PARMBLOCK+56,4,C'A'                           01316130
REPLY_PARM_DB_PTR     EQU PARMBLOCK+60,4,C'A'                           01316230
*                                                                     01317029
RULE_ARRAY_COUNT          EQU 0,4,C'F'                                  01318030
RULE_ARRAY              EQU 0,8,C'C'                                   01319030
REPLY_PARM_DATA_BLOCK_LEN EQU 0,4,C'F'                                 01319429
REPLY_PARM_DATA_BLOCK   EQU 0,,C'C'                                   01319530
END_REPLY_PARM_DATA_BLOCK EQU 0,,C'C'                                 01319629
*                                                                     01320029
* Variable Equates                                                      01330029
*-------------------------------------------------------------------    01340029
R03_EXPBPTR  EQU R03                                                    01350029
R04_LOCAL_RC  EQU R04                                                    01360029
R05_LOCAL_RS  EQU R05                                                    01370029
*                                                                     01380029
* Variable Data Areas                                                   01390029
*-------------------------------------------------------------------    01400029
LTORG                                                          01401028
*-------------------------------------------------------------------    01410020
* Constants

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACPOINTS</td>
<td>DC CL8'ACPOINTS'</td>
<td>01420020</td>
</tr>
<tr>
<td>RA_BLANK</td>
<td>DC CL8''</td>
<td>01440020</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01450020</td>
</tr>
</tbody>
</table>

* START OF THE UDX ACCESS CONTROL POINT TABLE STRUCTURE (UACPTS)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>01450020</td>
</tr>
</tbody>
</table>

* The UACPTS has Group Information followed by one or more entries

* of Access Control Point Information. The Group Information should

* NOT be changed. The Access Control Point Information (ACPT) can be

* copied and most of it modified to describe your UDX Access Control

* Point.

* Descriptions of the 2 types of entries are as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACP Group Information</td>
<td>(This information must not be changed) :</td>
<td>01474028</td>
</tr>
<tr>
<td>TYPE</td>
<td>- Type of table entry, 01 is grouping information entry</td>
<td>01477928</td>
</tr>
<tr>
<td>TXT_LEN</td>
<td>- Length of TXT field</td>
<td>01477928</td>
</tr>
<tr>
<td>TXT</td>
<td>- ASCII description of the Group</td>
<td>01477928</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACPT Information:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TYPE</td>
<td>- Type of table entry, 02 is an access control point entry. THIS VALUE MUST ALWAYS BE X'02'.</td>
<td>01479028</td>
</tr>
<tr>
<td>CODE</td>
<td>- The hexadecimal value of the Access Control Point</td>
<td>01479028</td>
</tr>
<tr>
<td>TXT_LEN</td>
<td>- Length of TXT field</td>
<td>01479028</td>
</tr>
<tr>
<td>TXT</td>
<td>- ASCII description of this Access Control Point</td>
<td>01479028</td>
</tr>
<tr>
<td>FLAG</td>
<td>- THIS VALUE MUST ALWAYS BE X'00000000'.</td>
<td>01479028</td>
</tr>
<tr>
<td>ACPTS_CNT</td>
<td>- The number of other ACPTs listed in ACPTS</td>
<td>01479028</td>
</tr>
<tr>
<td>ACPTS</td>
<td>- The hexadecimal value of the other ACPTs that must be enabled if this ACPT is enabled.</td>
<td>01479028</td>
</tr>
</tbody>
</table>

* To add a UDX ACPT:

* 1. Increase the size of the 'UACPTS' to include the new ACPT Information. This increase will vary based on the length of the descriptive text and the number of ACPTs needing to be enabled for this ACPT.

* 2. Copy the lines of code between the comments:

*   *** COPY STARTING HERE

*   *** ENDING HERE

*   And put them before the comment:

*   *** PLACE BEFORE THIS LINE

* 3. Change:

*     a. The names of the lines, for example A01,... would become A03,... for the third Access Control Point.

*     b. The length of the ACP section, 'A03      DS   CL??'. where ?? is the actual length of the ACPT information being added. This value may be different for each ACPT depending on the length of the text and the number of other ACPTs needing to be enabled.

*     c. '_CODE' to the 2 byte hexadecimal ACPT value assigned to the UDX.

*     d. '_TXT_LEN' to the length of the descriptive text.

*     e. '_TXT' to the ASCII representation of the descriptive text for this UDX.

*     f. '_ACPTS_CNT' to the number of ACPTS to be listed in the following field.

*     g. '_ACPTS' all the 2 byte hexadecimal value of the ACPTs which need to be active for this UDX to function.

* 4. Add the OFFSET of this UDX ACPT Information Block with the length of this UDX ACPT Information Block and change the offset to the next UDX ACPT Information Block. NOTE: This new offset should be the same as the length of the 'UACPTS'
* from step 1.  

* ACPTSTRT DS 0X  
UACPTS DS CL77 WILL CHANGE TO NEW LENGTH  
ORG UACPTS  

* ACP GROUP INFORMATION - SHOULD NEVER BE CHANGED  

GRP DS CL9 LENGTH OF GROUP INFORMATION  
ORG GRP  

GRP_TYPE DC X'01' GROUP TYPE OF 01  
GRP_TXT_LEN DC X'00000004' TEXT LENGTH  
GRP_TXT DC X'55445873' TEXT IS ASCII 'UDXs'  

* Access Control Point (ACPT) Information block for the 1st ACPT  

ORG UACPTS+9 A01 STARTING OFFSET  
A01 DS CL45 A01 LENGTH  
ORG A01 A01 BREAKDOWN  
A01_TYPE DC X'02' A01 TYPE -->MUST REMAIN X'02'  
A01_CODE DC X'8000' A01 ACCESS CONTROL POINT  
A01_TXT_LEN DC X'00000006' A01 TEXT LENGTH  
A01_TXT DC X'4F94E54' A01 ASCII DESCRIPTION OF ACP  
A01_FLAG DC X'00000000' A01 FLAGS  
A01_ACPTS_CNT DC X'00000001' A01 ACPS ENABLE COUNT - ONE  
A01_ACPTS DS 0F A01 ACPS NEEDING ENABLED - ONE  

* Access Control Point (ACPT) Information block for the NEXT ACPT  

ORG UACPTS+77 NEXT UDX ACPT OFFSET  

*** COPY STARTING HERE  

A02 DS CL23 A02 LENGTH  
ORG A02 A02 BREAKDOWN  
A02_TYPE DC X'02' A02 TYPE  
A02_CODE DC X'8FFF' A02 ACCESS CONTROL POINT  
A02_TXT_LEN DC X'00000006' A02 TEXT LENGTH  
A02_TXT DC X'4F94E54' A02 ASCII DESCRIPTION OF ACP  
A02_FLAG DC X'00000000' A02 FLAGS  
A02_ACPTS_CNT DC X'00000001' A02 ACPS ENABLE COUNT - ONE  
A02_ACPTS DS 0F A02 ACPS NEEDING ENABLED  

* Access Control Point (ACPT) Information block for the 2nd ACPT  

*** ENDING HERE  

*** PLACE BEFORE THIS LINE  

ACPTEND DS 0X  
ACPTLEN EQU (ACPTEND-ACPTSTRT)  

* Dynamic Data Definitions  

DATAD DSECT 0F  

********************************************************************  

UDX Sample Code - Host Piece - CSFPCI Post-Processing Exit page C-5
LOC_RA DS CL8

*   ORG *(+1-(*-DATAD)/(*-DATAD)

ENDDATAD DS 0X

*   ORG (*-DATAD+7)/8)*8

DYNSIZE EQU ((ENDDATAD-DATAD+7)/8)*8

* THE FOLLOWING INSTRUCTION WILL CAUSE AN ASSEMBLY ERROR IF THE
* SIZE OF THE AUTOMATIC STORAGE AREA IS GREATER THAN 4096 BYTES.

ORG DATAD+(2288-(*-DATAD))

ORG ENDDATAD

*   CSFASPB
CSFEXPB
CSFCCVT
CSFCCVE

*   END ZUDXEXIT
Appendix D.
UDX Sample Code - Coprocessor Piece

This appendix contains a listing of the sample file `zudx_samp.c`. Notice the conditional compilation variable TEST_LNX.

```c
/********************************************/
/*
/* Module Name: ZUDXSAMP.C
/*
/* Descriptive Name: User Defined Extension zudxPIN1
/*
/* (C) Copyright IBM Corporation 1999
/*
/* Version 001, Release 000, Level 000
/*
/* Author: Kenneth B Kerr
/*
/*
/* Function:
/*
/* This module is the command processor for the user defined
/* extension (verb) zudxPIN1.
/*
/* Module Type:
/*
/* Attributes: Serial usable
/*
/* Language: IBM Visual Age C++ Version 3.00
/*
/*
/* Entry Points:
/*
/* zudxPIN1
/*
/*
/* Input:
/*
/* paramIn Request CPRB
/*
/* RequestId Identifier number of the request
/*
/*
/* Output:
/*
/* paramOut Reply CPRB
/*
/*
/* Change history:
/*
/* Date Programmer Description
/* -------- ----------- -------------
/* 08/26/99 kbk Created
/*
/*
/*
/********************************************/
#include <stdlib.h>
#include <string.h>
#include "cmncryt2.h"                         /* Cryptographic T2 definitions. */
#include "cmnerrcd.h"                          /* Common error codes. */
#include "cam_xtrn.h"                          /* CCA managers */
#include "casfunct.h"                          /* Common command processor funct's */
#include "camacm.h"                            /* Needed for access check */
#include "cxt_cmds.h"                    /* UDX access control codes */
#include "camdmgr.h"                    /* Domain manager prototypes */
#include "cassub.h"                       /* Common subroutines */

 /*****************************************************************************************/
/* Constants */
 /*****************************************************************************************/
#define RA_COUNT_ZERO 2                /* Rule array length for zero keywords */
#define RA_COUNT_ONE 10                /* Rule array length for one keyword */
#define RA_COUNT_TWO 18                /* Rule array length for two keywords */
#define PIN_BLOCK_SIZE 8
#define EXTRA_DATA_SIZE 8
#define EXPECTED_VUD_LENGTH 18
#define SIZE_OF_DES_KEY 8
#define PINENCI                   0x21     /* PIN-Encrypting IN Key: IPINENC */
#define PINENCO                   0x24   /* PIN-Encrypting OUT Key: OPINENC */

typedef enum { PIN1_KEYWORD_1W, PIN1_KEYWORD_2W, PIN1_KEYWORD_3W } PIN1_RULE1;
typedef enum { PIN1_KEYWORD_1O, PIN1_KEYWORD_2O } PIN1_RULE2;

 /*****************************************************************************************/
** ENTER
**       your CCA command extension array entry after this comment.
**       =====================================================
**
** Each element of the table is a CCAX_CP_DEF type. That is, it
** contains one 2 character sub-function code, and a pointer to
** the corresponding command processor function.
**
 /*****************************************************************************************/
CCAX_CP_DEF ccax_cp_list[] = { { ZUDX_CODE, zudxPIN1 }};

 /*****************************************************************************************/
**
** Declare a variable which holds the number of CCA extension verbs
** defined in the ccax_cp_list table above.
**
 /*****************************************************************************************/
ULONG ccax_cp_list_size = ( sizeof(ccax_cp_list) / sizeof(CCAX_CP_DEF) );

void zudxPIN1(
    CPRB_structure *pCprbIn,         /* (input) request CPRB */
    CPRB_structure *pCprbOut,      /* (output) reply CPRB */
    unsigned long   RequestId,       /* (input) Adapter request identifier */
    role_id_t       roleID )                /* (input) role ID ptr */
{
    /*****************************************************************************************/
    /* Declarations */
    /*****************************************************************************************/
}
unsigned char *pExtraData; /* Pointer to extra data block */

/* Key block processing variables*/
key_data_structure *pThiskey; /* Key token from request block */
key_data_structure *pNextKey; /* Key token from request block */
generic_key_block_structure *pToken; /* Key in parameter block */
des_key_token_structure *pInputPinKeyToken; /* Input PIN key token */
des_key_token_structure *pOutputPinKeyToken; /* Output PIN key token */
KEY_FIELD_HEADER InputKeyHeader; /* header for key in reply block */
KEY_FIELD_HEADER OutputKeyHeader; /* header for key in reply block */

/* local variables*/
long ReturnMsg; /* Return code from function calls */
mk_status_var MstrKeyStatus; /* Master key status */
mk_selectors MKSelector; /* Master key selector */
boolean Authorized; /* Truth value that the caller is authorized to execute this command */
unsigned char OutputPinBlock[8]; /* Output PIN block */
int i; /* Iteration variable */
DES_TOKEN_CHECK ErrorMessage; /* error message from DES token check */
UCHAR MKVP[MKVP_LENGTH]; /* master key verification pattern */
ULONG TVV; /* calculated TVV for output key token */

/* Rule array processing variables*/
int RuleValue[2]; /* Output for rule_check */
USHORT RuleMapCount = 5; /* Number of entries in the rule map */
static RULE_MAP RuleMap[5] = { {"KEYW1", 1, PIN1_KEYWORD_1W},
{"KEYW2", 1, PIN1_KEYWORD_2W},
{"KEYW3", 1, PIN1_KEYWORD_3W},
{"KEYO1", 2, PIN1_KEYWORD_1O},
{"KEYO2", 2, PIN1_KEYWORD_2O} };
PIN1_RULE1 Rule1; /* Rule 1 specified */
PIN1_RULE2 Rule2; /* Rule 2 specified */

/***************************************************************************************/
/* Begin executable code. */
/***************************************************************************************/
if (RequestId == 0) /* Do nothing statement to get rid of compiler */
ReturnMsg = 0; /* warning messages because RequestId is not used. */

/***************************************************************************************/
/* Copy input CPRB to the output area. */
/***************************************************************************************/
memcpy(pCprbOut, pCprbIn, pCprbIn->CPRB_length);

/***************************************************************************************/
/* Initialize the CPRB request/reply parameter pointers and then */
/* set my local pointers to the request and reply parameter blocks. */
/***************************************************************************************/
InitCprbParmPointers( pCprbIn, pCprbOut );
pReqBlk = pCprbIn->req_parm_block;
pRepBlk = pCprbOut->reply_parm_block;

/***************************************************************************************/
/* Set the reply subfunction code early, because the Cas_proc_retc */
/* routine needs it set for negative return codes. */
/***************************************************************************************/
pRepBlk->subfunction_code = pReqBlk->subfunction_code;

/***************************************************************************************/
/* Check that the caller is authorized to use this domain. */
/* Set the domain in the master key selector. */
/***************************************************************************************/
if (!dmDomainCheck(pCprbIn))
{  
  Cas_proc_retc( pCprbOut, DOMAIN_MANAGER_ERROR );  
  return;  
}

/***************************************************************************************/
/* Initialize the master key selector. */
/***************************************************************************************/
MKSelector.mk_set = pCprbIn->Domain;  
MKSelector.type_mks = SYM_MK;  

/***************************************************************************************/
/* Make sure this service is authorized before we go any further */
/***************************************************************************************/
CHECK_ACCESS_AUTH( pCprbIn, 
  pCprbOut, 
  roleID, 
  UDX_COMMAND_PIN1, 
  &Authorized );
if ( !Authorized )
{
  Cas_proc_retc( pCprbOut, CP_NOT_AUTH );
  return;
}

/***************************************************************************************/
/* Make sure the current master key is valid before we go any further. */
/***************************************************************************************/
ReturnMsg = mkmGetMasterKeyStatus( MKSelector, &MstrKeyStatus );
switch ( ReturnMsg )
{
  case MK_NO_ERROR :  
    if (( MstrKeyStatus & mks_CMK_VALID ) != mks_CMK_VALID )
    {
      Cas_proc_retc(pCprbOut, MASTER_KEY_ERROR);
      return;
    }
    break;

  case MK_SRDI_OPEN_ERROR :
    Cas_proc_retc( pCprbOut, FT_MK_SRDI_OPENERR );
    return;
    break;

  default :  
    Cas_proc_retc(pCprbOut, MASTER_KEY_ERROR);
    return;
    break;
}

/***************************************************************************************/
/* Perform consistency check on the request parameter block */
/***************************************************************************************/
if ( parm_block_valid( pCprbIn, SEL_REQ_BLK ) == false )
{
  Cas_proc_retc( pCprbOut, RT_CONSISTENCY_ERROR );
  return;
}

/***************************************************************************************/
/* Perform consistency check on the rule array - for this verb, the */
/* rule array may have zero, one or two values. */
/***************************************************************************************/
switch( pReqBlk->rule_array_length )
{
    case RA_COUNT_ZERO: /* use default values*/
        Rule1 = PIN1_KEYWORD_1W; /* default*/
        Rule2 = PIN1_KEYWORD_2O; /* default*/
        break;
    case RA_COUNT_ONE :
    case RA_COUNT_TWO :
        RuleValue[ 0 ] = INVALID_RULE; /* rule_check requires this initialization.*/
        RuleValue[ 1 ] = INVALID_RULE;
        if ( rule_check ( (RULE_BLOCK *) &pReqBlk->rule_array_length,
            RuleMapCount,
            &RuleMap[0],
            (int *) &RuleValue,
            &ReturnMsg ) == false )
        {
            Cas_proc_retc ( pCprbOut, ReturnMsg );
            return ;
        }
    if ( RuleValue[ 0 ] == INVALID_RULE )
        Rule1 = PIN1_KEYWORD_1W; /* default*/
    else
        Rule1 = (PIN1_RULE1) RuleValue[ 0 ];
    if ( RuleValue[ 1 ] == INVALID_RULE )
        Rule2 = PIN1_KEYWORD_2O; /* default */
    else
        Rule2 = (PIN1_RULE2) RuleValue[ 1 ];
    break;
    default: /* count not valid */
        Cas_proc_retc( pCprbOut, E_RULE_ARRAY_CNT );
        return ;
}
**************************************************************************************
/* Perform consistency check on the verb unique data. Parse the PIN block */
/* and extra data block from the VUD. */
**************************************************************************************
pVUDBlock = (verb_unique_data_structure *)
    ((UCHAR *)&pReqBlk->rule_array_length +
    pReqBlk->rule_array_length);
if ( pVUDBlock->verb_unique_data_length != EXPECTED_VUD_LENGTH )
{
    Cas_proc_retc( pCprbOut, RT_CONSISTENCY_ERROR );
    return;
}
pPinBlock = (unsigned char *) &pVUDBlock->verb_unique_data;
pExtraData = (unsigned char *) pPinBlock + PIN_BLOCK_SIZE;
**************************************************************************************
/* Parse the PIN Encrypting Keys from the key block. */
**************************************************************************************
if ( ! find_first_key_block( pCprbIn, &pThiskey, SEL_REQ_BLK ))
{
    Cas_proc_retc( pCprbOut, RT_CONSISTENCY_ERROR );
    return;
}
pToken = (generic_key_block_structure *) pThiskey;
pInputPinKeyToken = ( des_key_token_structure * ) (UCHAR * ) &pToken->label_or_token);
/* Get the output PIN encrypting key token from the request block. */
if ( ! find_next_key_block( pCprbIn, pThiskey, &pNextkey, SEL_REQ_BLK ))
{
    Cas_proc_retc( pCprbOut, RT_CONSISTENCY_ERROR );
    return;
}
pToken = (generic_key_block_structure * ) pNextkey;
pOutputPinKeyToken = ( des_key_token_structure * ) ((UCHAR * ) &pToken->label_or_token);

/* Check the input PIN Encrypting Key. */
if ( ! cas_des_key_token_check( pInputPinKeyToken, &ErrorMessage ))
switch( ErrorMessage )
{
case DES_TOKEN_CHECK_VERSION :
    Cas_proc_retc( pCprbOut, E_INV_TKNVER );
    return;
    break;
case DES_TOKEN_CHECK_TOKENFLAG :
    Cas_proc_retc( pCprbOut, E_KEK_ID_FORM );
    return;
    break;
default :
    Cas_proc_retc( pCprbOut, RT_TKN_UNUSEABLE );
    return;
} /* select on error message */
if ( cas_key_tokentvv_check( pInputPinKeyToken ) == false )
{
    Cas_proc_retc( pCprbOut, E_INTRN_TOKEN_TVV);
    return;
}

/* Check the output PIN Encrypting Key if it's not a null token. */
if ( pOutputPinKeyToken->tokenFlag != EMPTY_TOKEN_FLAG )
{
    if ( ! cas_des_key_token_check( pOutputPinKeyToken, &ErrorMessage ))
switch( ErrorMessage )
{
case DES_TOKEN_CHECK_VERSION :
    Cas_proc_retc( pCprbOut, E_INV_TKNVER );
    return;
    break;
case DES_TOKEN_CHECK_TOKENFLAG :
    Cas_proc_retc( pCprbOut, E_KEK_ID_FORM );
    return;
    break;
default :
    Cas_proc_retc( pCprbOut, RT_TKN_UNUSEABLE );
    return;
} /* select on error message */
if ( cas_key_tokentvv_check( pOutputPinKeyToken ) == false )
{
    Cas_proc_retc( pCprbOut, E_INTRN_TOKEN_TVV);
    return;
}
}
FLICT */
/* Control vector checking. */
/* Perform any necessary checking of the control vector. */
/* Add other checks as appropriate. */
/* Check that the input PIN Key is of the input PIN encrypting class*/
if ( pInputPinKeyToken->cvBase[1] != PINENCI )
{
    Cas_proc_retc( pCprbOut, RT_CV_CONFLICT );
    return;
}
/* If Rule2 is PIN1_KEYWORD_20, check the output PIN Key */
if ( Rule2 == PIN1_KEYWORD_20 )
{
    if ( pOutputPinKeyToken->cvBase[1] != PINENCO )
    {
        Cas_proc_retc( pCprbOut, RT_CV_CONFLICT );
        return;
    }
}
/* Determine which master key to use to decipher the input PIN key. */
switch( cas_master_key_check( pInputPinKeyToken ))
{
    case OLD :
        /* The key token's MKVP matches the old master key's MKVP. Generate a */
        /* warning reason code that a key is encrypted under the old master key. */
        MKSelector.mk_register = old_mk;
        Cas_proc_retc( pCprbOut, RT_OMK_TOKEN_USED );
        break;
    case CURRENT :
        /* The key token's MKVP matches the current master key's MKVP. */
        MKSelector.mk_register = current_mk;
        break;
    case OUT_OF_DATE :
    default :
        /* The key token's MKVP doesn't match current or old master key's */
        /* MKVP. We don't know what to do with the key token. */
        Cas_proc_retc( pCprbOut, RT_KEY_INV_MKVN );
        return;
        break;
}
Decipher the input PIN key under the corresponding master key.

ReturnMsg = triple_decrypt_under_master_key_with_CV( &MKSelector,
&(pInputPinKeyToken->cvBase),
&(pInputPinKeyToken->keyLeft[0]),
&(pInputPinKeyToken->keyLeft[0]) );

if ( ReturnMsg != mk_NO_ERROR )
{
    Cas_proc_retc( pCprbOut, ReturnMsg );
    return;
}

ReturnMsg = triple_decrypt_under_master_key_with_CV( &MKSelector,
&(pInputPinKeyToken->cvExten),
&(pInputPinKeyToken->keyRight[0]),
&(pInputPinKeyToken->keyRight[0]) );

if ( ReturnMsg != mk_NO_ERROR )
{
    Cas_proc_retc( pCprbOut, ReturnMsg );
    return;
}

Generate warning return code if the PIN key does not have odd parity.

for ( i = 0; i < SIZE_OF_DES_KEY; i++ )
{
    if ( (cas_parity_odd( pInputPinKeyToken->keyLeft[i] ) == FALSE) ||
        (cas_parity_odd( pInputPinKeyToken->keyRight[i] ) == FALSE) )
    {
        Cas_proc_retc( pCprbOut, CP_KDATA_NOTODD );
        break;
    }
}

If the output PIN key is used, determine which master key to use to

decipher the output PIN key.

if ( Rule2 == PIN1_KEYWORD_2O )
{
    switch( cas_master_key_check( pOutputPinKeyToken ) )
    {
        case OLD :
            MKSelector.mk_register = old_mk;
            break;

        case CURRENT :
            MKSelector.mk_register = current_mk;
            break;
    }
}
case OUT_OF_DATE :
default :
	 /**************************************************************************/
	 /* The key token's MKVP doesn't match current or old master key's */
	 /* MKVP. We don't know what to do with the key token. */
	 /**************************************************************************/
	 Cas_proc_retc( pCprbOut, RT_KEY_INV_MKVN );
	 return;
	 break;

	 /**************************************************************************/
	 /* Decipher the input PIN key under the corresponding master key. */
	 /**************************************************************************/
	 ReturnMsg = triple_decrypt_under_master_key_with_CV( &MKSelector,
	 &pOutputPinKeyToken->cvBase,
	 &pOutputPinKeyToken->keyLeft[0]),
	 &pOutputPinKeyToken->keyLeft[0] );
	 if ( ReturnMsg != mk_NO_ERROR )
	 {
	  Cas_proc_retc( pCprbOut, ReturnMsg );
	  return;
	 }
	 ReturnMsg = triple_decrypt_under_master_key_with_CV( &MKSelector,
	 &pOutputPinKeyToken->cvExten),
	 &pOutputPinKeyToken->keyRight[0]),
	 &pOutputPinKeyToken->keyRight[0] );
	 if ( ReturnMsg != mk_NO_ERROR )
	 {
	  Cas_proc_retc( pCprbOut, ReturnMsg );
	  return;
	 }
	 /**************************************************************************/
	 /* Generate warning return code if the PIN key does not have odd parity. */
	 /**************************************************************************/
	 for ( i = 0; i < SIZE_OF_DES_KEY; i++ )
	 {
	  if ( (cas_parity_odd( pOutputPinKeyToken->keyLeft[i] ) == FALSE)  ||
	    (cas_parity_odd( pOutputPinKeyToken->keyRight[i]) == FALSE) )
	  {
	    Cas_proc_retc( pCprbOut, CP_KDATA_NOTODD );
	    break;
	  }
	 }
}
	 /**************************************************************************/
	 /* Rule2 is keyword 20 */
	 /**************************************************************************/
	 /* Processing required for this verb based on inputs. */
	 /**************************************************************************/
	 /**************************************************************************/
	 for ( i = 0; i < 8; i++ )
	 {
	  if( Rule1 == PIN1_KEYWORD_1W )
	    OutputPinBlock[ i ] = pExtraData[ i ] ^ pPinBlock[ i ];
	  else
	    OutputPinBlock[ i ] = pExtraData[ i ] & pPinBlock[ i ];
	 }
/* Reencipher the key token if the token was enciphered under the old */
/* master key. */
/***************************************************************************************/
if (pCprbOut->secy_return_code == RT_OMK_TOKEN_USED)
{
  if (cas_master_key_check(pInputPinKeyToken) == OLD)
  {
    MKSelector.mk_register = current_mk;
    /* Encipher the input PIN key under the new master key. */
    /*--------------------------------------------------------------------------------*/
    ReturnMsg = triple_encrypt_under_master_key_with_CV( &MKSelector,
      &(pInputPinKeyToken->cvBase),
      &(pInputPinKeyToken->keyLeft[0]),
      &(pInputPinKeyToken->keyLeft[0]) );
    if (ReturnMsg != mk_NO_ERROR)
    {
      Cas_proc_retc(pCprbOut, ReturnMsg);
      return;
    }
    /*--------------------------------------------------------------------------------*/
    /* Complete the target key token. */
    /*--------------------------------------------------------------------------------*/
    /* Get the MKVP of the current master key and put it in the token. */
    /*--------------------------------------------------------------------------------*/
    CasCurrentMkvp( &MKSelector, (UCHAR *) &MKVP );
    memcpy(pInputPinKeyToken->mkvp, &MKVP, sizeof(pInputPinKeyToken->mkvp));
    /*--------------------------------------------------------------------------------*/
    /* Calculate the TVV and copy it to the token. */
    /*--------------------------------------------------------------------------------*/
    pka96_tvvgen(DES_TOKEN_LENGTH, (UCHAR *) pInputPinKeyToken, &TVV);
    memcpy((UCHAR *) &(pInputPinKeyToken->tvv), (UCHAR *) &TVV, TVV_LENGTH);
  }
  /* input PIN key enciphered under the old master key */
  if (Rule2 == PIN1_KEYWORD_2O)
  {
    if (cas_master_key_check(pOutputPinKeyToken) == OLD)
    {
      MKSelector.mk_register = current_mk;
      /* Encipher the input PIN key under the new master key. */
      /*--------------------------------------------------------------------------------*/
      ReturnMsg = triple_encrypt_under_master_key_with_CV( &MKSelector,
        &(pOutputPinKeyToken->cvBase),
        &(pOutputPinKeyToken->keyLeft[0]),
        &(pOutputPinKeyToken->keyLeft[0]) );
    }
  }
}
if ( ReturnMsg != mk_NO_ERROR )
{
    Cas_proc_retc( pCprbOut, ReturnMsg );
    return;
}

ReturnMsg = triple_encrypt_under_master_key_with_CV( &MKSelector,
               &pOutputPinKeyToken->cvExten),
               &pOutputPinKeyToken->keyRight[0]),
               &pOutputPinKeyToken->keyRight[0]));

if ( ReturnMsg != mk_NO_ERROR )
{
    Cas_proc_retc( pCprbOut, ReturnMsg );
    return;
}

/****************************************************************************
/* Complete the target key token. */
/****************************************************************************
/* Get the MKVP of the current master key and put it in the token. */
/****************************************************************************
CasCurrentMkvp( &MKSelector, UCHAR *MKVP );
memcpy( pOutputPinKeyToken->mkvp,
        &MKVP,
        sizeof( pOutputPinKeyToken->mkvp ));

/****************************************************************************
/* Calculate the TVV and copy it to the token. */
/****************************************************************************
pka96_tvvgen( DES_TOKEN_LENGTH, UCHAR *pOutputPinKeyToken, &TVV );
memcpy( (UCHAR *) &TVV,
        pOutputPinKeyToken->tvv,
        TVV_LENGTH );

} /* output PIN key enciphered under the old master key */
}

/****************************************************************************
/* Build the reply CPRB. */
/****************************************************************************
/* */
/* */
/* Sub- | Rule | Rule | Verb | Verb | Key | Key Fields */
/* Function | Array | Array | Data | Unique | Block | */
/* Code | Block | Elements | Block | Data | | */
/* | Length | Length | | | | */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* */
/* For UDX zudxPIN1, the output PIN block and the key tokens are returned */
/* to the caller. */
/****************************************************************************
pCprbOut->reply_parm_block = pRepBlk;
pReplyBlockPtr = (UCHAR *) pRepBlk;
ReplyBlockLength = 4;

/****************************************************************************
/* Add the rule array which is empty. */
/****************************************************************************
pRepBlk->rule_array_length = NO_RULEARRAY;

/****************************************************************************
/* Add the output PIN block to the VUD. */
ReplyBlockLength += BuildParmBlock( pReplyBlockPtr + ReplyBlockLength, 1,
            (USHORT) PIN_BLOCK_SIZE, &OutputPinBlock );

/* Add the key block. */

InputKeyHeader.Length = KEY_HDR_LEN + DES_TOKEN_LENGTH;
InputKeyHeader.Flags = DES96_TYPE | ACTION_NOOP;
OutputKeyHeader.Length = KEY_HDR_LEN + DES_TOKEN_LENGTH;
OutputKeyHeader.Flags = DES96_TYPE | ACTION_NOOP;

ReplyBlockLength += BuildParmBlock( pReplyBlockPtr + ReplyBlockLength, 4,
            KEY_HDR_LEN, &InputKeyHeader,
            (USHORT) DES_TOKEN_LENGTH, pInputPinKeyToken,
            KEY_HDR_LEN, &OutputKeyHeader,
            (USHORT) DES_TOKEN_LENGTH, pOutputPinKeyToken );

/* Enough room in the CRPB? */

pCprbOut->replied_parm_block_length = ReplyBlockLength;

if ( pCprbOut->reply_msg_block_length < pCprbOut->replied_parm_block_length )
{
    Cas_proc_retc( pCprbOut, REPLY_TOO_LONG );
    return;
}

/* Return to the caller. */

Cas_proc_retc( pCprbOut, S_OK );
return;

} /* end-of zudxPIN1( ) */
Appendix E.

UDX Sample Code - Workstation Host - Test Code

This appendix contains a listing of the sample file sxt_samp.c. This file is a skeleton for the design of the host piece of a CCA extension.

/*----------------------------------------------------------------------------*/
/* Module Name: SXT_SAMP.C */
/*                                                                            */
/* Sample callable service for UDX - PIN Block Processing Service */
/*                                                                            */
/* (C) Copyright IBM Corporation, 2001 */
/*----------------------------------------------------------------------------*/
/* Function: This file contains the sample SAPI CCA API extension verb */
/* zPIN1. It illustrates interfacing with the application and */
/* shows how to send the request to the cryptographic adapter */
/* card. This program will process an encrypted PIN block (assume */
/* a proprietary block form) and return the block encrypted under */
/* the original or a second key. */
/*                                                                            */
/* Module Type: */
/* Attributes: Serial usable */
/* Language: IBM Visual Age C++ Version 3.00 */
/* Entry Points: */
/* zPIN1 */
/* Inputs: Rule Array Count Number of keywords passed in rule array */
/* Rule Array Keywords (0, 1 or 2 keywords may be passed. */
/* Input Pin Key Id Input PIN encrypting key identifier. The */
/* input PIN block is enciphered under this key. */
/* The identifier is a token. */
/* Input Pin Block Enciphered PIN block to be processed. */
/* Output Pin Key Id Output PIN encrypting key identifier or a */
/* null token. The identifier is a token. If */
/* the key is not used, a null token is supplied */
/* Extra Data Extra data to be used in processing the PIN */
/* block (always 8 bytes). */
/* Processing: 1. Build the request parameter block and request CRPB. */
/* 2. Submit the request. */
/* 3. Process the reply CPRB and parameter block. */
/* 4. Return to caller. */
/* Outputs: Return Code Return code from processing */
/* Reason Code Reason code from processing */
/* Output Pin Block Processed enciphered PIN block */
/* Input Pin Key Id Reenciphered token if the key was */
/* enciphered under the old master key. */
/* Output Pin Key Id Reenciphered token if the key was */
enciphered under the old master key.

Change History

Date   Programmer  Description
---------  ----------  -----------
03/14/01  kbk         Created

#define CSUC_32BIT_SOURCE
#include "csunincl.h" /* Callable services prototypes */

#define EXTRA_DATA_LENGTH 8
#define PIN_BLOCK_LENGTH 8
#define ZUDX_CODE 0x5852

void SECURITYAPI zPIN1(
    long *pReturnCode,
    long *pReasonCode,
    long *pRuleArrayCount,
    UCHAR *pRuleArray,
    UCHAR *pInputPINKeyId,
    UCHAR *pInputPINBlock,
    UCHAR *pExtraData,
    UCHAR *pOutputPINKeyId,
    UCHAR *pOutputPINBlock )
{
    CPRB_ptr pCprb; /* CPRB pointer */
    REQUEST_REPLY_BUF *pRequestReplyBuffer; /* buffer area pointer for request */
    UCHAR *pRequestParmBlock; /* request parm blk pointer */
    USHORT rgpb_len; /* request parm buffer length */
    KEY_FIELD_HEADER key_hdr_input; /* header for key parm */
    KEY_FIELD_HEADER key_hdr_output; /* header for key parm */
    long msg; /* message of SAPI routines */
    UCHAR *pReturnVUD; /* pointer to returned PIN block */

    /* Check if return code or reason code is NULL. */
    if (pReturnCode == NULL || pReasonCode == NULL)
        return; /* return right away */

    /* Check if pointers are NULL. */
if ( pInputPINBlock == NULL || pInputPINKeyId == NULL ||
    pExtraData == NULL || pOutputPINKeyId == NULL )
{
    CSUC_PROCRETC(pReturnCode, pReasonCode, E_NULL_PTR);
    return; /* return error if any of */
    /* the conditions are met */
}

/* Set return code and reason code to zero. */
*pReturnCode = 0;
*pReasonCode = 0;

/*-------------------------------------------------------------------------*/
/* Allocate space for a working area. */
/*-------------------------------------------------------------------------*/
pRequestReplyBuffer = malloc( sizeof( REQUEST_REPLY_BUF ));
if ( pRequestReplyBuffer == NULL )
{
    CSUC_PROCRETC( pReturnCode, pReasonCode, E_ALLOCATE_MEM );
    return;
}

pCprb = (CPRB_ptr) &(pRequestReplyBuffer->request_buf[ 0 ]);
pRequestParmBlock = &(pRequestReplyBuffer->request_buf[ 0 ])
    + sizeof( CPRB_structure );

/*-------------------------------------------------------------------------*/
/* Request parameter block */
/* */
/* */
/* +-----------------------------------/-------/----------/---+ */
/* | Sub- | Rule | Rule | Verb Unique | Key Block | */
/* | Function | Array | Array | Data Fields | Fields | */
/* | Code | Block | Elements |      | Data | Length | Fields | */
/* | | Length | Length | Data | Length | Fields | */
/* | | ( Length = X ) | ( Length = Y ) | ( Length = Z ) | */
/* | | | +----------------+----------------+----------------+ */
/* | 0 | 2 | 4 | 2+X | 4+X | 2+X+Y | 4+X+Y | */
/* +-----------------------------------/-------/----------/---+ */

/*-------------------------------------------------------------------------*/
/* Part 1 of 4. 2-byte subfunction code */
/*-------------------------------------------------------------------------*/
*((USHORT *)pRequestParmBlock) = htoa(ZUDX_CODE);
rgpb_len = 2;

/*-------------------------------------------------------------------------*/
/* Part 2 of 4. Rule array block */
/*-------------------------------------------------------------------------*/
if ( ( *pRuleArrayCount < RAC_MIN ) ||
    ( *pRuleArrayCount > RAC_MAX ) )
{
    free( pRequestReplyBuffer );
    CSUC_PROCRETC( pReturnCode, pReasonCode, E_RULE_ARRAY_CNT );
    return;
}
else
{
    rgpb_len += BuildParmBlock( pRequestParmBlock + rgpb_len, 1,
        (USHORT) ( 8 + *pRuleArrayCount ), pRuleArray );
}

/*-------------------------------------------------------------------------*/
/* Part 3 of 4. Verb-unique data block: Input PIN block and extra data */
住房公积
rqpb_len += BuildParmBlock( pRequestParmBlock + rqpb_len, 2,
    PIN_BLOCK_LENGTH, pInputPINBlock,
    EXTRA_DATA_LENGTH, pExtraData );

/* Part 4 of 4. Key block */
/* The keys are in this order: */
/* Input PIN Key Id (token) */
/* Output PIN Key Id (token) */

/* Input PIN Key Id */
key_hdr_input.Length = htoas(KEY_HDR_LEN + DES_TOKEN_LENGTH);
key_hdr_input.Flags = htoas(DES96_TYPE | ACTION_NOOP);

/* Output PIN Key Id */
key_hdr_output.Length = htoas(KEY_HDR_LEN + DES_TOKEN_LENGTH);
key_hdr_output.Flags = htoas(DES96_TYPE | ACTION_NOOP);

/* Build the key block. */

rqpb_len += BuildParmBlock( pRequestParmBlock + rqpb_len, 4,
    KEY_HDR_LEN, &key_hdr_input,
    (USHORT)DES_TOKEN_LENGTH, pInputPINKeyId,
    KEY_HDR_LEN, &key_hdr_output,
    (USHORT)DES_TOKEN_LENGTH, pOutputPINKeyId );

/* Call Security Server using function CSNC_SP_SCSRFBSS. */
CSNC_SP_SCSRFBSS((CPRB_ptr) pCprb, (long *) &msg);

/* Note: CSUC_PROCRETC returns ERROR if the error code in msg is higher */
/* than the error code already in *pReturn_code and *pReason_code. */
/* msg is the return code and reason code, concatenated in a single long */
/* integer - for example, msg=00080012 is equivalent to return code 8, */
/* reason code 12. */

if ( (msg != S_OK) && (CSUC_PROCRETC(pReturnCode, pReasonCode, msg) == ERROR))
{
    free( pRequestReplyBuffer );
    return;
}

/* Process the returned data, which is in the Reply Parameter Block. */
/* Examine the Reply Parameter Block to make sure it is OK. If not, */
/* something is wrong in the adapter - it should return valid data. */
if ( ! parm_block_valid( (CPRB_structure *) pCprb, SEL_REPLY_BLK ))
    CSUC_PROCRETC(pReturnCode, pReasonCode, CP_DEV_HWERR);
else
{
    /* The output PIN block is returned in the VUD block. Since the rule */
    /* array block is empty, the VUD block is two bytes after the        */
    /* rule array block. This corresponds to the first rule array element. */
    pReturnVUD = &pCprb->reply_parm_block->first_rule_array_element;
    memcpy( pOutputPINBlock, pReturnVUD, PIN_BLOCK_LENGTH );
}

free( pRequestReplyBuffer );
return;

} /* end of zPIN1 */
Appendix F.

Reserved Values

Certain values have been reserved for the use of UDX developers. IBM will not use these values in future upgrades to the CCA, so there will be no overlap between a UDX using these values and an upgrade of the toolkit. The specific values are as follows:

For completion codes: The values between 0x5000 and 0x5FFF have been reserved for UDX writers.

For access control points: For zSeries, the values between 0x8000 and 0xEFFF may be used. (Values between 0xF000 and 0xFFFF will be used on zSeries by IBM UDX writers.)

For subfunction codes: The values between “XA” (0x5841) to “XZ” (0x585A), “YA” (0x5941) to “YZ” (0x595A), “X0” (0x5830) to “X9” (0x5839), and “Y0” (0x5930), to “Y9” (0x5939) are reserved for UDX writers.

For DES control vectors: Bits 4, 5, and 61 will not affect or be affected by the import or export of a key. Bits 4 and 5 will be ignored by the CCA at all times. Bit 61 will prevent a key from use in any standard CCA verb, thus reserving a key for use only in a UDX function.
Appendix G. Data Structures

This appendix identifies useful data structures from the toolkit header files.

Structures Used in Communications Between the Workstation Host and the Coprocessor

These structures may be used on the coprocessor or on the workstation host machine. If you are writing code for the zSeries 4764, the host side instructions in this section will be useful only for building the workstation test library.

A REQUEST_REPLY_BUF structure should be declared in the host function to allocate the data storage for the CPRB Structures and the request and reply buffers. This structure has two fields, both of 5120 bytes (BLK_LEN_MAX).

REQUEST_REPLY_BUF

<table>
<thead>
<tr>
<th>Size of field</th>
<th>Field name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>5120 bytes</td>
<td>request_buf</td>
<td>Holds the CPRB structure and the request block.</td>
</tr>
<tr>
<td>5120 bytes</td>
<td>reply_buf</td>
<td>Holds the (return) CPRB structure and the reply block.</td>
</tr>
</tbody>
</table>

The REQUEST_REPLY_BUF structure is filled with the following structures (hence they are declared as pointers into the REQUEST_REPLY_BUF structure).

First, a CPRB structure:

Note that the (request) CPRB structure is filled (as completely as it needs to be) by calling CSUC_BULDCPRB() with the appropriate lengths and pointers from within the host function. Fields not filled by this function will be filled by the Security Server when the coprocessor is called. Within the coprocessor code, the output CPRB fields are filled by copying the values from the input CPRB. Changing the values of these fields is not recommended, except for the replied_parm_block_length and replied_data_block_length fields in the coprocessor code.

Changing the values in a “FILLED AUTOMATICALLY” field will have one of two effects:

1. SECY will overwrite the changed value with the correct value.
2. The call will fail because of an invalid value.

Since the CPRB_structure is used exclusively as a pointer into the REQUEST_REPLY_BUF structure, the type CPRB_ptr has been typedefed as a pointer to the CPRB_structure.
**CPRB_structure, or *CPRB_ptr**

<table>
<thead>
<tr>
<th>Size of field</th>
<th>Field name</th>
<th>Purpose</th>
<th>Filled automatically</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint16_t</td>
<td>CPRB_length</td>
<td>CPRB length in bytes</td>
<td></td>
</tr>
<tr>
<td>1 byte</td>
<td>cprb_version_id</td>
<td>Version of the CPRB</td>
<td>✓</td>
</tr>
<tr>
<td>2 bytes</td>
<td>cprb_rsvd_01</td>
<td>Reserved field</td>
<td>✓</td>
</tr>
<tr>
<td>1 byte</td>
<td>cprb_internal_flags</td>
<td>Indicators about CPRB internals</td>
<td>✓</td>
</tr>
<tr>
<td>2 bytes</td>
<td>function_id</td>
<td>Requested function ID</td>
<td></td>
</tr>
<tr>
<td>2 bytes</td>
<td>cprb_flags</td>
<td>Indicators about request/reply</td>
<td>✓</td>
</tr>
<tr>
<td>uint32_t</td>
<td>req_parm_block_length</td>
<td>Request parameter block length</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>req_data_block_length</td>
<td>Request data block length</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>reply_msg_block_length</td>
<td>Reply message block length</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>replied_parm_block_length</td>
<td>Replied parameter block length</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>reply_data_block_length</td>
<td>Reply data block length</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>replied_data_block_length</td>
<td>Replied data block length</td>
<td></td>
</tr>
<tr>
<td>uint32_t</td>
<td>request_extension_block_len</td>
<td>Request ext. block length</td>
<td>✓</td>
</tr>
<tr>
<td>uint32_t</td>
<td>cprb_rsvd_01</td>
<td>Reserved</td>
<td>✓</td>
</tr>
<tr>
<td>uint32_t</td>
<td>replied_extension_block_len</td>
<td>Replied ext. block length</td>
<td>✓</td>
</tr>
<tr>
<td>Pointer</td>
<td>req_parm_block</td>
<td>Address of request parm block</td>
<td></td>
</tr>
<tr>
<td>(0/8 byte)</td>
<td>cprb_rsvd_02</td>
<td>Pad that precedes pointer, if any</td>
<td>✓</td>
</tr>
<tr>
<td>Pointer</td>
<td>req_data_block_addr</td>
<td>Address of request data block</td>
<td></td>
</tr>
<tr>
<td>(0/8 byte)</td>
<td>cprb_rsvd_03</td>
<td>Pad that precedes ptr, if any</td>
<td>✓</td>
</tr>
<tr>
<td>Pointer</td>
<td>reply_parm_block</td>
<td>Address of reply parameter block</td>
<td></td>
</tr>
<tr>
<td>(0/8 byte)</td>
<td>cprb_rsvd_04</td>
<td>Pad that precedes ptr, if any</td>
<td>✓</td>
</tr>
<tr>
<td>Pointer</td>
<td>reply_data_block</td>
<td>Address of reply data block</td>
<td></td>
</tr>
<tr>
<td>(0/8 byte)</td>
<td>cprb_rsvd_05</td>
<td>Pad that precedes ptr, if any</td>
<td>✓</td>
</tr>
<tr>
<td>Pointer</td>
<td>request_extension_block_addr</td>
<td>Request extension block addr</td>
<td></td>
</tr>
<tr>
<td>(0/8 byte)</td>
<td>cprb_rsvd_06</td>
<td>Pad that precedes ptr, if any</td>
<td>✓</td>
</tr>
<tr>
<td>Pointer</td>
<td>reply_extension_block_addr</td>
<td>Reply extension block address</td>
<td></td>
</tr>
<tr>
<td>(0/8 byte)</td>
<td>cprb_rsvd_07</td>
<td>Pad that precedes ptr, if any</td>
<td>✓</td>
</tr>
</tbody>
</table>
On the host workstation, you will only need one CPRB_ptr, since the request CPRB you build will be replaced by the reply CPRB from the coprocessor during the call to CSNC_SP_SCSRFBSS(). On the coprocessor, two of the parameters for a command function are pCprbIn, and pCprbOut. Therefore, you do not need to declare either a REQUEST_REPLY_BUF or a CPRB_ptr.

Following the CPRB_structure in the buffer is a request block:

The ESSS_request_block_structure defines the structure for the request or reply block. Since request and reply blocks are variable length, this structure is used purely as a pointer into the request_buf or reply_buf field of the REQUEST_REPLY_BUF structure. RBFPTR is typedefed as a pointer to an ESSS_request_block_structure, and thus is more commonly used.

On the host side, you may want to declare an RBFPTR for the request buffer. On the coprocessor code, you may want to declare an RBFPTR for both the request buffer and the reply buffer.

### Size of field | Field name | Purpose | Filled automatically
---|---|---|---
uint32_t | secy_return_code | Security server return code |
uint32_t | MAC_data_length | Length of MACed data area |
8 bytes | logon_identifier | Logon user identifier |
8 bytes | MAC_value | Resulting MAC value |
1 byte | MAC_content_flags | MAC data content flags |
1 byte | cprb_rsvd_08 | Reserved to maintain alignment |
uint16_t | Domain | Usage/control domain |
4 bytes | UsageDomainMask | Usage domain mask |
4 bytes | ControlDomainMask | Control domain mask |
4 bytes | S390EnforcementMask | S390 Enforcement mask |
36 bytes | cprb_rsvd_09 | Reserved for future use |
ESSS_request_block_structure, or *RBFPTR

<table>
<thead>
<tr>
<th>Size/Type</th>
<th>Field name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bytes</td>
<td>subfunction_code</td>
<td>Holds the two-byte subfunction code in big-endian format.</td>
</tr>
<tr>
<td>2 bytes</td>
<td>rule_array_length</td>
<td>Total length of the rule array and this field, in big-endian order.</td>
</tr>
<tr>
<td>1 byte</td>
<td>first_rule_array_element</td>
<td>First character of first rule array element, if rule_array_length is greater than 2; otherwise, this will be the first byte of the verb unique data length field.</td>
</tr>
</tbody>
</table>

Filling the rule array is easy using the BuildParmBlock() function:

```c
BuildParmBlock ( ptr1,
               1,
               SIZE_OF_RULE * (*pRuleCount), pRuleArray);
```

To parse a rule array with the rule_check() function, two more structures are used. A pointer to a RULE_BLOCK is passed to the function to be parsed. Note that the rule array format within the ESSS_request_block structure is, in fact, a RULE_BLOCK.

RULE_BLOCK

<table>
<thead>
<tr>
<th>Size/Type</th>
<th>Field name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned short</td>
<td>length</td>
<td>Total length of rule block (big-endian).</td>
</tr>
<tr>
<td>80 bytes</td>
<td>data</td>
<td>Up to 10 (8-byte) rules.</td>
</tr>
</tbody>
</table>

The other structure required is a RULE_MAP structure. This maps 8-byte strings into a value array, assigning a unique value to each string, and 1 or more strings to each position in the array, depending on mutual exclusion issues.

RULE_MAP

<table>
<thead>
<tr>
<th>Size/Type</th>
<th>Field name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 bytes (8 chars plus null terminator)</td>
<td>keyword</td>
<td>String to be matched in rule array.</td>
</tr>
<tr>
<td>1 byte</td>
<td>order_no</td>
<td>Group number: all rules which are mutually exclusive to each other will have the same group number.</td>
</tr>
<tr>
<td>int (4 bytes)</td>
<td>map_value</td>
<td>The numeric value associated with this rule.</td>
</tr>
</tbody>
</table>

To check the values in the rule array, use the rule_check() function:

```c
rule_check((RULE_BLOCK *)&pReq->rule_array_length,
           sizeof(aRuleMap)/sizeof(RULE_MAP),
           aRuleMap,
           aRuleValue,
           &returnMessage);
```
Immediately following the rule array in the REQUEST_REPLY_BUF is the verb unique data. Two types of structures are supplied for working with verb unique data, the VUD_DATA_RECORD, which is a length/tag/data structure (the data preceded by a DATA_RECORD_HEADER structure), and the verb_unique_data_structure, which is a length/data structure.

**DATA_RECORD_HEADER**

<table>
<thead>
<tr>
<th>Size</th>
<th>Field name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned short</td>
<td>Length</td>
<td>Length of this verb data (big-endian).</td>
</tr>
<tr>
<td>unsigned short</td>
<td>Flag</td>
<td>User defined: usually type of data.</td>
</tr>
</tbody>
</table>

```
#define DATA_HEADER_LENGTH sizeof( DATA_RECORD_HEADER )
```

If you want to use the length/tag/data format for your verb unique data, declare a DATA_RECORD_HEADER structure to place before the data, and use the BuildParmBlock() function to place it before the data.

```
BuildParmBlock(ptr,
    2,
    DATA_HEADER_LENGTH, &DataHeader,
    dataLength, Data);
```

The FindFirstDataBlock() function returns a pointer to a VUD_DATA_RECORD, so that you can access your data in this format easily.

**VUD_DATA_RECORD**

<table>
<thead>
<tr>
<th>Size/Type</th>
<th>Field name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned short</td>
<td>Length</td>
<td>Length of this verb data (big-endian).</td>
</tr>
<tr>
<td>unsigned short</td>
<td>Flag</td>
<td>See above DATA_RECORD_HEADER</td>
</tr>
<tr>
<td>1 byte</td>
<td>Data</td>
<td>The first byte of the data.</td>
</tr>
</tbody>
</table>

```
FindFirstDataBlock(pCPRB, SEL_REPLY_BLK, &pVerbDataRecord);
if(pVerbDataRecord->Flag == EncryptedKey) {
    memcpy(pKeyParameter, &pVerbDataRecord->Data,
        pVerbDataRecord->Length - DATA_HEADER_LENGTH);
}
```

On the other hand, if you have no need to access the Flags field, you can use the verb_unique_data_structure type instead:

**verb_unique_data_structure**

<table>
<thead>
<tr>
<th>Size/Type</th>
<th>Field name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned short</td>
<td>verb_unique_data_length</td>
<td>Length of this verb data (big-endian).</td>
</tr>
</tbody>
</table>
BuildParmBlock(ptr,
    2,
    sizeof(short), &vudLength,
    dataLength, &Data);

To retrieve the above data, you must first cast the verb_unique_data_structure as a VUD_DATA_RECORD:

FindFirstDataBlock ( pCPRB,
    SEL_REPLY_BLK,
    (VUD_DATA_RECORD **)&pVerbUniqueDataStructure);
*pLengthParm = atohs(pVerbUniqueDataStructure->verb_unique_data_length) -
    LENGTH_FIELD_SIZE;
memcpy(pReturnedData,
    &pVerbUniqueDataStructure->verb_unique_data,
    *pLengthParm);

If the only piece of data which is being passed has a fixed length (for example, if it is a structure), you need not use either of the verb structures shown:

BuildParmBlock(ptr,
    1,
    sizeof(Structure), &Structure);

Then to access the data:

FindFirstDataBlock(pCPRB,
    SEL_REPLY_BLOCK,
    (VUD_DATA_RECORD **)&pData);
memcpy(pStructure, pData, sizeof(*pStructure));

If you use this method, you must not pass more than one piece of verb unique data, as the FindNextDataBlock() function uses the length field to determine where to look for the next piece of data.

Following the verb unique data, the key data is organized into key fields and key data structures. Each key is preceded by a KEY_FIELD_HEADER structure.

**KEY_FIELD_HEADER**

<table>
<thead>
<tr>
<th>Size/Type</th>
<th>Field name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned short</td>
<td>Length</td>
<td>Total length of this key block (big-endian).</td>
</tr>
<tr>
<td>unsigned short</td>
<td>Flags</td>
<td>Flags indicating action required by the Security server and type of key (big-endian).</td>
</tr>
</tbody>
</table>
On the host side, you will need to declare a KEY_FIELD_HEADER structure for each key you will be passing to the coprocessor. On the coprocessor, you will need to declare a KEY_FIELD_HEADER structure for each key you will be passing to the host. If you are passing a token to be written to the key storage file, you must declare two KEY_FIELD_HEADER structures, and pass first the label of the key to write to, then the key token to write into the key storage file.

\[
\text{BuildParmBlock}(\text{ptr}, \\
4, // 2 for each key you will be passing \\
\text{sizeof}(\text{KEY_FIELD_HEADER}), \&\text{keyFieldHeader1}, \\
\text{KEY_LABEL_LENGTH}, \text{keyLabel}, \\
\text{sizeof}(\text{KEY_FIELD_HEADER}), \&\text{keyFieldHeader2}, \\
\text{keyTokenLength}, \text{keyToken});
\]

The \text{find_first_key_block()} function returns a pointer to a \text{key_data_structure}:

\begin{verbatim}
key_data_structure
\end{verbatim}

\[
\begin{array}{|c|c|}
\hline
\text{Size/Type} & \text{Field name} & \text{Purpose} \\
\hline
\text{unsigned short} & \text{key_field_data_length} & \text{Total length of this key data (big-endian).} \\
\hline
\text{1 byte} & \text{key_data} & \text{First byte of keyFieldHeader.Flags} \\
\hline
\end{array}
\]

Since there is no reason to access the first byte of the keyFieldHeader.Flags field, you will usually declare a \text{generic_key_block_structure} pointer, and cast it as a \text{key_data_structure} in the function call.

\begin{verbatim}
generic_key_block_structure
\end{verbatim}

\[
\begin{array}{|c|c|}
\hline
\text{Size/Type} & \text{Field name} & \text{Purpose} \\
\hline
\text{unsigned short} & \text{length} & \text{Total length of this key data (big-endian).} \\
\hline
\text{unsigned short} & \text{flags} & \text{Flag bytes (big-endian). (ignore)} \\
\hline
\text{1 byte} & \text{label_or_token} & \text{First byte of key token or label.} \\
\hline
\end{array}
\]

\[
\text{find_first_key_block}(\text{pCprb}, \\
(\text{key_data_structure} **)&\text{pGenericKeyBlockStructure}, \\
\text{SEL_REQ_BLK});
\]

\[
\text{keyLength} = \text{atohs}(\text{pGenericKeyBlockStructure->length}) - \\
\text{sizeof}(\text{KEY_FIELD_HEADER});
\]

\[
\text{pKeyToken} = &\text{pGenericKeyBlockStructure->label_or_token};
\]
Notice that the value of the byte in the label_or_token field can be used in the macro
TOKEN_LABEL_CHECK to determine whether the token is a key token with key data or the label of a key
in key storage.

If the key which has been passed is an RSA key, some of the functions which manipulate and check it take
parameters of type RsaKeyTokenHeader:

**RsaKeyTokenHeader**

<table>
<thead>
<tr>
<th>Size/Type</th>
<th>Field name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte</td>
<td>tokenId</td>
<td>Indicates Internal PKA, External PKA, Label, or “not RSA.”</td>
</tr>
<tr>
<td>1 byte</td>
<td>version</td>
<td>Version of RSA token.</td>
</tr>
<tr>
<td>2 byte int</td>
<td>tokenLength</td>
<td>Total length of token (big-endian).</td>
</tr>
<tr>
<td>4 byte int</td>
<td>reserved</td>
<td>Valued to 0.</td>
</tr>
</tbody>
</table>
| 1 byte    | nextSection| First byte of next token section - indicates public or privateMo-
|           |            | dexponent, private Chinese remainder, and so on. |

In most cases, you should simply cast the pointer to the token as an RsaKeyTokenHeader pointer.

**Data Structures for Caching Functions**

Only one data structure is required for the use of the cache functions, the short_tag_t:

**short_tag_t**

<table>
<thead>
<tr>
<th>Size/Type</th>
<th>Field name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte</td>
<td>tag_1</td>
<td>First byte of 2 byte short tag, index into linked list of second bytes.</td>
</tr>
<tr>
<td>1 byte</td>
<td>tag_2</td>
<td>Second byte of 2 byte short tag, index into linked list of entries.</td>
</tr>
</tbody>
</table>

You may choose to cast a 2-byte value as a short_tag_t for the function call.

**Other Useful Data Structures**

The mk_selectors data structure is used to indicate which of several master keys to use in a given master
key function. There are separate old, current, and new master keys for asymmetric and symmetric master keys in each available domain.

**mk_selectors**

<table>
<thead>
<tr>
<th>Size/Type</th>
<th>Field name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned short</td>
<td>mk_set</td>
<td>Domain of master key set: This should contain the same value as the domain field in the input CPRB structure.</td>
</tr>
<tr>
<td>enumeration old_mk, current_mk, new_mk</td>
<td>mk_register</td>
<td>To determine which of the three registers to access.</td>
</tr>
</tbody>
</table>
The `RsaRecoverClearKeyTokenUnderXport()` function requires a type of `double_length_key`.

### double_length_key

<table>
<thead>
<tr>
<th>Size/Type</th>
<th>Field name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bytes</td>
<td>left</td>
<td>The first 8 bytes of the key.</td>
</tr>
<tr>
<td>8 bytes</td>
<td>right</td>
<td>The final 8 bytes of the key.</td>
</tr>
</tbody>
</table>

The functions `load_first_mk_part()` and `combine_mk_parts()` require a `TRIPLE_LENGTH_KEY`:

### TRIPLE_LENGTH_KEY

<table>
<thead>
<tr>
<th>Size/Type</th>
<th>Field name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bytes</td>
<td>first</td>
<td>The first 8 bytes of the key.</td>
</tr>
<tr>
<td>8 bytes</td>
<td>middle</td>
<td>The middle 8 bytes of the key.</td>
</tr>
<tr>
<td>8 bytes</td>
<td>last</td>
<td>The final 8 bytes of the key.</td>
</tr>
</tbody>
</table>
Appendix H. Moving a UDX from the IBM 4758 to the IBM 4764 Coprocessor

Byte Order Changes
The IBM 4764 runs on a PowerPC chip, so all message lengths which are used or validated by the coprocessor side code must be passed in big-endian order, rather than the little-endian order used by the IBM 4758. This includes the rule array length, vud length, and key block length fields in the request parameter structure, as well as the key header Length and Flag fields, and any vud lengths which will be processed using FindFirstDataBlock or FindNextDataBlock on either the host or the coprocessor.

The macro htoas() has been provided to do the swapping for workstation host code. zOS code for the IBM 4758 may have been written with the swapping included (to swap from big-endian to little-endian). If your zOS code swapped bytes manually, you need to remove this. If lengths were passed to the CSFADSPI macro, however, the macro provided the swapping and the only changes to be made are to switch to the CSFGAPSI macro.

Also, because of the byte swapping, the byte-order of your subfunction code (the numeric value used in the UDX command extension array) must be changed to match the string value.

Be aware that the byte-order of fields in the PKA key tokens has not changed - they are still big-endian. If you are accessing length fields in a PKA key token on the coprocessor, you must remove the intel_word_reverse() call used to translate the token format into little-endian order. Also, the token validation value field of a DES key token is still stored in big-endian order. Thus, if you have calculated it and then reversed (as was done in the UDX sample code zudx.c) you must remove the reversal.

Makefile Changes
The IBM 4764 toolkit is designed to be installed on a SLES 9 system, using a gcc cross-compiler for the powerPC chip for card-side code. Thus, the coprocessor makefiles are completely new. References to the SCCTK_FS_ROOT environment variable have been transferred to XCTK_FS_ROOT.

Changes for the coprocessor makefile
There are three example makefiles in the new toolkit for the coprocessor code. You will first use either the ~/udxtk/wks/card/cardinx.mak makefile (to build code without zSeries restrictions ) or the ~/udxtk/zos/card/zos/cardzos.mak makefile to build code using the zSeries library. Then you must modify the ~/udxtk/build/Makefile makefile to determine which executable to place on the coprocessor, and whether or not to include the debugger. This makefile copies the executable, and (if needed) the debugger daemon into the correct places in the user0 directory, then changes the ownership and permissions of the files and directories so that the udx can correctly access the SRDIs on the coprocessor, and finally calls sudo mkfs to create the segment3.bin file which will be loaded on the coprocessor.

It is important that the executable which is placed on the card be named "csulcca". This is required by the startup function. The name of the .bin file is up to you.

The cpqxtl.exe utility has been eliminated. You must now use a cross-compiler to build the code on the (Intel) Linux workstation for the (PowerPC) coprocessor. The makefile expects the cross-compiler to be found at the "CROSS" environment variable.
The `sccrodsk.exe` utility has been eliminated. You must now use the `mkfs` command to build a JFFS2 filesystem for the flash memory on the coprocessor.

**Changes for the host workstation makefile**

On Linux, the host makefile should use the normal gcc compiler for SLES 9.

On Windows, the host makefile should use Microsoft Visual C++ 6.0.

On AIX, the host makefile should use xlc_r.

**Changes to function calls**

**In zOS**

The CSFADSPI macro has been replaced by the CSFGAPSI macro. The new macro has no “SERVICE_ALET” parameter. Instead, separate parameters for the GAPSI_REQ_ALET and the GAPSI_REPLY_ALET are used.

**On the coprocessor**

The *IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Interface Reference* contains a complete list of changes in the low-level API. Each of the low-level API calls requires a file descriptor for the hardware used in the call. The file descriptor is returned by the function `CasGetFileDescriptor()`, which is described in Chapter 12, “Miscellaneous Functions.”

The PKA length functions which return integer results (for example, GeteLength), now return them in big-endian order, which is native byte-order on the coprocessor.

The functions `des_engine_edec2_triple_encrypt` and `des_engine_edec2_triple_decrypt` have been added to simplify TDES operations.

The function `S390KeyLength` has been added. This function determines the length of a DES key from the control vector, if it is not a zero-value control vector, or the key token itself, if the control vector is all zeroes. This causes the function to return the correct key length for an external DATA key, which has a control vector of all zeroes.

In the SHA-1 functions, the “dbl_ulong” type parameter has been replaced by a “long long” parameter.

**Data Block Changes**

With the 4758 card, the process for receiving large amounts of data on the coprocessor was to use the SCC API function `sccGetBufferData` to DMA the data from the host system. With the 4764, the DMA is not required, you merely copy the data (pointed to by the `req_data_block_addr` field in the CPRB structure) into local memory.

Also, when sending large blocks of data from the coprocessor to the host, rather than calling the SCC API function `sccPutBufferData`, place the address of the data you wish moved in the `reply_data_block_addr` field of the CPRB structure, and the length of the field in the `reply_data_block_length` field of the structure. If the buffer was allocated dynamically, you must also set a bit in the CPRB structure to tell the CCA manager to free the buffer. To do this, logically OR the value `FREE_DATA_BLOCK` into the first byte of the `cprb_internal_flags` field of the CPRB structure.
Structure Changes

Many fields have changed in the CPRB structure. If your UDX for the 4758 used the CSUC_BULDCPRB function to fill the CPRB structure, only the following fields are of interest:

- The field formerly named reply_parm_block_length has been renamed reply_msg_block_length.
- A field has been added: The new field is cprb_internal_flags - see the discussion of passing large data blocks, above, for an explanation of this field.
- The CPRB structure is now 4 bytes longer in the libcsuzlib.so library. For this reason, you cannot run code built for the workstation host with code built for the zSeries coprocessor, but must use separate host-side libraries when testing zSeries code in a workstation.

The dbl_ulong structure is no longer used. Hashing functions which had used this structure to describe the number of bits to be hashed now use uint64_t variable (8 bytes).
Appendix I.

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.

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.
**Trademarks**

AIX, IBM, OS/2, OS/390, RS/6000, S/390, zSeries, and z/OS are registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Intel is a trademark of Intel Corporation in the United States, other countries, or both.

SET and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Windows and NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.
### 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>API</td>
<td>application program interface</td>
<td>OMK</td>
<td>old master key</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information</td>
<td>PCI</td>
<td>peripheral component interconnect</td>
</tr>
<tr>
<td></td>
<td>Interchange</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BBRAM</td>
<td>battery-backed random access memory</td>
<td>PCI-X</td>
<td>peripheral component interconnect extended</td>
</tr>
<tr>
<td>CCA</td>
<td>Common Cryptographic Architecture</td>
<td>PDF</td>
<td>portable document format</td>
</tr>
<tr>
<td>CMK</td>
<td>current master key</td>
<td>PIN</td>
<td>personal identification number</td>
</tr>
<tr>
<td>CPRB</td>
<td>Cooperative Processing Request Block</td>
<td>PKA</td>
<td>public key algorithm</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard</td>
<td>RAM</td>
<td>random access memory</td>
</tr>
<tr>
<td>DLL</td>
<td>dynamic load library</td>
<td>RNG</td>
<td>random number generator</td>
</tr>
<tr>
<td>EPROM</td>
<td>erasable programmable read-only memory</td>
<td>RSA</td>
<td>Rivest-Shamir-Adleman (algorithm)</td>
</tr>
<tr>
<td>FIPS</td>
<td>Federal Information Processing Standard</td>
<td>SET</td>
<td>Secure Electronic Transaction</td>
</tr>
<tr>
<td>KEK</td>
<td>key encrypting key</td>
<td>SHA</td>
<td>Secure Hash Algorithm</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines</td>
<td>SRDI</td>
<td>security relevant data item</td>
</tr>
<tr>
<td>MAC</td>
<td>message authentication code</td>
<td>TVV</td>
<td>token validation value</td>
</tr>
<tr>
<td>MKVP</td>
<td>master key verification pattern</td>
<td>UDX</td>
<td>user defined extension</td>
</tr>
<tr>
<td>NMK</td>
<td>new master key</td>
<td>xCrypto</td>
<td>Linux Cryptographic Coprocessor</td>
</tr>
</tbody>
</table>
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.

agent. (1) An application that runs within the xCrypto card. (2) Synonym for Linux cryptographic coprocessor application.

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 a user, 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 xCrypto card uses BBRAM to store persistent data for xCrypto applications, as well as the coprocessor device key.

BBRAM. Battery-backed random access memory.

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. (I) (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.

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 4758 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.

Cryptographic Coprocessor (xCrypto). 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.

data encrypting key. (1) A key used to encipher, decipher, or authenticate data. (2) Contrast with key-encrypting key.

Data Encryption Standard (DES). The National Institute of Standards and Technology (NIST) Data Encryption Standard, adopted by the U.S. government as Federal Information Processing Standard (FIPS) Publication 46, which allows only hardware implementation of the data encryption algorithm.

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

DES. Data Encryption Standard.

e
encipher. (1) To scramble data or convert it to a secret code that masks its meaning. (2) Contrast with decipher.

enciphered data. (1) Data whose meaning is concealed from unauthorized users or observers. (2) See also ciphertext.

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.

F
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.

H
host. As regards to the xCrypto card, the workstation into which the coprocessor is installed.

I
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. (2) Hardware, software, or both that links systems, programs, and devices.

K
key. In computer security, a sequence of symbols used with an algorithm to encipher or decipher data.
L

Linux cryptographic coprocessor (XC). (1) An application that runs within the IBM xCrypto Coprocessor. (2) Synonym for agent.

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.

N


NIST. National Institute of Science and Technology.

P

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.

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 xCrypto card uses a hardware-based system to generate true random values for cryptographic use.

return code. (1) A code used to influence the execution of succeeding instructions. (A) (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.

U

utility program. A computer program in general support of computer processes. (1)

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.

X

XC. Linux Cryptographic Coprocessor.
Index

Numerics
2-byte values, convert 13-7

A
ac_check_authorization 11-5
ac_chg_prof_auth_data 11-6
ac_chg_prof_exp_date 11-8
ac_del_profile 11-9
ac_del_role 11-10
ac_get_list_sizes 11-11
ac_get_profile 11-12
ac_get_role 11-13
ac_init 11-14
ac_list_profiles 11-15
ac_list_roles 11-16
ac_load_profiles 11-17
ac_load_roles 11-19
ac_lu_add_user 11-20
ac_lu_drop_user 11-21
ac_lu_get_ks 11-22
ac_lu_get_num_users 11-23
ac_lu_get_user 11-24
ac_lu_get_role 11-24
ac_lu_query_user 11-32
ac_query_role 11-34
ac_reinit 11-35
ac_reset_logon_fail_cnt 11-36
acm_xlt_errorcode 11-37
au_lu_get_role 11-24
au_lu_query_user 11-32
authority, user 13-3

B
building a CCA user defined extension 2-1
building a CPRB in the host 4-32
building a parameter block 4-26
building a UDX 2-1
BuildParmBlock 4-26
byte order changes, moving 4758 to 4764 H-1

C
cache management functions
  cache_clear 12-3
  cache_delete 12-4
  cache_delete_item 12-5
  cache_get_item 12-6
  cache_get_item_b 12-7
  cache_init 12-8
  cache_status 12-9
  cache_write_item 12-10
  overview 12-1
  cache_clear 12-3
  cache_delete 12-4
  cache_delete_item 12-5
  cache_get_item 12-6
  cache_get_item_b 12-7
  cache_init 12-8
  cache_status 12-9
  cache_write_item 12-10
  caching_functions, data structures G-8
  calculate_token_validation_value 9-34
  CalculateWordLength 9-7
  callable functions
    ac_check_authorization 11-5
    ac_chg_prof_auth_data 11-6
    ac_chg_prof_exp_date 11-8
    ac_del_profile 11-9
    ac_del_role 11-10
    ac_get_list_sizes 11-11
    ac_get_profile 11-12
    ac_get_role 11-13
    ac_lu_add_user 11-20
    ac_lu_drop_user 11-21
    ac_lu_get_ks 11-22
    ac_lu_get_num_users 11-23
    ac_lu_get_user 11-24
    ac_lu_get_role 11-24
    ac_lu_query_user 11-32
    ac_query_role 11-34
    ac_reinit 11-35
    ac_reset_logon_fail_cnt 11-36
    acm_xlt_errorcode 11-37
    adapter_interface 1-4
    adjust parity 8-4
    architecture of the UDX environment 1-1
    atohl 13-6
    atohs 13-6
    atoxl 13-6
    atoxs 13-7
    au_lu_get_role 11-24
    au_lu_query_user 11-32
    authority, user 13-3
ac_init 11-14
ac_list_profiles 11-15
ac_list_roles 11-16
ac_load_profiles 11-17
ac_load_roles 11-19
ac_lu_add_user 11-20
ac_lu_drop_user 11-21
ac_lu_get_ks 11-22
ac_lu_get_num_users 11-23
ac_lu_get_user 11-24
ac_lu_get_role 11-25
ac_lu_query_profile 11-26
ac_lu_query_role 11-27
ac_reinit 11-28
acm_xlt_errorcode 11-29
au_lu_get_role 11-30
au_lu_query_user 11-31
BuildParmBlock 4-26
cache_clear 12-3
cache_delete 12-4
cache_delete_item 12-5
cache_get_item 12-6
cache_get_item_b 12-7
cache_init 12-8
cache_status 12-9
cache_write_item 12-10
CalcutenWordLength 9-7
cas_adjust_parity 8-4
cas_build_default_cv 8-5
cas_build_default_token 8-6
cas_current_mkv 8-7
cas_des_key_token_check 8-9
cas_get_key_type 8-10
cas_key_length 8-11
cas_key_tokenvv_check 8-12
cas_master_key_check 8-13
cas_old_mkv 8-14
cas_parity_odd 8-15
cas_proc_retc 4-30
CasBuildCv 8-5
CasBuildToken 8-6
CasCurrentMkv 8-7
CasGetFileDescriptor 13-2
CasMasterKeyCheck 8-13
CasOldMkv 8-14
check_access_auth_fcn 13-3
clear_master_keys 6-6
close_cca_srdi 10-7
combine_mk_parts 6-7
compute_mk_verification_pattern 6-19
computeHMAC_SHA1 7-3
create_cca_srdi 10-8
create4update_cca_srdi 10-9
CreateInternalKeyToken 9-8
CreateInternalKeyTokenWithMK 9-8
CreateRsaInternalSection 9-9
CreateRsaInternalSectionWithMK 9-9
CSNC_SP_SCSRFBSS 4-31
CSUC_BULDCPRB 4-32
CSUC_PROCRETC 4-34
delete_cca_srdi 10-10
delete_KeyToken 9-10
d_get_sha_hash_message 7-4
d_sha_hash_msg_to_bfr 7-6
der3_triple_decrypt_under_master_key 6-26
der3_triple_encrypt_under_master_key 6-27
ExtractRsaPublicKeySection 9-11
find_first_key_block 4-37
find_next_key_block 4-38
FindFirstDataBlock 4-35
FindNextDataBlock 4-36
generate_dSSig 9-14
generate_mk_shares 6-9
generate_random_mk 6-11
GenerateCcaRsaToken 9-12
GenerateRsaInternalToken 9-13
get_cca_srdi_length 10-11
get_master_key_status 6-23
get_mk_verification 6-21
GeteLength 9-15
GetKeyLength 13-5
getKeyToken 9-16
GetModulus 9-17
GetNBitLength 9-18
GetNByteLength 9-19
GetPublicExponent 9-20
GetRSAKeyNameSection 9-21
GetRsaPrivateSection 9-22
GetRsaPublicKeySection 9-23
GetRsaPuKeyCertSection 9-24
getSymmetricMaxModulusLength 5-2
GetTokenLength 9-25
hw_sha_hash_message 7-7
init_master_keys 6-12
InitCprbParmPointers 4-39
intel_long_reverse 13-6
intel_word_reverse 13-7
isFunctionEnabled 5-3
IsPrivateExponentEven 9-26
IsPrivateKeyEncrypted 9-27
IsPublicExponentEven 9-28
IsRsaToken 9-29
IsS390Token 9-30
IsTokenInternal 9-31
keyword_in_rule_array 4-40
load_first_mk_part 6-13
mkmCombineMKParts 6-7
mkmGenerateMKParts 6-7
mkmGenerateRandomMK 6-11
mkmGetAsymVerificationPattern 6-22
mkmGetMasterKeyStatus 6-23
mkmLoadFirstMKPart 6-13
mkmLoadMKFromShares 6-14
mkmSetMasterKey 6-16
mkmTDESEncryptUnderMasterKey 6-28
mkmTDESDecryptUnderMasterKey 6-29
open_cca_srdi 10-12
parm_block_valid 4-41
pka96_tvvgen 9-34
PkaHashQueryWithMK 9-32
PkmvppQuery 9-33
PkmvppQueryWithMK 9-33
RecoverDesDataKey 8-18
RecoverDesDataKeyWithMK 8-18
RecoverDesKekImporter 8-20
RecoverDesKekImporterWithMK 8-20
RecoverPkaClearKeyTokenUnderXport 9-35
RecoverPkaClearTokenUnderMk 9-36
RecoverPkaClearTokenUnderMkWithMKWithMinModulus 9-36
ReEncipherPkaKeyToken 9-38
ReEncipherPkaKeyTokenWithMK 9-38
Reinit_master_keys 6-17
RequestRSACrypto 9-39
resize_cca_srdi 10-13
rule_check 4-42
saf_process_key_label 4-45
save_cca_srdi 10-14
sccRSAKeyGenerate 9-40
set_master_key 6-16
sha_hash_message 7-9
store_KeyToken 9-41
TOKEN_IS_A_LABEL 13-12
TOKEN_LABEL_CHECK 13-13
TokenMkvpMatchMasterKey 9-42
triple_decrypt_under_master_key 6-30
triple_decrypt_under_master_key_with_CV 6-31
triple_encrypt_under_master_key 6-32
triple_encrypt_under_master_key_with_CV 6-33
update_cca_srdi 10-15
ValidatePkaToken 9-43
verify_dSig 9-45
VerifyKeyTokenConsistency 9-44
cas_adjast_partly 8-4
cas_build_default_cv 8-5
cas_build_default_token 8-6
cas_current_mkvp 8-8
cas_des_key_token_check 8-9
cas_get_key_type 8-10
cas_key_length 8-11
cas_key_tokenvvv_check 8-12
cas_master_key_check 8-13
cas_old_mkvp 8-14
cas_parity_odd 8-15
Cas_proc_retc 4-30
CasBuildCv 8-5
CasBuildToken 8-6
CasCurrentMkvp 8-8
CasGetFileDescriptor 13-2
CasMasterKeyCheck 8-13
CasOldMkvp 8-14
CCA communication structures 1-11
chaining, SHA-1 hash 7-9
changes to function calls, moving 4758 to 4764 H-2
check authorization, Access Control Manager functions 11-5
CHECK_ACCESS_AUTH
See check_access_auth_fcn
check_access_auth_fcn 13-3
clear cache 12-3
clear_master_keys 6-6
close_cca_srdi 10-7
code sample
coprocessor piece D-1
workstation host piece E-1
z/OS host piece C-1
combine master key 6-7
combine_mk_parts 6-7
command processor 1-1, 1-11
command processor API, defining 2-7
command processor, identifier 1-3
command processors to array, adding 2-9
communication functions
BuildParmBlock 4-26
Cas_proc_retc 4-30
CSNC_SP_SCSRFBSS 4-31
CSUC_BULDPRPB 4-32
CSUC_PROCRETC 4-34
find_first_key_block 4-37
find_next_key_block 4-38
FindFirstDataBlock 4-35
FindNextDataBlock 4-36
InitCprbParmPointers 4-39
keyword_in_rule_array 4-40
parm_block_valid 4-41
rule_check 4-42
communication structures, CCA 1-11
communications, structures used between host and coprocessor G-1
CPRB_structure or *CPRB_ptr G-1, G-2
data structures for caching functions G-8
DATA_RECORD_HEADER structure G-5
ESSS_request_block_structure G-3
FindFirstDataBlock() function G-5
generic_key_block_structure G-7
key_data_structure G-7
KEY_FIELD_HEADER structure G-6
REQUEST_REPLY_BUF G-1
RsaKeyTokenHeader G-8
rule_check () function G-4
verb_unique_data_structure G-5
completion codes, defining new 2-6
compute verification pattern, master key 6-19
compute_mk_verification_pattern 6-19
computeHMAC_SHA1 7-3
control points 1-10
cooporative processing request/reply block (CPRB) 1-3
coprocessor

UDX structure 1-8

adding command processors to the array 2-9
building the executable 2-10
defining access control points 2-8
defining command processor API 2-7
designing and coding the logic 2-9

coprocessor piece sample code D-1

CPRB 1-3
CPRB parameter pointers, initialize 4-39
CPRB, building in the host 4-32
CPRB_ptr G-2
create, master key 6-12
create_cca_srdi 10-9
CreateInternalKeyToken 9-8
CreateInternalKeyTokenWithMK 9-8
CreateRsalInternalSection 9-9
CreateRsalInternalSectionWithMK 9-9
CSFACKDS 4-3
CSFADSCP 4-14
CSFAKDS 4-6
CSFAPKTV 4-19
CSFASEC 4-25
CSFGAPSI 4-21
CSFGCCPN 4-8
CSFGCPRB 4-11
CSFGPBK 4-17
CSFGVPBL 4-15
CSNC_SP_SCSRFBSS 4-31
CSUC_BULDCPRB 4-32
CSUC_PROCRETC 4-34
current master key verification pattern 8-8
CXT_SAMP.C listing D-1

D
data block changes, moving 4758 to 4764 H-2
data structures
communications between host and coprocessor G-1
data structures, Access Control Manager
generic 11-3
profile 11-3
role 11-3
user information 11-4
data structures, other G-8
double_length_key G-9
mk_selectors G-8
TRIPLE_LENGTH_KEY G-9
data, clear from cache 12-3
decrypted private keys, cache 12-1
default control vector, build 8-5
default token, build 8-6
delete_cca_srdi 10-10
delete_KeyToken 9-10
DES data key, recover 8-18
DES importer KEK, recover 8-20
DES key token, verify 8-9

DES utility functions

cas_adjust_parity 8-4
cas_build_default_cv 8-5
cas_build_default_token 8-6
cas_current_mkvp 8-8
cas_des_key_token_check 8-9
cas_get_key_type 8-10
cas_key_length 8-11
cas_key_token_check 8-12
cas_master_key_check 8-13
cas_old_mkvp 8-14
cas_parity_odd 8-15
CasBuildCv 8-5
CasBuildToken 8-6
CasCurrentMkvp 8-8
CasMasterKeyCheck 8-13
CasOldMkvp 8-14
RecoverDesDataKey 8-18
RecoverDesDataKeyWithMK 8-18
RecoverDesKekImporter 8-20
RecoverDesKekImporterWithMK 8-20
development overview 2-1
do_sha_hash_message 7-4
do_sha_hash_msg_to_bfr 7-6

E
dede3_triple_decrypt master key 6-26
ede3_triple_encrypt master key 6-27
ede3_triple_decrypt_under_master_key 6-26
ede3_triple_encrypt_under_master_key 6-27
enabled function, check 5-3
environment, understanding 1-1
ExtractRsaPublicSection 9-11

F
generated C source, coprocessor 2-2
generated header, coprocessor 2-2
makefile 2-2
func cio C source, host 2-1
header, host 2-1
header, host and coprocessor 2-1
files you use in building a UDX 2-1
find address of next key data block 4-38
find_first_key_block 4-37
find_next_key_block 4-38
FindFirstDataBlock 4-35
FindNextDataBlock 4-36
first data block, search for address 4-35
first key data block, search 4-37
function control vector management functions
getSymmetricMaxModulusLength 5-2
isFunctionEnabled 5-3
functions
coprocessor-side 3-1
host-side 3-1
See callable functions
G
generate random, master key 6-11
generate shares, master key 6-9
generate_dSig 9-14
generate_mk_shares 6-9
generate_random_mk 6-11
GenerateCcaRsaToken 9-12
GenerateRsalInternalToken 9-13
get_cca_srdi_length 10-11
get_master_key_status 6-23
get_mk_verification_pattern 6-21
GeteLength 9-15
GetKeyLength 13-5
getKeyToken 9-16
GetModulus 9-17
GetnBitLength 9-18
GetnByteLength 9-19
GetPublicExponent 9-20
GetRSAKeyNameSection 9-21
GetRsaPrivateKeySection 9-22
GetRsaPuKeyCertSection 9-24
GetSymmetricMaxModulusLength 5-2
GetTokenLength 9-25
H
hardware, calculate SHA-1 hash 7-4
hash in hardware, compute SHA-1 7-7
hash of requested data, SHA-1 7-7
hash, calculate SHA-1 7-4
hash, SHA-1 wrapper 7-6
hashing functions, SHA-1 7-1
header files
  Access Control Manager functions 11-1
  Caching functions 12-1
  Communications functions 4-1
  DES utility functions 8-1
  Function Control Vector functions 5-1
  Master Key Manager functions 6-1
  Miscellaneous functions 13-1
  RSA functions 9-1
  SHA-1 functions 7-1
  SRDI Manager functions 10-1
HMAC-SHA1, computing 7-3
host piece of a UDX
  installing 2-12
  libraries 2-7
  logic, designing and coding 2-6
  shared object file, building 2-6
htoal 13-6
htoas 13-6, 13-7
htoxl 13-6
htoxs 13-7
hw_sha_hash_message 7-7
I
init_master_keys 6-12
InitCprbParmPointers 4-39
initialize CPRRB parameter pointers 4-39
initial master key 6-12
initialize the Access Control Manager 11-14
installing the host piece of a UDX 2-12
int_long_reverse 13-6
  atoih 13-6
  atois 13-6
  atoxl 13-6
  htoal 13-6
  htoas 13-6
  htoxl 13-6
  itoal 13-6
  itoas 13-6
  stao 13-6
  xoal 13-6
  xtoal 13-6
  xtoh 13-6
internal key token, create 9-8
isFunctionEnabled 5-3
IsPrivateExponentEven 9-26
IsPrivateKeyEncrypted 9-27
IsPublicExponentEven 9-28
IsRsaToken 9-29
IsS390Token 9-30
IsTokenInternal 9-31
K
key blocks 1-3, 1-4, 1-15
key label, checking and processing 4-45
key labels 1-3
key length, return 8-11
key record 1-4
key register status 6-23
key storage server 1-4
key token
  consistency, verify 9-44
  format 9-31
  length 9-25
  signature 9-45
key tokens 1-11
key type, return 8-10
keyword_in_rule_array 4-40
L
LIBR_Alloc_Storage 13-8
LIBR_FreeStorage 13-9
LIBR_ReleaseSemaphore 13-10
LIBR_RequestSemaphore 13-11
load first part, master key 6-13
load shares, master key 6-14
load_first_mk_part 6-13
load_mk_from_shares 6-14
logged on users, Access Control Manager
add to list 11-20
copy session key 11-22
list 11-31
number of 11-23
query 11-32
remove 11-21
role 11-24
logon failure count, reset (Access Control Manager) 11-36
long values, convert 13-6
ltoal 13-6
ltoas 13-6,
M
makefile changes, moving 4758 to 4764 H-1
Master Key Manager (CCA) functions
clear_master_keys 6-6
common processing 6-3
compute_mk_verification_pattern 6-19
ede3_triple_decrypt_under_master_key 6-26
ede3_triple_encrypt_under_master_key 6-27
generate_mk_shares 6-9
generate_random_mk 6-11
get_master_key_status 6-23
get_mk_verification_pattern 6-21
header files 6-1
init_master_keys 6-12
initialization of the SRDI 6-2
initializing master keys 6-12
key register status 6-2, 6-23
load_first_mk_part 6-13
load_mk_from_shares 6-14
location 6-2
master key registers 6-1
mkmCombineMKParts 6-7
mkmGenerateMKShares 6-9
mkmGenerateAsymVerificationPattern 6-22
mkmGenerateMasterKeyStatus 6-23
mkmLoadFirstMKPart 6-13
mkmLoadMKFromShares 6-14
mkmSetMasterKey 6-16
mkmTDESDecryptUnderMasterKey 6-28
mkmTDESEncryptUnderMasterKey 6-29
overview 6-1
reinit_master_keys 6-17
reinitializing 6-17
required variables 6-3
set_master_key 6-16
TDESDecryptUnderMasterKey 6-28
TDESEncryptUnderMasterKey 6-29
test encryption 9-42
triple_decrypt_under_master_key 6-30
triple_decrypt_under_master_key_with_CV 6-31
triple_encrypt_under_master_key 6-32
triple_encrypt_under_master_key_with_CV 6-33
variables, required 6-3
verification pattern 6-2
version 9-33
version check 8-13
master key status 6-23
master key, return version 9-33
master key, set 6-16
miscellaneous functions
Cas_proc_retc 4-30
CasGetFileDescriptor 13-2
check_access_auth_fcn 13-3
GetKeyLength 13-5
intel_long_reverse 13-6
intel_word_reverse 13-7
LIBR_Alloc_Storage 13-8
LIBR_FreeStorage 13-9
LIBR_ReleaseSemaphore 13-10
LIBR_RequestSemaphore 13-11
TOKEN_IS_A_LABEL 13-12
TOKEN_LABEL_CHECK 13-13
mkmCombineMKParts 6-7
mkmGenerateMKShares 6-9
mkmGenerateRandomMK 6-11
mkmGetAsymVerificationPattern 6-22
mkmGetMasterKeyStatus 6-23
mkmLoadFirstMKPart 6-13
mkmLoadMKFromShares 6-14
mkmSetMasterKey 6-16
mkmTDESDecryptUnderMasterKey 6-28
mkmTDESEncryptUnderMasterKey 6-29
moving a UDX from the 4758 to 4764 H-1
byte order changes H-1
changes to function calls H-2
data block changes H-2
makefile changes H-1
structure changes H-3
N
next data block, search for address 4-36
next key data block, find address 4-36
O
old master key verification pattern 8-14
open_cca_srdi 10-12
overview 9-4
overview of cache functions 12-1
overview, development 2-1
P
parameter block
building 4-26
examine 4-41
verify 4-41
parity, adjust 8-4
parity, verify 8-15
parm_block_valid 4-41
parts, master key 6-7
PKA clear key
clear under DES export key, recover 9-35
recover under master key 9-36
re-encipher 9-38
pka96_tvvgen 9-34
PkaHashQueryWithMK 9-32
PkaMkvpQuery 9-33
PkaMkvpQueryWithMK 9-33
process key label 4-45

R
RBFPTR G-4
recover DES data key 8-18
recover DES importer KEK 8-20
RecoverDesDataKey 8-18
RecoverDesDataKeyWithMK 8-18
RecoverDesKekImporter 8-20
RecoverDesKekImporterWithMK 8-20
RecoverPkaClearKeyTokenUnderXport 9-35
RecoverPkaClearTokenUnderMk 9-36
RecoverPkaClearTokenUnderMkWithMKWithMinModulus 9-36
ReEncipherPkaKeyToken 9-38
ReEncipherPkaKeyTokenWithMK 9-38
reinit_master_keys 6-17
reinitialize master key 6-17
reinitialize the Access Control Manager 11-35
request and reply blocks, format 1-11
request parameter block 1-3, 1-4
request, sending to the coprocessor 4-31
requested data, SHA-1 hash 7-7
RequestRSACrypto 9-39
reserved values F-1
resize_cca_srdi 10-13
return code, prioritize 4-30, 4-34
return key length 8-11
return key type 8-10
return master key version 9-33
role, Access Control Manager
delete 11-10
get information 11-13
length 11-34
list 11-16
load 11-19
size of profile list 11-11
RSA 9-22
RSA functions
CalculateWordLength 9-7
CreateInternalKeyToken 9-8
CreateInternalKeyTokenWithMK 9-8
CreateRsaInternalSection 9-9
CreateRsaInternalSectionWithMK 9-9
delete_KeyToken 9-10
ExtractRsaPublicKeySection 9-11
generate_dSig 9-14
GenerateCcaRsaToken 9-12
GenerateRsaInternalToken 9-13
GetLength 9-15
getKeyToken 9-16
GetModulus 9-17
GetnBitLength 9-18
GetnByteLength 9-19

GetPublicExponent 9-20
GetRSAKeyNameSection 9-21
GetRsaPrivateKeySection 9-22
GetRsaPublicKeySection 9-23
GetRsaPuKeyCertSection 9-24
GetTokenLength 9-25
IsPrivateExponentEven 9-26
IsPrivateKeyEncrypted 9-27
IsPublicKeyExponentEven 9-28
IsRsaToken 9-29
IsS390Token 9-30
IsTokenInternal 9-31
pka96_tvvgen 9-34
PkaHashQueryWithMK 9-32
PkaMkvpQuery 9-33
PkaMkvpQueryWithMK 9-33
RecoverPkaClearKeyTokenUnderXport 9-35
RecoverPkaClearTokenUnderMk 9-36
RecoverPkaClearTokenUnderMkWithMKWithMinModulus 9-36
ReEncipherPkaKeyToken 9-38
ReEncipherPkaKeyTokenWithMK 9-38

RSA key
format 9-31
generate CCA RSA key token 9-12
length 5-2, 13-5
validate 9-43
verify signature 9-45
RSA key token
generate 9-13
verify 9-29, 9-44, 9-45
RSA modulus
bit length 9-18
byte length 9-19
extract and copy 9-20
RSA private exponent, verify 9-26
RSA public exponent
byte length 9-15
generate_dSig 9-14
get PKA token 9-16
validate 9-43
verify 9-29
rule array
checking for CSUAACI 4-44
CSNBPKI 4-43
rule map example 4-44
CSUAACI
rule map example 4-44
verify 4-42
rule array keyword, search 4-42
rule_check 4-42
S
S390 key token
verify 9-30
saf_process_key_label 4-45
sample code
coprocessor piece D-1
workstation host piece E-1
z/OS host piece C-1
save_cca_srdi 10-14
sccRSAKeyGenerate 9-40
search for first key data block 4-37
security relevant data items (SRDI) 1-1
security server (SECY) 1-4
sending a request to the coprocessor 4-31
session key, Access Control Manager
compute a MAC 11-29
compute HMAC using Session key 11-27
decrypt data 11-25
encrypt data 11-26
translate ACM error code 11-37
verify a MAC 11-30
verify HMAC using Session key 11-28
set master key 6-16
set_master_key 6-16
sha_hash_message 7-9
sha_hash_msg_to_bfr 7-11
SHA-1 functions
computeHMAC_SHA1 7-3
do_sha_hash_message 7-4
do_sha_hash_msg_to_bfr 7-6
hw_sha_hash_message 7-7
sha_hash_message 7-9
sha_hash_msg_to_bfr 7-11
SHA-1 hash 7-4, 7-6, 7-7, 7-9, 7-11
shares, master key 6-9
short_tag_t G-8
software, calculate SHA-1 hash 7-4
SRDI 1-1
close 10-7
concurrent access to 10-5
create 10-8, 10-9
delete 10-10
length 10-11
open 10-12
operation 10-3
resize 10-13
save 10-14
update 10-15
SRDI Manager (CCA) functions
close_cca_srdi 10-7
concurrent access protection 10-5
create_cca_srdi 10-8
create4update_cca_srdi 10-9
delete_cca_srdi 10-10
example code 10-16
get_cca_srdi_length 10-11
open_cca_srdi 10-12
opening, example 10-4
operation 10-3
overview 10-1
resize_cca_srdi 10-13
save_cca_srdi 10-14
semaphore to control concurrent access 10-5
update_cca_srdi 10-15
status, master key 6-23
stoal 13-6
stoas 13-7
store_KeyToken 9-41
structure
UDX on a workstation 1-2
UDX on the coprocessor 1-8
UDX on zSeries system 1-5
structure changes, moving 4758 to 4764 H-3
structures, data (Access Control Manager)
generic 11-3
profile 11-3
role 11-3
user information 11-4
sub-function code 1-3
subfunction codes, defining 2-5
SXT_SAMP.C listing E-1
T
test encryption of master key 9-42
token validation value, calculate 9-34
token validation value, verify 8-12
TOKEN_IS_A_LABEL 13-12
TOKEN_LABEL_CHECK 13-13
TokenMkvpMatchMasterKey 9-42
triple decrypt
master key 6-30
master key with CV 6-31
triode DES decrypt data using a master key 6-28
triple DES encrypt data under master key 6-29
triple encrypt
master key 6-32
master key with CV 6-33
triple_decrypt_under_master_key 6-30
triple_decrypt_under_master_key_with_CV 6-31
triple_encrypt_under_master_key 6-32
triple_encrypt_under_master_key_with_CV 6-33
U
UDX environment 1-1
update_cca_srdi 10-15
user authority
verify 13-3
V
ValidatePkaToken 9-43
values, reserved F-1
verb unique data (VUD) 1-3
verification pattern
current master key 8-8
old master key 8-14
specified maser key 6-21
specified master key 6-22
verify parameter block 4-41
verify rule array 4-42
verify_dSig 9-45
VerifyKeyTokenConsistency 9-44
version check, master key 8-13
version, master key 9-32, 9-33
VUD 1-3

W
word length of modulus, return 9-7
workstation host piece sample code E-1
workstation piece of a UDX 2-4
  API, defining 2-4
  completion codes, defining 2-6
  subfunction codes, defining 2-5
workstation, structure 1-2
wrapper, SHA-1 hash 7-6, 7-11

X
XCrypto functions
  coprocessor-side functions 3-1
  host-side functions 3-1
xtoal 13-6
xtoas 13-7
xtohl 13-6
xtohs 13-7

Z
z/OS host piece sample code C-1
z/OS piece of a UDX 2-10
  define the API 2-10
  define the subfunction code 2-11
zSeries system, structure 1-5
zudxexit.bal C-1