IBM 4758 PCI Cryptographic Coprocessor
Version 1: 4758-001 and 4758-013
CCA User Defined Extensions
Reference and Guide

June 9, 2000

Security Solutions & Technology
VM9A/204-3 MG81
8501 IBM Drive
Charlotte, North Carolina 28262-8563

09-JUN-00, 07:20

Note!

Before using this information and the product it supports, be sure to read the general information under Appendix C, “Notices” on page C-1.

First Edition (April, 2000)

Changes are made periodically to the information herein; before using this publication in connection with the operation of IBM systems, consult your IBM representative to be sure you have the latest edition and any Technical Newsletter.

IBM does not stock publications at the address given below; requests for IBM publications should be made to your IBM representative or to the IBM branch office that serves your location. This and other publications related to the coprocessor can be obtained in PDF format from the Library page at http://www.ibm.com/security/cryptocards.

Reader’s comments can be communicated by e-mail to George Dolan, gmdolan@us.ibm.com, or the comments can be addressed to the IBM Corporation, Department VM9A, MG81/204-3, 8501 IBM Drive, Charlotte, NC 28262-8563, U.S.A. IBM employees can send comments to TSSWS FORUM on the IBMPC conference disk. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Contents

Figures

About This Book

The *IBM 4758 PCI Cryptographic Coprocessor CCA User Defined Extensions Reference and Guide*, Version 1: 4758-001 and 4758-013 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 three classes:

1. Functions that may be called by the portion of a UDX that runs inside the coprocessor.
2. Functions that may be called by the portion of a UDX that runs outside the coprocessor.
3. Functions that are available both inside the coprocessor and on the host.

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 xi and "Custom Software Publications" on page xi.

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 should understand the use of IBM's CCA Support Program (as described in the *IBM 4758 PCI Cryptographic Coprocessor CCA Support Program Installation Manual* manual and the *IBM 4758 PCI Cryptographic Coprocessor CCA Basic Services Reference and Guide*). Familiarity with the SCC application development process (as described in the *IBM 4758 PCI 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, “SCC Functions” summarizes the secure cryptographic coprocessor (SCC) API on top of which IBM’s CCA coprocessor application modules are built. A UDX may use the SCC 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 a UDX can use to store and retrieve data in the coprocessor’s nonvolatile memory areas (flash memory and battery-backed RAM [BBRAM]).

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

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

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

Appendix C, “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.ibm.com/security/cryptocards*.

General Interest

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

- *IBM 4758 PCI Cryptographic Coprocessor General Information Manual* (version -01 or later)
- *IBM 4758 PCI Cryptographic Coprocessor Installation Manual*

CCA Support Program Publications

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

- *IBM 4758 PCI Cryptographic Coprocessor CCA Support Program Installation Manual*
- *IBM 4758 PCI Cryptographic Coprocessor CCA Basic Services Reference and Guide*

Custom Software Publications

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

- *IBM 4758 PCI Cryptographic Coprocessor Custom Software Installation Manual*
- *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference*
- *IBM 4758 PCI Cryptographic Coprocessor Interactive Code Analysis Tool (ICAT) User’s Guide*
- *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide*
- *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Overview*
- *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference*
- *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System C Runtime Library Reference*

- *AMCC S5933 PCI Controller Data Book*, available from Applied Micro Circuits Corporation, 6290 Sequence Drive, San Diego, CA 92121-4358. Phone 1-800-755-2622 or 1-619-450-9333. The manual is available online as an Adobe Acrobat** PDF file at *http://www.amcc.com/pdfs/5933db.pdf*.

Cryptography Publications

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

- "Application Support Architecture for a High-Performance, Programmable Secure Coprocessor," J. Dyer, R. Perez, S.W. Smith, and M. Lindemann, 22nd National Information Systems Security Conference, October 1999.
- "Validating a High-Performance, Programmable Secure Coprocessor," S.W. Smith, R. Perez, S.H. Weingart, and V. Austel, 22nd National Information Systems Security Conference, October 1999.
- "Building a High-Performance, Programmable Secure Coprocessor," S.W. Smith and S.H. Weingart, Research Report RC21102, IBM T.J. Watson Research Center, February 1998.
- "Using a High-Performance, Programmable Secure Coprocessor, S.W. Smith, E.R. Palmer, and S.H. Weingart, in *FC98: Proceedings of the Second International Conference on Financial Cryptography*, Anguilla, February 1998. Springer-Verlag LNCS, 1998. ISBN 3-540-64951-4
- "Smart Cards in Hostile Environments," H. Gobioff, S.W. Smith, J.D. Tygar, and B.S. Yee, *Proceedings of the Second USENIX Workshop on Electronic Commerce*, 1996.
- "Secure Coprocessing Research and Application Issues," S.W. Smith, Los Alamos Unclassified Release LA-UR-96-2805, Los Alamos National Laboratory, August 1996.
- "Secure Coprocessing in Electronic Commerce Applications," B.S. Yee and J.D. Tygar, in *Proceedings of the First USENIX Workshop on Electronic Commerce*, New York, July 1995.
- "Transaction Security Systems," D.G. Abraham, G.M. Dolan, G.P. Double, and J.V. Stevens, in *IBM Systems Journal* Vol. 30 No. 2, 1991, G321-0103.
- "Trusting Trusted Hardware: Towards a Formal Model for Programmable Secure Coprocessors," S.W. Smith and V. Austel, in *Proceedings of the Third USENIX Workshop on Electronic Commerce*, Boston, August 1998.
- "Using Secure Coprocessors," B.S. Yee (Ph.D. thesis), Computer Science Technical Report CMU-CS-94-149, Carnegie-Mellon University, May 1994.
- "Cryptography: It's Not Just for Electronic Mail Anymore," J.D. Tygar and B.S. Yee, Computer Science Technical Report, CMU-CS-93-107, Carnegie Mellon University, 1993.
- "Dyad: A System for Using Physically Secure Coprocessors," J.D. Tygar and B.S. Yee, Harvard-MIT Workshop on Protection of Intellectual Property, April 1993.
- "An Introduction to Citadel—A Secure Crypto Coprocessor for Workstations," E.R. Palmer, Research Report RC18373, IBM T.J. Watson Research Center, 1992.

- "Introduction to the Citadel Architecture: Security in Physically Exposed Environments," S.R. White, S.H. Weingart, W.C. Arnold, and E.R. Palmer, Research Report RC16672, IBM T.J. Watson Research Center, 1991.
- "An Evaluation System for the Physical Security of Computing Systems," S.H. Weingart, S.R. White, W.C. Arnold, and G.P. Double, Sixth Computer Security Applications Conference, 1990.
- "ABYSS: A Trusted Architecture for Software Protection," S.R. White and L. Comerford, IEEE Security and Privacy, Oakland 1987.
- "Physical Security for the microABYSS System," S.H. Weingart, IEEE Security and Privacy, Oakland 1987.
- *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*, Bruce Schneier, John Wiley & Sons, Inc. ISBN 0-471-12845-7 or ISBN 0-471-11709-9
- *ANSI X9.31 Public Key Cryptography Using Reversible Algorithms for the Financial Services Industry*
- *IBM Systems Journal* Volume 30 Number 2, 1991, G321-0103
- *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
- *Derived Test Requirements for FIPS PUB 140-1*, W. Havener, R. Medlock, L. Mitchell, and R. Walcott. MITRE Corporation, March 1995.
- *ISO 9796 Digital Signal Standard*
- *Internet Engineering Taskforce RFC 1321*, April 1992, MD5
- *Secure Electronic Transaction Protocol Version 1.0*, May 31, 1997

IBM Research Reports can be obtained from:

IBM T.J. Watson Research Center
Publications Office, 16-220
P.O. Box 218
Yorktown Heights, NY 10598

Back issues of the *IBM Systems Journal* and the *IBM Journal of Research and Development* may be ordered by calling (914) 945-3836.

Other IBM Cryptographic Product Publications

The following publications describe products that utilize the &ICCA. (CCA) &api. (API).

- *IBM Transaction Security System General Information Manual*, GA34-2137
- *IBM Transaction Security System Basic CCA Cryptographic Services*, SA34-2362

- *IBM Transaction Security System I/O Programming Guide*, SA34-2363
- *IBM Transaction Security System Finance Industry CCA Cryptographic Programming*, SA34-2364
- *IBM Transaction Security System Workstation Cryptographic Support Installation and I/O Guide*, GC31-4509
- *IBM 4755 Cryptographic Adapter Installation Instructions*, GC31-4503
- *IBM Transaction Security System Physical Planning Manual*, GC31-4505
- *IBM Common Cryptographic Architecture Services/400 Installation and Operators Guide, Version 2*, SC41-0102
- *IBM Common Cryptographic Architecture Services/400 Installation and Operators Guide, Version 3*, SC41-0102
- *IBM ICSF/MVS General Information*, GC23-0093
- *IBM ICSF/MVS Application Programmer's Guide*, SC23-0098

Summary of Changes

This edition of the *CCA User Defined Extensions Reference and Guide* contains product information that is current with IBM 4758 PCI Cryptographic Coprocessor Version 1: 4758-001 and 4758-013.

Chapter 1. Understanding the UDX Environment

The *UDX Development Toolkit for the IBM 4758* provides scaffold code, object modules, and header files that you can use to extend the IBM-developed Common Cryptographic Architecture (CCA) application program which employs the IBM 4758 PCI Cryptographic Coprocessor. You can use as much or as little of the CCA application function as required to meet your processing requirements.

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 4758 PCI Cryptographic Coprocessor CCA Basic Services Reference and Guide*. In particular, read chapter 1, the introductory information of chapters 2 through 6, and become aware of the material in appendixes B, C, and D.

This manual also assumes that you are familiar with the techniques for creating and testing coprocessor application programs as described in the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer's Toolkit Guide*. You may benefit from understanding the services that you can obtain from the CP/Q^{++} application program interface (API). Refer to the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference*.

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 can not 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 CP/Q^{++} 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 either battery-backed RAM (BBRAM) or (optionally) encrypted 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[1]. A common infrastructure is employed to format a verb request, transport the request to the coprocessor, dispatch the command processor, and return the reply to the host. Command processors and the top layer of CCA host code, security application program interface (SAPI), make extensive use of a set of common subroutines described in this manual.

The code that implements a user-defined extension (UDX) to CCA can be separated into two distinct pieces. One (the "host piece") runs as a DLL on the host. The other (the "coprocessor piece") is linked with a library containing IBM's CCA coprocessor application modules and downloaded to the coprocessor. The

[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.

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.

Figure 1-1 depicts the major elements of code that form the CCA implementation for the coprocessor. 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 DLLs (shared libraries) with block six actually split between a DLL and the physical device driver. An overview of these code blocks follows.

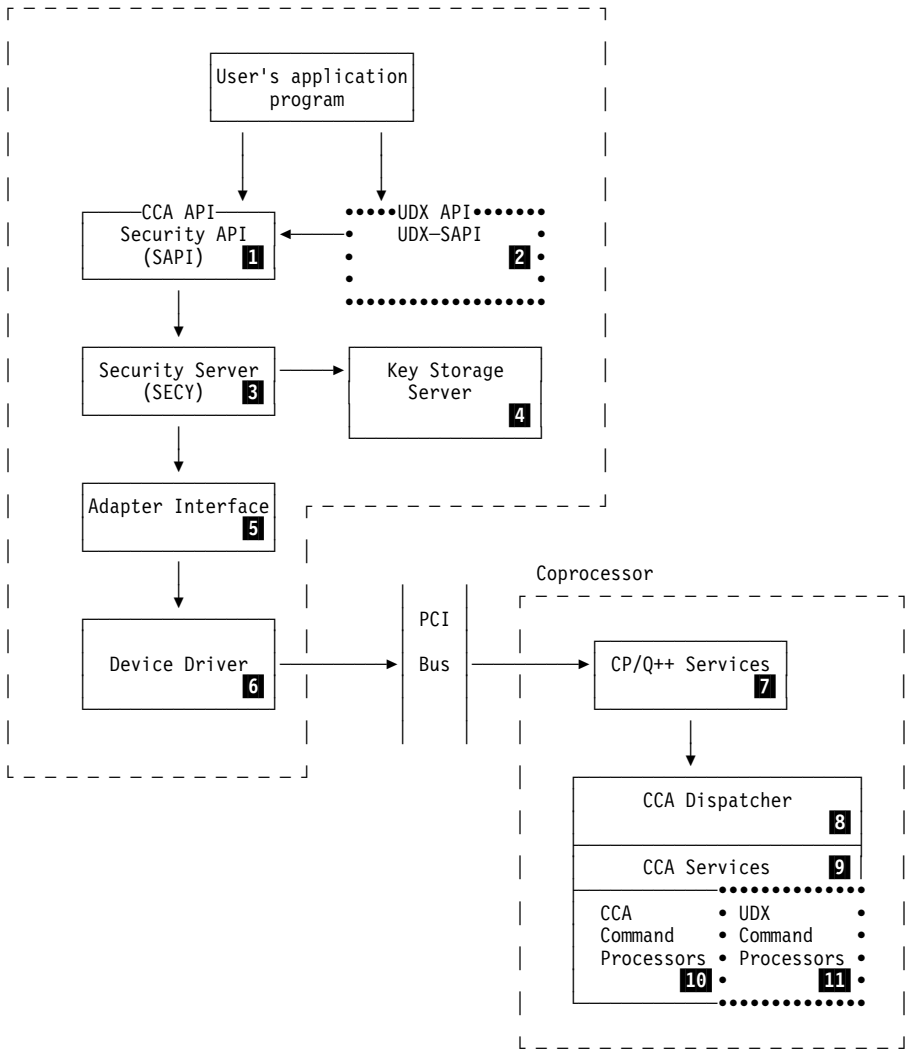


Figure 1-1. View of CCA with User-Defined Extensions

1 Security API (SAPI)

The Security API (SAPI) code, CSUxSAPI.DLL[2] , 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 DMAed to/from the coprocessor. The variable-length request and reply parameter blocks (see Figure 1-2 on page 1-7) 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 or 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 2 . See Chapter 4, "Communications Functions" on page 4-1.

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 little-endian format (Intel byte-reversed format).

Because the CCA SAPI code is compiled for both personal computer and IBM RS/6000 systems, C macros are used to ensure that the integers exchanged with the coprocessor are in little endian format. Note, however, that most CCA data structures, such as 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.

2 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)

---

[2] Typical CCA host code file names begin with CSUx where the "x" is "N" for Windows NT, "E" for OS/2, and "F" for AIX.

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. See Appendix A, "UDX Sample Code - Host Piece."

3 Security Server (SECY)

The Security Server (SECY), CSUxSECY.DLL, 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 1-9) triggers the security server to process a key block.

4 Key Storage Server

The key storage server, CSUxDIR.DLL, 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.

5 Adapter Interface

The adapter interface, CSUxCALL.DLL, 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 DMAed to the coprocessor. A few requests (for example, data ciphering and MACing requests) also require that data be scheduled for DMA interchange with 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 DMA transfers from/to the identified fields. The adapter interface layer creates the control blocks and issues the I/O request to the device driver DLL.

6 Device Driver and Access Layer

The device driver code is split between a DLL and a physical device driver. The API and function of the device driver is explained in the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference*.

The device driver and CP/Q^{++} work together to ensure that the reply to a request is routed back to the source process and thread that initiated the associated request.

7 CP/Q++ Services

CP/Q++ becomes aware of an application in coprocessor segment three following a reset sequence. The application's entry point is called and CCA registers itself with CP/Q++.

When CP/Q++ 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, which is passed on by CP/Q++, so that the CCA Dispatcher can request CP/Q++ to obtain the CPRB and request parameter block.

Other CP/Q++ services for DES, RSA, DSA, random number, date and time, storage of data in BBRAM and flash memory, and communication with external functions as described in the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* and *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System C Runtime Library Reference* are available 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 CP/Q++ directly.

8 CCA Dispatcher

When CP/Q++ 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 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 CCA command processor entry point. If a match is not found, the dispatcher checks the UDX 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 CP/Q++ to DMA the reply CPRB, and (optionally) the reply parameter block, back to the host.

In the current CCA coprocessor implementation, processing is performed on three multi-tasked threads of execution:

1. The CCA dispatcher
2. The RSA key generation command processor, because this is a long-running function
3. All other command processors

A future implementation might introduce additional threads of execution. Therefore, as you write code for the CCA application, you must consider the impact of multi-tasking. All CP/Q++ functions are thread-safe.

9 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 BBRAM and 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. See Chapter 11, “Access Control Manager Functions” on page 11-1. The sample SAPI code (Appendix A, “UDX Sample Code - Host Piece” on page A-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 all of the CCA coprocessor code and much of CP/Q++ operates at “protection ring 3” in the Intel 80x86 architecture. Therefore, all of this 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.

10 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 CP/Q++ API. In general, a command processor will perform the following steps. See Appendix B, “UDX Sample Code - Coprocessor Piece.”

- 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.
- 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 8*x*+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. You can use the TOKEN_LABEL_CHECK() routine to determine if a key identifier is a key label.
- 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.

11 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 4758 PCI 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 1-2 shows the request and reply parameter block format.

Note: Be careful that the SAPI code processes the lengths in little-endian format (“Intel byte-reversed order”).

Figure 1-2. Request and Reply Parameter Block Formats

Field:	Sub-function Code	Rule Array Length	Rule Array Data	Verb Unique Data Length	Verb Unique Data	Key Block Fields Length	Key Block Fields
Size:	2	2	X	2	Y	2	Z
Offset:	0	2	4	4+X	6+X	6+X+Y	8+X+Y

Field Name	Description
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).
Rule Array Data	Zero or more 8-byte character arrays (not NULL-terminated). If no rule-array elements are specified, this field is empty (0-length).
Verb Unique Data Length	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.
Verb Unique Data	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).
Key Block Fields Length	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.
Key Block Fields	Optional key block(s) exchanged between the host and coprocessor code. If no key tokens or key labels are specified, this field should be empty (0-length).

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 routine `BuildParmBlock`. 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 parmeter 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 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 A, “UDX Sample Code - Host Piece” on page A-1.

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. 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 four bytes long. In order for the device driver to manipulate the buffers efficiently, they should be aligned on 4-byte boundaries. Access to these buffers is managed by the coprocessor application using the sccGetBufferData and sccPutBufferData functions, respectively, using the defined constants CPRB_REQ_DATA or CPRB_REPLY_DATA as buffer indices. Refer to the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference* for more details on using these two functions.

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. 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 little-endian format followed by a USHORT *Flags* field in little-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:

Flags indicating the type of key (one required)	
PKA96_TYPE	Indicates that the key token or label is an PKA token.
DES96_TYPE	Indicates that the key token or label is a DES key token.
Flags indicating the action to be taken (one required)	
ACTION_READ	Request SECY to retrieve the key token from the storage file. This header must be followed by a key label.
ACTION_WRITE	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.
ACTION_NOOP	Requests no action from SECY.

If a DES key label were being passed with the intent that SECY obtain a key token for forwarding to the coprocessor, the Length and Flags fields would be set as follows:

```
KeyHeader.Flags    = htoas(DES96_TYPE | ACTION_READ);
KeyHeader.Length  = htoas(KEY_HDR_LEN + sizeof(KeyLabel));
```

This would cause SECY to locate the label which follows this header in the key block within the DES Key Storage File and replace the key label in the key block with the correct key token, for use by the coprocessor.

If an RSA key token were being passed, the Length and Flags fields would be set in this way:

```
KeyHeader.Flags   = htoas (PKA96_TYPE | ACTION_NOOP);
KeyHeader.Length = htoas(KEY_HDR_LEN + xtohs(KeyToken.KeyLength));
```

Notice that the KeyLength field of a PKA key token is stored in big-endian format.

Structuring the Key Block

If a key in the key storage file is to be read from key storage and changed in the command processor (for example, re-enciphering under the current master key) two copies of the key label must be passed from the host to the coprocessor, one with the ACTION_READ flag set, the other with the ACTION_NOOP flag set. This ensures that the coprocessor will receive both a key token to work with, and a key label with which to write the token to the key storage file when done. Upon return, the key block is built with one header, the Flags field set to ACTION_WRITE, followed by the Key Label and the Key Token. See the sample function in Appendix A and Appendix B for an example.

If the key is to be read from the SECY server, but not changed (for example, for an ENCIPHER service) the key label may be passed alone, with the Flags field set to ACTION_READ. On return, the key block is empty and the length is set to two.

Key Labels and key tokens are further described in the *IBM 4758 PCI Cryptographic Coprocessor CCA Basic Services Reference and Guide*.

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, and that you have knowledge of the CP/Q++ API as explained in the other Toolkit publications (see "Related Publications" on page xi).

Files You Use in Building a UDX

A developer must create the following files (or modify the samples that are provided with the UDX Developer's Toolkit) to produce a UDX:

- A header file (for example, ***csueextn.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 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 2-3.
- A header file (for example, ***cxt_cmds.h***) that defines the interface between the host piece of the UDX and the coprocessor piece of the UDX. This header file is `#included` by both pieces and should define UDX subfunction codes, a UDX command processor for each subfunction, and the access control points and completion codes used by the UDX. The sample provided with the UDX Developer's Toolkit includes comments that indicate the range of acceptable values for each of these elements.
- One or more C source files (for example, ***sxt_samp.c***) that implement the host piece of the UDX. The sample provided with the UDX Developer's Toolkit is a skeleton that exports a single function to the user's application. The 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.
- One or more C source files (for example, ***cxt_cmds.c***) that implement the coprocessor piece of the UDX. The sample provided with the UDX Developer's Toolkit is a skeleton that 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.

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

- A makefile that builds the DLL that implements the host piece of the UDX and the import library that describes the entry points the DLL exports. The UDX Developer's Toolkit provides two such files: ***csueextn.mak*** (which creates ***csueextn.dll*** and ***csueextn.lib***) for use on OS/2 and ***csunextn.mak*** (which creates ***csunextn.dll*** and ***csunextn.lib***) for use on Windows NT. These makefiles are customizable for use with either Microsoft Visual C++ or IBM Visual Age C++.

- A makefile (for example, ***camextn.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 may be customized for use with either Microsoft Visual C++ or IBM Visual Age C++.
- A linker definition file for the DLL that implements the host piece of the UDX. This file specifies the names of the entry points exported by the UDX and lists the CCA functions the host piece of the UDX invokes. The UDX Developer's Toolkit provides two such files: ***csueextn.def*** for use on OS/2 and ***csunextn.def*** for use on Windows NT.
- A resource definition file (for example, ***csunextn.rc***) that supplies the version information that appears in the properties of the DLL that implements the host piece of the UDX on Windows NT. No resource definition file is needed to build the host piece of a UDX on OS/2.

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. This library is linked with the user's application. The UDX Developer's Toolkit makefiles generate the appropriate library: ***csueextn.mak*** creates ***csueextn.lib*** on OS/2 and ***csunextn.mak*** creates ***csunextn.lib*** on Windows NT.
- 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 Developer's Toolkit provides two libraries: ***csuelib.lib*** for use when building a UDX on OS/2 and ***csunlib.lib*** for use when building a UDX on Windows NT.

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.

Host Piece of a UDX

The host piece of a UDX is a dynamic link library (DLL) that converts requests for service from the user's application into one or more calls to the standard CCA host API module (*csuesapi.dll* on OS/2, *csunsapi.dll* on Windows NT). 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.

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, *csueextn.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 *csueincl.h* (OS/2 versions) and *csunincl.h* (Windows NT versions). Both header files are part of the UDX Developer's Toolkit. Figure 2-1 illustrates a representative prototype.

```
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 Function Prototype

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.

Refer to the *IBM 4758 PCI 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 zero or more command processors. 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 2-4 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 Developer's Toolkit.

```
/*****************************************************************
** 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. Because of the INTEL architecture the hex
** values are reversed.
**
** 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, W0 - W9
**       XA - XZ, X0 - X9
**       YA - YZ, Y0 - Y9
*************************/
#define  CCAXFNC1_ID       0x4158   /* 'XA' - Sample CCA extension 1 */
#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 4758 PCI 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 contains an example of such a definition.

```
/*****************************************************************
** 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
*************************/
#define CXT_INFO_xxxxxx         0x00005000L   /* 00/20480  sample     */
#define CXT_WARN_xxxxxx         0x00045001L   /* 04/20481  sample     */
#define CXT_ERR8_xxxxxx         0x00085002L   /* 08/20482  sample     */
#define CXT_ERR12_xxxxxx        0x000C5003L   /* 12/20483  sample     */
#define CXT_ERR16_xxxxxx        0x00105004L   /* 16/20484  sample     */
```

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 A, “UDX Sample Code - Host Piece” on page A-1 for a sample (*sxt_samp.c*). 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. This approach makes it much easier to port the host piece to different platforms if the need arises.

5. **Export the UDX API entry points**.

   The entry points in the DLL that implements the host piece of the UDX must be exported so that the user’s application can invoke them. Sample link definition files for OS/2 (*csueextn.def*) and for Windows NT (*csunextn.def*) are included in the UDX Developer’s Toolkit. An EXPORT statement should be added for each function the UDX exports to the user’s application.

6. **Build the UDX DLL and LIB files**.

   The UDX Developer’s Toolkit includes sample makefiles for OS/2 (*csueextn.mak*) and for Windows NT (*csunextn.mak*). Statements should be added to compile the source files that contain the host piece of the UDX and create the UDX DLL and 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, Developing and Debugging an SCC Application” of the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide*.

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 (*csuelib.lib* on OS/2, *csunlib.lib* on Windows NT) 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 CP/Q^{++} 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 both pieces of the UDX. The prototype must have the same parameters and return type as the example shown in Figure 2-4 on page 2-6.

```
/*****************************************************************
** 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.  It is required
**                as input for some scc.... library calls.
**    roleID    - (input) The user's role identifier.  It is required
**                as input when checking the requestor's access
**                authority to this function.
*************************/
void ccax_fcn_1(
         CPRB_structure *pCprbIn,
         CPRB_structure *pCprbOut,
         unsigned long   RequestId,
         role_id_t       roleID);
```

Figure 2-4. Example UDX Command Processor Prototype

On entry to a command processor:

`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 `pCprb` argument the host piece of the UDX passed to the call to **CSNC_SP_SCSRFBSS** that caused the command processor to gain control.

`pCprbOut` contains the address of a buffer large enough to hold a CPRB header and the result of the operation.

`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.[1] A command processor that invokes basic coprocessor operating system functions may need to pass this handle as an argument to those functions.

`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, *cxt_cmds.h*) that is `#include`d by both pieces of the UDX. Figure 2-5 on page 2-7 contains an example of such a definition.

[1] `RequestId` is the value returned in the `pRequestHeader->RequestID` output from the call to sccGetNextHeader that received the message. Refer to the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference* for details.

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 12-2 for details.

These values must also be added to the *csuap.def* file in the *cnm* subdirectory of the CCA (for example, Program Files\IBM\4758\cnm). The cnm utility uses this file to enable editing of roles. Refer to the *IBM 4758 PCI Cryptographic Coprocessor CCA Support Program Installation Manual* manual for more information.

```
/*****************************************************************
** 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.      */
```

Figure 2-5. Example UDX Access Control Points

3. **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 `ccax_cp_list` array, which must be defined in a program file (for example, *cxt_cmds.c*) that is compiled with both pieces of the UDX. Each entry contains a subfunction code and the name of the corresponding command processor.

   The `ccax_cp_list_size` variable must be initialized to the number of entries in the array.

   Figure 2-6 on page 2-8 contains an example of the requisite definitions.

```
/************************************************************
** 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 } };


/*********************************
** Declare a variable which holds the number of CCA extension
** command processors defined in the ccax_cp_list table above.
*********************************/

ULONG ccax_cp_list_size = (sizeof(ccax_cp_list)/sizeof(CCAX_CP_DEF));
```

Figure 2-6. Example UDX Command Decoding Array Definition

4. **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 (*cxt_samp.c*) appears in Appendix B, “UDX Sample Code - Coprocessor Piece” on page B-1. It can be used as a skeleton and customized to meet the requirements of most UDXs.

5. **Build the UDX coprocessor executable**.

   The UDX Developer’s Toolkit includes a sample makefile (*camextn.mak*) that works on OS/2 and on Windows NT. Statements should be added to compile the source files that contain the coprocessor piece of the UDX. The makefile generates two versions of the UDX binary, one that contains debug information and one that does not. The version without debug information should be incorporated into a read-only disk image using the SCCRODSK utility and downloaded to the coprocessor using the DRUID utility. The version with debug information remains on the host and is used by the ICAT debugger to support source-level debug of the UDX. See the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide* for more information.

Note: To securely load your application into a coprocessor requires that the application be signed with keys certified by Development in IBM Charlotte. See the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide* for an explanation of the process to obtain certified keys and to sign your application.

Chapter 3. SCC Functions

The CCA API is built on top of the secure cryptographic coprocessor (SCC) 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. SCC API functions can also be invoked by a UDX. The SCC 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 SCC API. A more detailed description may be found in the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference*.

Host-Side SCC API Functions

The host-side portion of the SCC API (host API) allows an application running on the host to exchange information with an application running on a coprocessor. The host API provides a uniform interface for applications running on AIX, OS/2, and Windows NT.

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.

Coprocessor-Side SCC API Functions

The coprocessor API includes functions in the following categories:

Functions Category	Description
Communications	Allows a coprocessor application to interact with a host application and obtain permission to request services from the coprocessor device managers.
Hash	Allows a coprocessor application to compute a condensed representation of a block of data using various standard hash algorithms.
DES	Allows a coprocessor application to request services from the Data Encryption Standard (DES) Manager, which uses the coprocessor's DES chip to support DES operations with key lengths of 40, 56, 112, or 168 bits and the Commercial Data Masking Facility (CDMF) algorithm.[1]
Public Key Algorithm	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.
Large Integer Modular Math	Allows a coprocessor application to direct the PKA Manager to perform specific operations on large integers.

Functions Category	Description
Random Number Generator	Allows a coprocessor application to request services from the Random Number Generator (RNG) Manager, which uses a hardware noise source to deliver random bits that meet the standards described in FIPS Publication 140-1, section 4.11.
Nonvolatile Memory	Allows a coprocessor application to request services from the Program Proprietary Data (PPD) Manager, which controls the coprocessor's nonvolatile memory areas (flash memory and battery-backed RAM [BBRAM]).
Coprocessor Configuration	Configures certain processor features or return information about the coprocessor.

[1] CDMF is a DES-based data confidentiality algorithm with a key strength equivalent to 40 bits. In general, it is used when import or export regulations prohibit the use of longer keys.

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.

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

Summary of Functions

Request and reply processing includes the following functions.

BuildParmBlock	Build a parameter block.
Cas_proc_retc	Prioritizes a return code in the reply CPRB.
CSNC_SP_SCSRFBSS	Send a request to the coprocessor.
CSUC_BULDCPRB	Construct a well-formed CPRB block.
CSUC_PROCRETC	Prioritize a return code.
FindFirstDataBlock	Search for the first data block.
FindNextDataBlock	Search for the next data block.
find_first_key_block	Search for the first key block.
find_next_key_block	Search for the next key block.
InitCprbParmPointers	Initialize CPRB parameter pointers.
keyword_in_rule_array	Search for a keyword in the rule array.
parm_block_valid	Examine and verify a parameter block.
rule_check	Verify a rule array.
saf_process_key_label	Process a key label.

BuildParmBlock - Build a Parameter Block

Note: This function is available on both the 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 an unsigned char 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
  ... )
```

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.

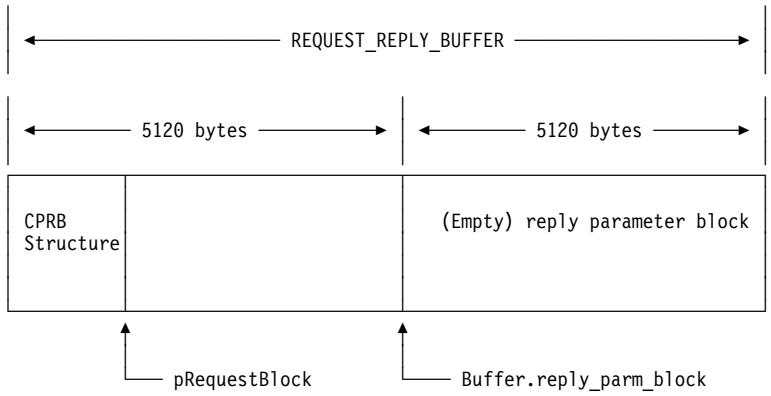
`Data`*i*`_length` is the length of the *i*th. item, in bytes.

`Data`*i* is a pointer to the *i*th data item to be added.

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

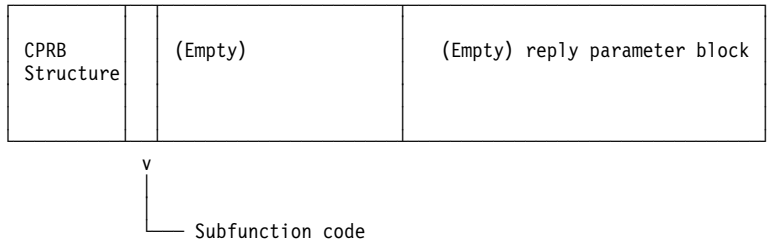
If 2 or more items of verb unique data are to be added, each item should be preceded by a short field containing the length of the individual item +2. This will allow the function FindNextDataBlock to parse the result.

```
BlockLength = 0;
pCprb = (CPRB *)&(Buffer.request_parm_buffer[0]);
pRequestBlock = &(Buffer.request_parm_buffer[0]) +sizeof(CPRB_structur
```



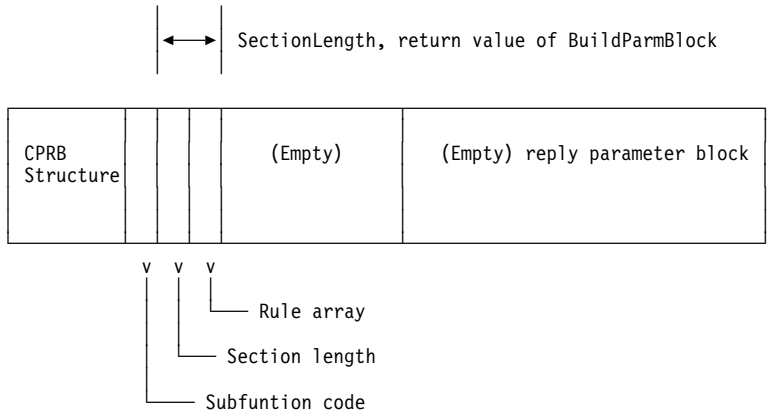
Step one: add the subfunction code

```
BlockLength +=2;
*((USHORT *) pReqBlk) = htoas ( CCAXFNC1_ID ) ;
```



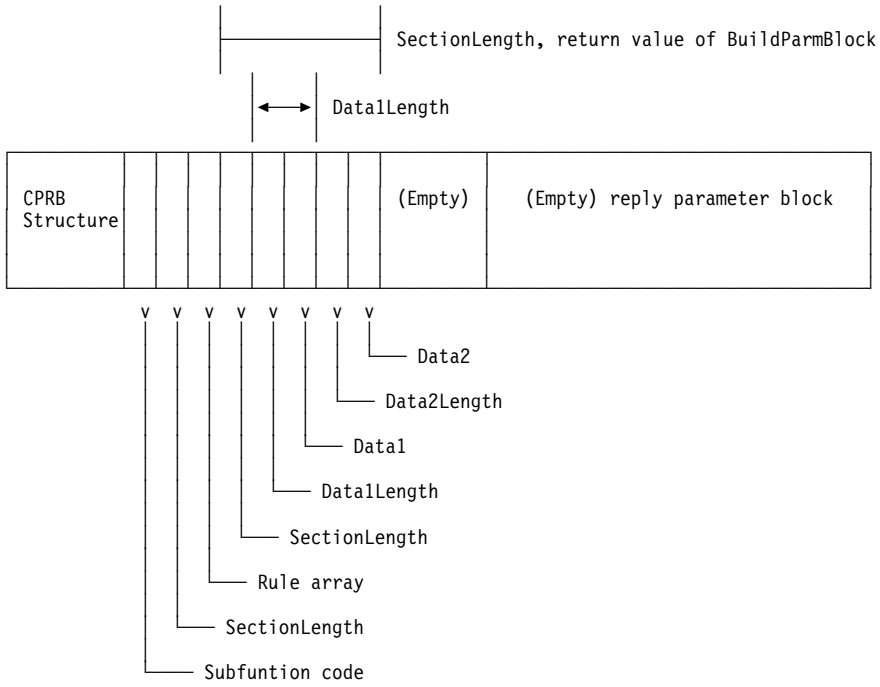
Step two: add the rule array

```
BlockLength += BuildParmBlock(pRequestBlock+BlockLength,
                        1,   /* adding 1 rule array */
                        (*pRuleArrayCount) *8, /* length of rule array */
                        pRuleArray);
```



Step three: add the verb unique data

```
Data1Length = Data1Size + sizeof(short);
Data2Length = Data2Size + sizeof(short);
BlockLength += BuildParmBlock(pRequestBlock + BlockLength,
                        4,  /* adding 2 data items, plus their lengths */
sizeof(short), &Data2Length, /* length of 1st item, including this field */
Data1Size, pData1,
sizeof(short), &Data2Length, /* length of 2nd item, including this field */
Data2Size, pData2);
```



Output

On successful exit from this function:

`BuildParmBlock` returns the total length of the block built by the function. The buffer at `pBuffer` contains the parameter block.

Return 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. No more than one rule array is used per call, although up to 5 separate rules can be included in the array.

For more information about key block structures, see “Key Blocks” on page 1-9.

See Appendix A, UDX Sample Code - Host Piece for sample code which includes key label to token translation and 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 on 1-byte boundaries. 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 compiler aligns the structure on one-byte boundaries. This can be done by adding a “#pragma pack(1)” directive in the include file before the structure is defined, or by compiling with the “/Zp1” (for MSVC++) or “Sp1” (for VACPP) directives in the makefile.

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

```
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 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 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

```
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.

---

[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.

Return 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.

CSUC_BULDCPRB - Build CPRB

Note: This function is available on the host.

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

Function Prototype

```
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 aligned on a 4-byte boundary, and must be a multiple of 4 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 aligned on a 4-byte boundary, and must be a multiple of 4 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. It must begin on a 4-byte boundary.
- `reply_parm_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. It must begin on a 4-byte boundary.

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 begin on 4-byte boundaries.

Return Codes

This function has no return codes.

CSUC_PROCRETC - Prioritize Return Code

Note: This function is available on the 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
(
  ADDRESS4_PTR   return_code_ptr,
  ADDRESS4_PTR   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 Codes

This function has no return codes.

FindFirstDataBlock - Search for Address of First Data Block

Note: This function is available on both the 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

```
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` is a location where the function stores the address of the first data block in the Verb Unique Data.

Return Codes

This function has no return codes.

FindNextDataBlock - Search for Address of Next Data Block

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

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

Function Prototype

```
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` is a location where the function stores 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 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 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` is a location which receives 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 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 host and the coprocessor.

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

```
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` is a pointer to a location where the function puts 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 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 Codes

This function has no return codes.

keyword_in_rule_array - Search for Rule Array Keyword

Note: This function is available on both the 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 4-18 for information about *parm_block_valid*.

Function Prototype

```
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 Codes

This function has no return codes.

parm_block_valid - Examine and Verify a Parameter Block

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

parm_block_valid examines the parameter block associated with a specified connectivity programming 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

```
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 Codes

This function has no return codes.

rule_check - Verify Rule Array

Note: This function is available on 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

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

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 *ith* integer is the map value of the keyword from the *ith* set which is present in the rule array, or INVALID_RULE if there is no keyword from that set.

Return 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.

```
static RULE_MAP RuleMap[3] = { { "FIRST   ", 1, 1 } ,
                               { "MIDDLE  ", 1, 2 } ,
                               { "LAST    ", 1, 3 } };
```

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.

```
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.

For the following set of rules, where either COPY or REVERSE is required, and OFFSET is optional:

```
int RuleValue[2];       /* to hold the rule values only 2 */
USHORT RuleMapCount = 3;
static RULE_MAP RuleMap[] = { {"COPY    ", 1 , COPY },
                              {"REVERSE ", 1 , REVERSE },
                              {"OFFSET  ", 2 , OFFSET  },};
/*****************************
error checking, etc.
*****************************/
/*************************************************
** Compare for valid rule array values.
*************************************************/

RuleValue[0] = INVALID_RULE;         /* initialize */
RuleValue[1] = INVALID_RULE;

if ( rule_check ((RULE_BLOCK *) &pReqBlk->rule_array_length,
                  RuleMapCount,
                  &RuleMap[0], &RuleValue[0], &ReturnMsg)
     == false)
{
            Cas_proc_retc(pCprbOut, ReturnMsg);
            return;
}
/*********************************************
verb unique data and keys, if needed
*********************************************/

if (RuleValue[1] == OFFSET)
{
      /* Do what OFFSET requires */
} else
{
      /* Do default things */
}

if (RuleValue[0] == COPY)
{
      /* copy the data */
} else if (RuleValue[0] = REVERSE)
{
      /* reverse the data */
}
/*Return needed data */
```

saf_process_key_label - Process Key Label

Note: This function is available on the host.

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

```
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 Codes

This function has no return codes.

Chapter 5. Function Control Vector Management Functions

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 using the *csuncnm* utility on Windows NT, *csuecnm* utility on OS/2, and the *csufcnm* utility on AIX. Refer to *IBM 4758 PCI Cryptographic Coprocessor CCA Support Program Installation Manual* for more detailed information.

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

Header Files for Function Control Vector Management Functions

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

```
#include "cmncryt2.h"            /* Crypto ESSS definitions        */
#include "cam_fcv.h"             /* Function control vector def.   */
```

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

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

Function Prototype

```
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

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

```
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 on page 5-4 for a list of possible values.

`FunctionBitSelect` is the bit to be checked in the specified Function Control Vector byte. See Figure 5-1 on page 5-4 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

Function Byte Name	Function Bit Name	Description
CCA_BASE_FUNCTION_BYTE		Byte index of the CCA base services bits.
	FCV_CCA_BASE	Base CCA services-enabled bit.
DES_FUNCTION_BYTE		Byte index of the DES-enabled bits.
	FCV_CDMF_DES	CDMF function-enabled bit
	FCV_56_BIT_DES	56-bit DES enabled bit
	FCV_TRIPLE_DES	Triple DES enabled bit
SET_FUNCTION_BYTE		Byte index of the bits that are SET™ enabled
	FCV_SET_SERVICES	SET services enabled bits

Return Codes

This function has no return codes.

Examples

To determine whether SET functions for encoding and decoding are enabled in the coprocessor:

```
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:

```
if ( IsFunctionEnabled(DES_FUNCTION_BYTE, FCV_56_BIT_DES) )
{
    /* use 56-bit DES encryption */
}
```

Chapter 6. CCA Master Key Manager Functions

Header Files for Master Key Manager Functions

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

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

The CCA Master Key Manager provides access to the CCA master key registers, 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.

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.

CCA supports 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 is entered in the form of one or more *key parts*, which are combined to form the final key.

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.

  This value is calculated using SHA-1.

- 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 is named `MSTRKEYS`.

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, `MSTRKEYS`. 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 two 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, this must be set to MK_SET_DEFAULT.
- 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 following functions are summarized in this chapter.

Functions to Set and Manage the Master Key Values

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.
combine_mk_parts	Combines an additional master key part into the value already in the New Master Key register.
generate_mk_shares	Splits a 24-byte master key into shares.
generate_random_mk	Generates a random 24-byte master key.
init_master_keys	Creates and initializes the master key SRDI.
load_first_mk_part	Loads the first part of a multi-part master key into the new master key register.
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.
reinit_master_keys	Deletes all master key data and then creates and initializes the master key SRDI.
set_master_key	Activates the master key.

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 *Empty* for NMK, or *invalid* for CMK or OMK.

Three separate calls 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:

`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.

Output

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

`clear_master_keys` clears a specified master key register.

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.

combine_mk_parts - Combine Master Key Parts

combine_mk_parts combines an additional master key part into the value already in the NMK register. 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` 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

```
long combine_mk_parts(TRIPLE_LENGTH_KEY *key_part,
                      boolean            final_part);
```

Input

On entry to this routine:

`key_part` is a 24-byte cleartext key part, which is combined into the value in the NMK register.

`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 `load_first_mk_part` and zero or more calls to `combine_mk_parts`.

Output

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

`combine_mk_parts` combines an additional master key part 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.

generate_mk_shares - Generate Master Key Shares

generate_mk_shares splits a 24-byte master key into shares. 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

```
long generate_mk_shares(mk_src_t KeySource,
                        UCHAR    mShareKey,
                        UCHAR    nShareGen,
                        UCHAR   *pShares[]);
```

Input

On entry to this routine:

`KeySource` is a value which specifies the source for the master key value that is split. The possible values are:

Source	Description
Randomly generated value (src_random)	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.
New Master Key (NMK) Register (src_nmk)	The value in the NMK register is split into shares. An error is returned if the NMK state is not *Full*.
Current Master Key (CMK) Register (src_cmk)	The value in the CMK register is split into shares. An error is returned if the register does not contain a valid value.
Old Master Key (OMK) Register (src_omk)	The value in the OMK register is split into shares. An error is returned if the register does not contain a valid value.

`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 *load_mk_from_shares* 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[]` is a pointer to an array where the function 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.

generate_random_mk - Generate Random Master Key

generate_random_mk generates a random 24-byte master key, and stores the value in the NMK register.

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

```
long generate_random_mk(void);
```

Input

This function has no input.

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_INCORRECT_STATE	The master key is in the incorrect state. If the verification patterns match with the previous, current, or old master keys, another random key is generated.
mk_SAVE_ERROR	Unable to save the master key to flash memory.

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.

load_first_mk_part - Load First Master Key Part

load_first_mk_part loads the first part of a multi-part cleartext master key into the new master key (NMK) register.

Note: The purpose behind requiring a `load_first_mk_part` 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

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

Input

On entry to this routine:

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

The `check_and_adjust` parity routines make sure that the input key part has odd parity. If it does not, the parity is adjusted.

Output

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

`load_first_mk_part` loads the first part of a multi-part cleartext master key 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.

load_mk_from_shares - Load Master Key Shares

load_mk_from_shares loads a master key into the New Master Key register, reconstructing the key from *m* supplied shares, which were originally produced by the *generate_mk_shares* function.

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 *set_master_key* function to make it the current master key.

Function Prototype

```
long load_mk_from_shares(UCHAR  mShareKey,
                         UCHAR  nShareGen,
                         UCHAR *pShares[]);
```

Input

On entry to this routine:

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

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

`*pShares[]` is a pointer to an array which contains 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.

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.

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

```
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.

set_master_key - Set Master Key

set_master_key activates the master key which has been accumulated in the NMK register. 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

```
long set_master_key(void);
```

Input

This function has no input.

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.

Functions to Check Master Key Values and Status

Summary of Functions

compute_mk_verification_pattern	Computes a key verification pattern.
get_master_key_status	Returns the status of the master key register.
get_mk_verification_pattern	Returns the 20-byte master key verification pattern for a specified master key.

compute_mk_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. The verification pattern is an 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'.

Function Prototype

```
long compute_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 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 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.

  The `type_mks` variable should be set to SYM_MK, as this function calculates the verification pattern for a master key for symmetric keys.

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_SEM_CLAIM_FAILED	Unable to access the SRDI Manager.
mk_VP_CALCULATE_FAIL	SHA calculation error.

get_master_key_status - Get Master Key Status

get_master_key_status returns the status of the three master key registers. The results indicate whether the register holds a valid value, and whether a value in the NMK register is complete.

Function Prototype

```
long get_master_key_status(mk_status_var *mk_status);
```

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` should be set to MK_SET_DEFAULT
- `mk_register` must be set to new_mk or current_mk

`mk_status` is a pointer to a one-byte variable.

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	
Bit 0 (LSB)	NMK register is empty.
Bit 1	NMK register is partially full.
Bit 2	NMK register is full.
Bit 3	CMK register holds a valid value.
Bit 4	OMK register holds a valid value.
Bits 5-7	Reserved, set to 0.

`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.

get_mk_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

```
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 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.

  The `type_mks` variable should be set to SYM_MK, as this function calculates the verification patter for a master key for symmetric keys.

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.

Functions to Encrypt and Decrypt Using the Master Key

Summary of Functions

ede3_triple_decrypt_under_master_key	Triple decrypts multiple 8-byte data strings using EDE3 triple DES.
ede3_triple_encrypt_under_master_key	Triple encrypts multiple 8-byte data strings using EDE3 triple DES.
triple_decrypt_under_master_key	Triple-DES decrypts an 8-byte block of data.
triple_decrypt_under_master_key_with_CV	Triple-DES decrypts an 8-byte block of data using a control vector.
triple_encrypt_under_master_key	Triple-DES encrypts an 8-byte block of data.
triple_encrypt_under_master_key_with_CV	Triple-DES encrypts an 8-byte block of data using a control vector.

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 cleared.
- `type_mks` is 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.

`cleartext` is a pointer to a buffer large enough to store the `ciphertext`. 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 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_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 cleared.
- `type_mks` is 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.

`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 `cleartext`. 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 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_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

```
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` is 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.

`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 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_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

```
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 cleared.
- `type_mks` is 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.

`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 cleared.
- `type_mks` is 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.

`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

```
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 cleared.
- `type_mks` is 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.

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

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:

sha_hash_message - SHA-1 Hash with Chaining

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,
                        dbl_ulong   *pBitCount,
                        sha_context *pContext,
                        owh_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 7-3 for details.

`middle` The input data constitutes an additional portion of a block of data to be hashed. See “Chained Operations” on page 7-3 for details.

`final` The input data constitutes the final portion of a block of data to be hashed. See “Chained Operations” on page 7-3 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`. `*pBitCount` is interpreted as a 64-bit integer. `pBitCount->upper` contains the most-significant 32 bits of `*pBitCount` and `pBitCount->lower` contains the least-significant 32 bits of `*pBitCount`.

Note: Both fields are regular 32-bit integers (that is, C `unsigned longs`) that are stored in the native byte order of the processor on which the code is running.

For example, `pBitCount->lower` and `pBitCount->upper` are stored in little-endian order on the coprocessor.

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 7-3 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.

Notes

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 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.

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*.

Examples

To compute the SHA-1 hash of a contiguous block of 150 bytes of text at pBlock:

```
BitCount = ((dbl_ulong) 150)*8;
memset ((UCHAR *)pContext, 0x00, sizeof(sha_context) );

sha_hash_message(pBlock, &Hash, &BitCount, pContext, only);
```

To compute the SHA-1 hash of only the name fields of the following structure:

```
struct emp_data{
      char ID[10];
      double salary;
      char name[64];
}employee[MAX_EMP];

BitCount = (dbl_ulong)512;
memclr ((UCHAR *)&Context, 0x00, 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-1;  i++)
{
            /* it is important that the value in BitCount is divisible by 512 */
       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 */
```

sha_hash_msg_to_bfr - SHA-1 Hash

sha_hash_msg_to_bfr is a wrapper for sha_hash_message that simplifies the interface when chained operations (see page 7-3) are not necessary.

Function Prototype

```
void sha_hash_msg_to_bfr(UCHAR       *pBlock,
                         UCHAR       *pHash,
                         dbl_ulong   *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`. `*pBitCount` is interpreted as a 64-bit integer. `pBitCount->upper` contains the most-significant 32 bits of `*pBitCount` and `pBitCount->lower` contains the least-significant 32 bits of `*pBitCount`.

Note: Both fields are regular 32-bit integers (that is, C `unsigned long`s) that are stored in the native byte order of the processor on which the code is running.

For example, `pBitCount->lower` and `pBitCount->upper` are stored in little-endian order on the coprocessor.

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

```
{
  sha_context Context;
  memset(&Context,0,sizeof(Context));
  sha_hash_message(pBlock,pHash,pBitCount,&Context,only);
}
```

Return Codes

This function has no return codes.

Chapter 8. DES Utility Functions

This chapter describes functions to assist in the use of key tokens and other cryptographic structures.

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.

```
#include "cmncryt2.h"           /* T2 structures, constants, functions */
#include "castyped.h"           /* Adapter typedefs and structures     */
#include "cassub.h"             /* DES 96 function prototypes          */
#include "casfunct.h"
```

Summary of Functions

DES utility routines includes the following functions.

cas_adjust_parity	Adjusts the parity of a DES key token.
cas_build_default_cv	Builds a default control vector.
cas_build_default_token	Builds a default DES key token.
cas_current_mkvp	Returns the current master key verification pattern.
cas_des_key_token_check	Verifies the integrity of a DES key token.
cas_get_key_type	Returns the type of DES key token.
cas_key_length	Returns the length of a DES key.
cas_key_tokentvv_check	Verifies a DES key token validation value.
cas_master_key_check	Performs a master key version check.
cas_old_mkvp	Returns the old master key verification pattern.
cas_parity_odd	Determines whether a DES key has odd parity.
RecoverDesDataKey	Recovers the cleartext form of a DES importer data key.
RecoverDesKekImporter	Recovers the cleartext form of a DES key encrypting key (KEK).

Overview

The routines described in this chapter are used to analyze, modify, and validate CCA DES key tokens.

Refer to the *IBM 4758 PCI 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.
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

```
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 been parity-adjusted.

Return Codes

This function has no return codes.

cas_build_default_cv - Build a Default Control Vector

cas_build_default_cv builds a default control vector for the specified key type.

Function Prototype

```
void  cas_build_default_cv( KEY_TYPES  KeyType,
                            UCHAR     *pCV )
```

Input

On entry to this routine:

`KeyType` is the type of key your control vector is used with.

`pCv` is a pointer to a 20-byte location which will hold the new control vector.

Output

On successful exit from this routine:

`pCV` contains the new control vector.

Return Codes

This function has no return codes.

cas_build_default_token - Build a Default Token

cas_build_default_token builds a default key token, of the type specified by parameter *KeyType*.

Function Prototype

```
void cas_build_default_token( UCHAR                    TokenFlag,
                              KEY_TYPES                KeyType,
                              des_key_token_structure *pKeyToken )
```

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 and EXTERNAL_TOKEN_FLAG.

`KeyType` is the type of key token to be generated. Examples include data key, exporter key, and MAC key.

`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.

cas_current_mkvp - Current Master Key Verification Pattern

cas_current_mkvp returns the 20-byte master key verification pattern (MKVP) for the current master key. 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

```
boolean cas_current_mkvp( UCHAR  *pMKVP )
```

Input

On entry to this routine:

`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.

`cas_current_mkvp` returns TRUE if the verification pattern was found, and FALSE otherwise.

Return Codes

This function has no return codes.

cas_old_mkvp - Old Master Key Verification Pattern

cas_old_mkvp returns the 20-byte master key verification pattern (MKVP) for the old master key. 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

```
boolean cas_old_mkvp( UCHAR        *pMKVP )
```

Input

On entry to this routine:

Output

On successful exit from this routine:

`pMKVP` contains the current master key verification pattern.

`cas_current_mkvp` 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.

Function Prototype

```
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.

Output

On successful exit from this routine:

`pMessageFlag` is a pointer to a location where the function stores an error code, if the key token is found to have an error.

`cas_des_key_token_check` returns a boolean value of TRUE, if the token has no errors, 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_RESERVED*i*	Reserved field *i* is incorrectly set.
DES_TOKEN_CHECK_VERSION	The version number is incorrect.
DES_TOKEN_CHECK_FLAGBYTES	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.

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

This function returns no 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.

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

```
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.

cas_master_key_check - Master Key Version Check

cas_master_key_check determines which version of the master key was used to encrypt the specified key token. 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  cas_master_key_check( 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_master_key_check` 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.

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 4758 PCI 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.

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.

RecoverDesDataKey - Recover DES Data Key

RecoverDesDataKey recovers the cleartext form of a DES data key that has been enciphered with the master key. The input token is checked to ensure it is valid.

Function Prototype

```
long RecoverDesDataKey(des_key_token_structure *pDesToken,
                       UCHAR                   *pClearKey,
                       long                    *pMsg)
```

Input

On entry to this routine:

`pDesToken` is a pointer to the input key token.

Output

On successful exit from this routine:

`pClearKey` is a pointer to the location where the function stores the recovered, cleartext key.

`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.
RT_KDATA_NOTODD	The cleartext key failed a parity check.

RecoverDesKekImporter - Recover DES Importer KEK

RecoverDesKekImporter recovers the cleartext form of a DES importer key encrypting key (KEK) that has been enciphered with the master key. The token validation value is then confirmed, and the key is checked for parity.

Function Prototype

```
long   RecoverDesKekImporter ( des_key_token_structure *pDesToken,
                               UCHAR                   *pClearKey,
                               long                    *pMsg )
```

Input

On entry to this routine:

`pDesToken` is a pointer to the input key token.

Output

On successful exit from this routine:

`pClearKey` is a pointer to the location where the function stored the recovered, cleartext key.

`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.
RT_KDATA_NOTODD	The cleartext key failed a parity check.

Chapter 9. RSA Functions

This chapter contains functions for dealing with RSA keys and key tokens.

Refer to Appendix B of the *IBM 4758 PCI 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.

```
#include "cmncryt2.h"                  /* T2 CPRB definitions      */
#include "scc_int.h"                   /* CP/Q++ API               */
#include "cam_xtrn.h"                  /* CCA managers             */
#include "cacdtkn.h"                   /* public header file       */
#include "casfunct.h"
```

Summary of Functions

RSA keys and key tokens include the following functions.

CalculatenWordLength	Returns the word length.
CreateInternalKeytoken	Receives a clear key token and creates the internal form.
CreateRsaInternalSection	Creates the RSA internal section.
delete_KeyToken	Remove a registered public or private key from storage.
GenerateCcaRsaToken	Generates a CCA RSA key token from an internal format key token and a CCA PKA skeleton token.
GenerateRsaInternalToken	Creates an internal RSA token from a CCA RSA key token.
GeteLength	Returns the RSA public exponent byte length.
getKeyToken	Retrieves a PKA token from the SRDI where it is stored.
GetModulus	Extracts and copies the RSA modulus.
GetnBitLength	Returns the bit length of the RSA modulus.
GetnByteLength	Returns the byte length of the RSA modulus.
GetPublicExponent	Extracts and copies the RSA key public exponent.
GetRsaPrivateKeySection	Returns a pointer to the private key section of an RSA key token.

GetRsaPublicKeySection	Returns a pointer to the public key section of an RSA key token.
GetTokenLength	Returns the length of the specified token.
IsPrivateExponentEven	Verifies whether the RSA private exponent is an even valued integer.
IsPrivateKeyEncrypted	Verifies whether the private key section of the specified key token is encrypted.
IsPublicExponentEven	Verifies whether the RSA public exponent is an even valued integer.
IsRsaToken	Verifies whether the key token contains an RSA key.
IsTokenInternal	Identifies whether the key token is in internal format.
PkaMkvpQuery	Returns a value indicating which master key was used to encrypt the specified key token.
pka96_tvvgen	Calculates the token validation value (TVV) for the specified key token.
RecoverPkaClearKeyTokenUnderMk	Recovers the PKA clear key token under the master key.
RecoverPkaClearKeyTokenUnderXport	Recovers the PKA clear key token under the DES export key.
ReEncipherPkaKeyToken	Re-enciphers an internal PKA key token from the old master key to the current master key.
RequestRSACrypto	Performs an RSA operation.
store_KeyToken	Saves a public or private key to the SRDI.
TokenMkvpMatchMasterKey	Tests whether the key token was encrypted using a specified version of the master key.
ValidatePkaToken	Verifies that the RSA key token is valid for use in the system.
VerifyKeyTokenConsistency	Verifies the consistency of a key token.

Overview

An RSA key consists of a public modulus which is the product of two large prime numbers, a public exponent which is relatively prime to the modulus, and a private exponent *d*. In the coprocessor, keys may be stored in CCA RSA tokens in the key storage file and used in the SCC complete tokens. Either form of key has a public and a private version.

The public version SCC complete 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. This version of a key is used in the cryptographic engine for sccRSA() requests and is the type returned by sccRSAKeyGenerate().

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. and a required public key section which holds encryption information (the public exponent and the modulus.)

Parts of the private key section may be encrypted under the master key (*internal keys*) or under a transport key (*external keys*). This is the version of a key which is 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.
GeteLength	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 encrypted.
IsRsaToken	Verifies whether the supplied token is an RSA token.
IsTokenInternal	Verifies whether the CCA RSA key token has been encrypted with the master key (that is, an *internal token*).
PkaMkvpQuery	Identifies which master key was used to encrypt the specified internal CCA RSA key token.
TokenMkvpMatchMasterKey	Tests whether the internal CCA RSA key token was encrypted with the specified master key.
ValidatePkaToken	Verifies that a CCA RSA key token is valid for use in the system.

VerifyKeyTokenConsistency	Tests the length fields of the CCA RSA key token, ensuring that they are consistent.

Key manipulation

CreateInternalKeyToken	Receives a clear CCA RSA key token and encrypts the private key data, creating an internal CCA RSA key token.
CreateRsaInternalSection	Receives an SCC complete token and creates an internal CCA RSA key token, including encrypting with the master key.
GenerateRsaInternalToken	Receives a CCA RSA key token and creates an SCC complete token.
GetModulus	Returns the public modulus of the CCA RSA key token.
GetPublicExponent	Returns the public exponent of the CCA RSA key token.
GetRsaPrivateKeySection	Returns a pointer to the private key section of the CCA RSA key token.
GetRsaPublicKeySection	Returns a pointer to the private key section of the CCA RSA key token.
pka96_tvvgen	Calculates the token validation value for the CCA RSA key token.
	Decrypts an internal CCA RSA key token.
RecoverPkaClearKeyTokenUnderXport	Decrypts an external CCA RSA key token.
ReEncipherPkaKeyToken	Decrypts a CCA RSA key token with the old master key and encrypts it with the current master key.
RequestRSACrypto	Performs an encryption or decryption operation with the CCA RSA key token.

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

```
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.

CreateInternalKeyToken - Create Internal Key Token

CreateInternalKeyToken receives a clear CCA RSA key token and creates the internal form by encrypting the private key areas under the master key.

Function Prototype

```
long CreateInternalKeyToken( RsaKeyTokenHeader  *pTokenIn,
                             RsaKeyTokenHeader  *pTokenOut )
```

Input

On entry to this routine:

`pTokenIn` is a pointer to the cleartext key token.

Output

On successful exit from this routine:

`pTokenOut` is a pointer to a location which contains the encrypted internal key token.

Return Codes

Common return codes generated by this routine are:

ERROR	The token is not an RSA token, or already has an internal section.
mk_KEY_NOT_VALID	The current master key is not valid.
mk_SEM_CLAIM_FAILED	Could not access the master keys.

CreateRsaInternalSection - Create RSA Internal Section

CreateRsaInternalSection receives an SCC complete token and creates an internal CCA RSA key token by calculating the validation values and encrypting under the master key.

Function Prototype

```
long CreateRsaInternalSection ( RsaKeyTokenHeader *pTokenOut,
                                sccRSAKeyToken_t  *pRsaTokenIn )
```

Input

On entry to this routine:

`pTokenOut` is a pointer to a variable which will hold the new CCA RSA key token.

`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 new 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

```
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.

GenerateCcaRsaToken - Generate CCA RSA Key Token

GenerateCcaRsaToken generates a CCA RSA key token from an internal (CP/Q++) format key token and a CCA PKA skeleton token. The skeleton token must be initialized to indicate the required format of the final token.

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 the desired CCA Key token type (RSA_PRIVATE_SECTION_NOPT, RSA_PRIVATE_SECTION_CR, RSA_PRIVATE_SECTION_NOPT_VAR, (for version 0 keys) or RSA_PRIVATE_SECTION_NOPT_NEW or RSA_PRIVATE_SECTION_CR_NEW (for version 1 keys)).

`pRsaKeyToken` must be a pointer to a valid internal (CP/Q++) RSA key token.

`internal` must be initialized to the version of the requested token.

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 complete token with all data aligned on 4-byte boundaries, for use in RSA computations.

Function Prototype

```
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.

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.

Note: The member field is a 16-bit field and is in S/390 (big-endian) format. This routine returns the 16-bit integer in Intel** (little-endian) format.

Function Prototype

```
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

```
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:

srdi_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

```
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.

Note: The member field is a 16-bit field and is in S/390 (big-endian) format. This routine returns the 16-bit integer in Intel** (little-endian) format.

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 *pDest*.

Function Prototype

```
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 64-byte buffer provided for the exponent.

Output

On successful exit from this routine:

`pDest` is a pointer to the caller's buffer where the RSA key public exponent is stored.

`GetPublicExponent` returns the length of the exponent.

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 RSA key token.

Output

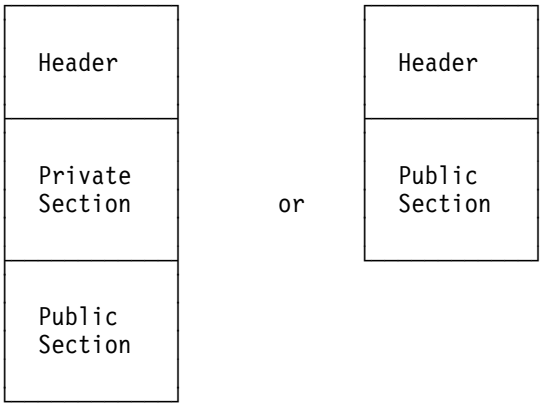
This function returns no output. On successful exit from this routine:

`GetRsaPrivateKeySection` returns a pointer to the private key section of an RSA key token.

Notes

Refer to Appendix B of the *IBM 4758 PCI 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:



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.

GetTokenLength - Return Key Token Length

GetTokenLength returns the length of the specified token, as contained in the member field of the header.

Note: The member field is a 16-bit field and is in S/390 (big-endian) format. This routine returns the 16-bit integer in Intel** (little-endian) format.

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

```
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

```
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.

IsTokenInternal - Key Token Format

IsTokenInternal returns TRUE if the specified key token is in *internal* 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 *internal* format, or FALSE if it is in *external* format.

Notes

Internal key tokens contain private key information that has been multiply-enciphered with the master key. RecoverPkaClearKeyTokenUnderMk() is used to decipher an internal key token so that it may be used.

Return Codes

This function has no return codes.

PkaMkvpQuery - Return Master Key Version

PkaMkvpQuery returns a value indicating which master key was used to encrypt the specified key token.

Function Prototype

```
MK_VERSION PkaMkvpQuery ( RsaKeyTokenHeader *pToken )
```

Input

On entry to this routine:

`pToken` is a pointer to the key token that is checked.

Output

On successful exit from this routine:

`PkaMkvpQuery` 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.

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

```
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` is a pointer to the location where the calculated TVV is stored.

Return Codes

This function has no return codes.

RecoverPkaClearKeyTokenUnderMk

RecoverPkaClearKeyTokenUnderMk receives a PKA key token which is encrypted under the master 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

```
long RecoverPkaClearKeyTokenUnderMk( RsaKeyTokenHeader  *pTokenIn,
                                     RsaKeyTokenHeader  *pTokenOut,
                                     long               *pMsg )
```

Input

On entry to this routine:

`pTokenIn` is a pointer to the encrypted key token.

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.

Notes

`RecoverPkaClearKeyTokenUnderMk` determines which master key was used to encipher.

Return Codes

Common return codes generated by this routine are:

OK	The operation was successful.
ERROR	`pRC` returns a CP/Q 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.

RecoverPkaClearKeyTokenUnderXport

RecoverPkaClearKeyTokenUnderXport receives a PKA key token which is encrypted under a DES export 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

```
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 the DES exporter key token.

`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.

ReEncipherPkaKeyToken - Re-Encipher PKA Key Token

ReEncipherPkaKeyToken re-enciphers an internal PKA key token from the old master key to the current master key.

Function Prototype

```
long ReEncipherPkaKeyToken( RsaKeyTokenHeader  *pToken,
                            UCHAR              *pWorkArea )
```

Input

On entry to this routine:

`pToken` is a pointer to the input key token, enciphered under the old master key.

`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 current master key.

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 RecoverPkaClearKeyTokenUnderMk().

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 CCA format RSA key token.

`DataBitLength` is the length of the input data, in bits.

`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.

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 little-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 a 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

```
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 cleared.

`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.

ValidatePkaToken - Validate RSA Key Token

ValidatePkaToken accepts a cleartext RSA key token, and verifies that the token is valid for use in the system.

Function Prototype

```
long ValidatePkaToken( 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.

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.

Chapter 10. CCA SRDI Manager Functions

This section 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.

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

Overview

The security relevant data item (SRDI)[1] Manager is the single interface through which all CCA-related functions access security related data. Only the SRDI Manager interacts with the physical medium on which the SRDI data is stored. The CCA verbs and any other CCA code read and write SRDI information only through the SRDI Manager interface. In turn, the SRDI Manager accesses the physical SRDI storage through the CP/Q++ PPD Manager, which controls the flash EPROM and BBRAM memories. This relationship is shown in Figure 10-1 on page 10-2.

[1] SRDI's are the sensitive data elements owned by the cryptographic application, and requiring protection. Examples include cryptographic keys and access control profiles.

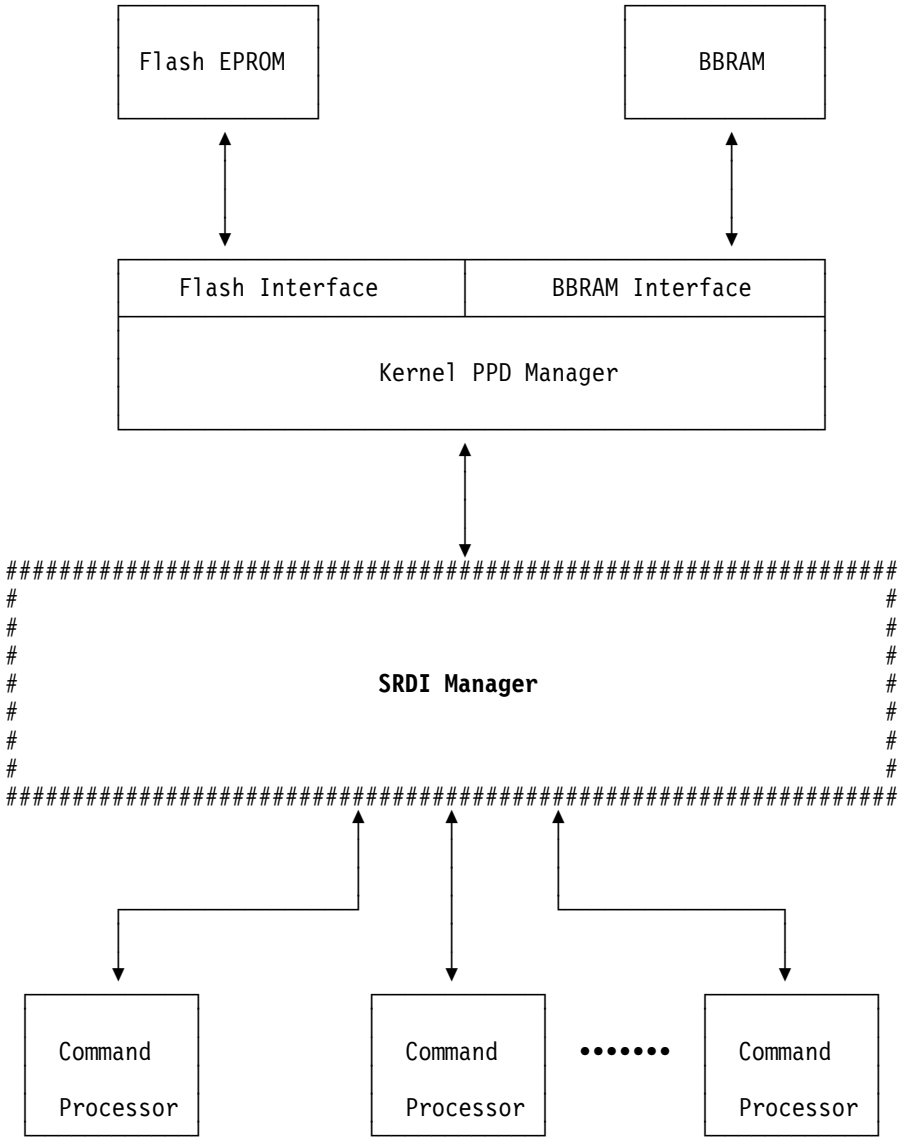


Figure 10-1. Master SRDI Manager Overview

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. Names that are less than eight characters should be left-justified, and padded on the right with ASCII spaces.

CCA SRDI Manager Operation

The CP/Q++ PPD Manager can store SRDI data in either of two physical memory types.

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 CP/Q++ PPD 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 SRDI data that is small, and frequently updated. Examples include session keys, sequence counters, and state information.

The interface functions provide a parameter to select whether an SRDI is created in flash or BBRAM.

CCA applications do not have direct access to the SRDI information in the persistent memories.[2] 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.

Step	Description
1.	A CCA command processor sends a request to the CCA SRDI Manager, asking for access to an SRDI named *ABC*, which resides in flash EPROM. At this time, SRDI *ABC* is not open. No copy of the SRDI data exists in the CCA application RAM address space.
2.	The CCA Manager sends a request to the Kernel PPD Manager, asking for the length of SRDI *ABC*. It needs to know the length, so it can allocate the required buffer in RAM.
3.	The Kernel PPD Manager returns the length of SRDI *ABC*.
4.	The CCA SRDI Manager allocates a buffer to hold *ABC*. This buffer is in RAM addressable by the CCA application.

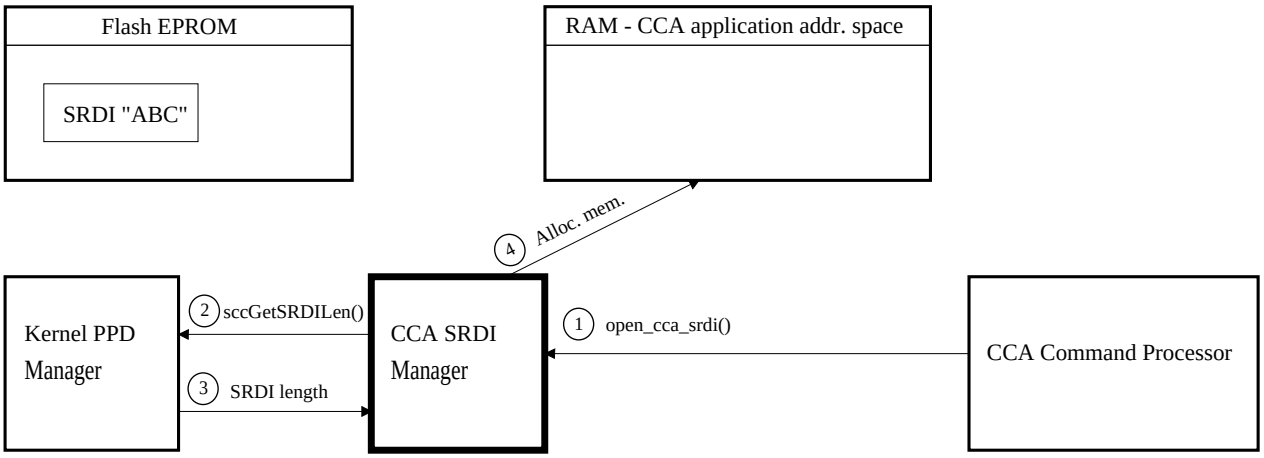


Figure 10-2. Master SRDI Read Illustration, Part 1

Step	Description
5.	The CCA SRDI Manager sends a request to the Kernel PPD Manager, asking it to read *ABC* into the buffer allocated in step 4 above.
6.	The SRDI is read from flash EPROM, decrypted, and deposited in the specified buffer.

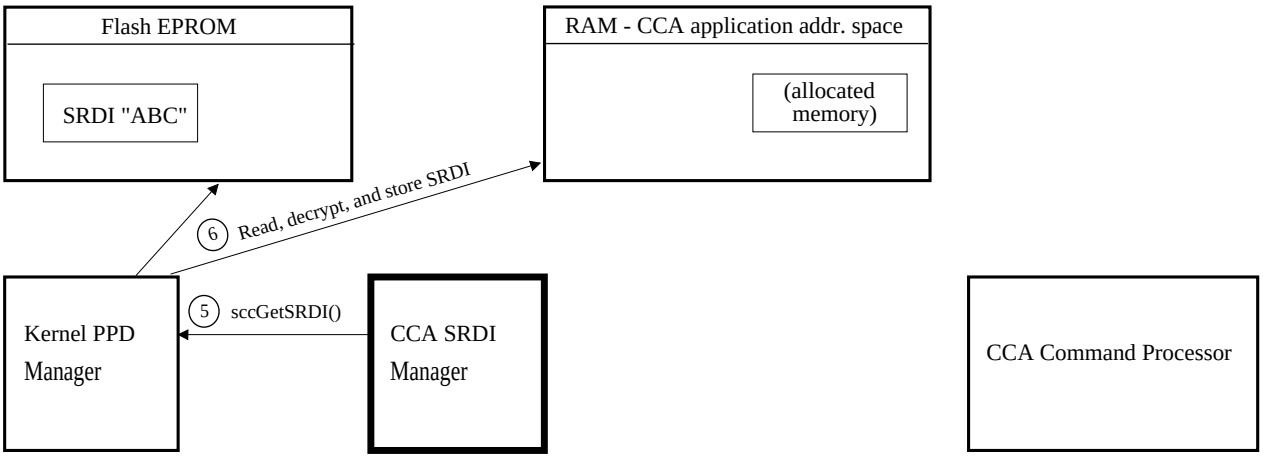


Figure 10-3. Master SRDI Read Illustration, Part 2

Step	Description
7.	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 *ABC*.

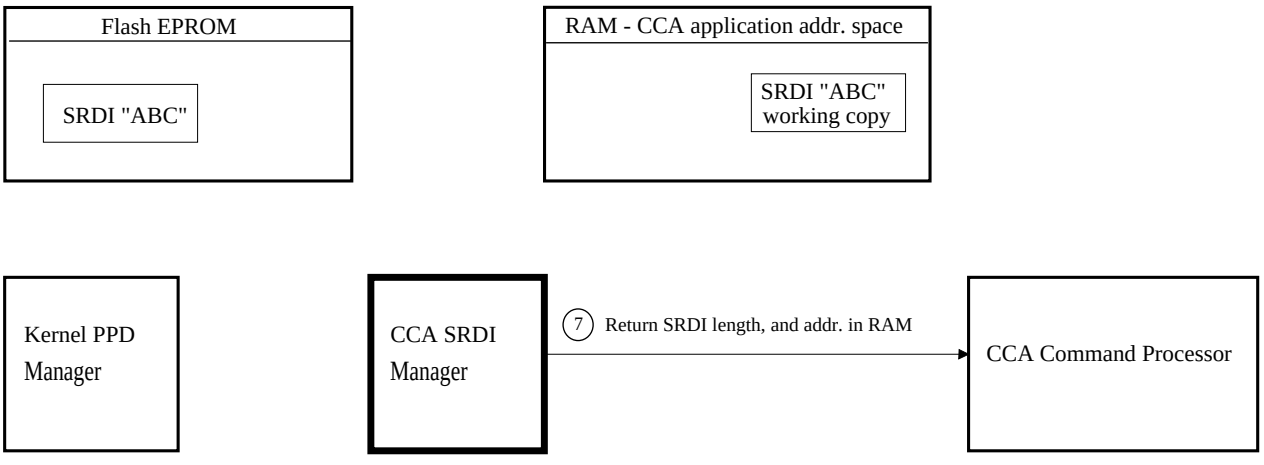


Figure 10-4. Master SRDI Read Illustration, Part 3

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 passed back to the caller whenever an SRDI is opened.

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 CP/Q system calls *CPSemClaim* and *CPSemRelease*. 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.

close_cca_srdi	Closes the open copy of an SRDI.
create_cca_srdi	Creates an SRDI.
delete_cca_srdi	Deletes an SRDI from memory.
get_cca_srdi_length	Obtains the length of an SRDI, in bytes.
open_cca_srdi	Opens and gains access to an SRDI.
resize_cca_srdi	Increases or decreases the length of an SRDI, in bytes.
save_cca_srdi	Stores SRDI data.

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.

Note: 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.

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	Can not 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 or BBRAM using the specified name.

Function Prototype

```
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.

`srdi_options` holds bit-significant options which are passed on to the CP/Q++ PPD Manager's *sccSaveSRDI()* function. There are two fields in the options value:

1. A value that indicates whether the SRDI data should be stored in flash EPROM, or in BBRAM. Flash is large, but slow to access, and each cell has a limited number of possible write cycles before it fails. BBRAM is fast and has unlimited write cycles, but it is much smaller than the flash.
2. A value which indicates how the SRDI data should be encrypted, if it is to be stored in flash EPROM.[3] There are three options.
   a. Do not encrypt the data at all.
   b. Single-encrypt with DES.
   c. Triple-encrypt with DES.

The options are defined with constants in header file *scc_int.h*. The values defined there are as follows.

Symbol	Value	Description
PPD_BBRAM	X'01'	Store in BBRAM
PPD_SINGLE	X'10'	Store in flash, encrypted using a single-length DES key.
PPD_TRIPLE	X'30'	Store in flash, encrypted using DES triple encryption.
PPD_NONE	X'00'	Store in flash, unencrypted.

`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.

[3] 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.

Output

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

`create_cca_srdi` creates an SRDI in flash EPROM or BBRAM using the specified name.

Return Codes

Common return codes generated by this routine are:

srdi_NO_ERROR	The operation was successful.
srdi_GENERAL_ERROR	Can not access the SRDI Manager, the operation cannot be completed.
srdi_EXISTS	The SRDI item already exists.

delete_cca_srdi - Delete CCA SRDI

delete_cca_srdi deletes an SRDI from the persistent memory area where it is stored (either flash EPROM or BBRAM). This is equivalent to erasing a file from a hard disk.

Function Prototype

```
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	Can not 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

```
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.

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	Can not access the SRDI Manager, the operation cannot be completed.
srdi_READ_ERROR	Unable to read the SRDI item from BBRAM or 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 or BBRAM.

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. 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 "Controlling Concurrent Access to an SRDI" on page 10-6 for further details.

Function Prototype

```
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.

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 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.

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

```
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 or BBRAM) using the encryption method specified when the SRDI was created.

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 “Controlling Concurrent Access to an SRDI” on page 10-6 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 SRDI’s can have the same name, even if one resides in flash EPROM and the other resides in BBRAM. The CP/Q++ PPD 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.

Example Code

The following C-language code shows a general structure for the way a CCA application would open, use, and close an SRDI.

```
#define QSVCgood 0                      /* CP/Q semaphore fcn. error code   */
#define TIMEOUT_FOREVER 0xFFFFFFFF      /* Parameter for CPSemClaim         */
#define MY_SRDI_NAME "MY_SRDI "         /* 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... */

    /* Gain exclusive access rights to the SRDI */

    if (QSVCgood == CPSemClaim(semaphore_id, TIMEOUT_FOREVER))
    {

      /* This is where the code will read and/or write to the SRDI data,
         in the area pointed to by my srdi addr. */

      /* Release semaphore, allowing others to access the SRDI */

      if (QSVCgood == CPSemRelease(semaphore_id))
      {

        /* do other stuff as needed... */

      }
      else
      {
        /* handle semaphore release error... */
      }

    }

    else
    {
      /* handle semaphore claim error... */
    }

     /* Close the SRDI */

     srdi_rc = close_cca_srdi(MY_SRDI_NAME);

     if (srdi_NO_ERROR != srdi_rc)
     {
       /* handle SRDI close error... */
     }

  }

  else
  {
    /* handle SRDI open 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 the coprocessor.

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 routines */
#include "camacm_p.h"           /* Access Control Manager routines */
```

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_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.

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_vers;               // Profile structure version
   two_byte profile_lth;                // Length of entire profile
   char comment[20];                    // 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_it_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_vers;                  // 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
   two_byte reqd_auth_str;              // Required authentication strength
   role_time_t upper_time_limit;        // Upper time limit for access
   dow_t valid_dow;                     // Valid days of the week for access
   UCHAR reserved_2;                    // Reserved field
   // Permitted operations definition (access control points) attached here
   role_time_t lower_time_limit;        // Lower time limit for access
} role_t
```

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.

```
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

```
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 us 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

```
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.
acm_PROFILE_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

```
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.
acm_PROFILE_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

```
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

```
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

```
long ac_get_profile(profile_id_t profile_id,
                    ULONG *profile_size, 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.
acm_PROFILE_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

```
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

```
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

```
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_MEM_ALLOC_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

```
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

```
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

```
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

```
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

```
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

```
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.

ac_lu_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

```
long ac_lu_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_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

```
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

```
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

```
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.

au_lu_query_user - Check if a User is Logged On

au_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

```
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:

`au_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

```
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

```
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

```
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

```
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.
acm_PROFILE_NOT_FOUND	The specified profile was not in the profile list.

Chapter 12. 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"                    /* DES 96 function prototypes */
#include "camacm.h"
#include "cmnfunct.h"
```

Summary of Functions

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.

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

```
#define CHECK_ACCESS_AUTH(Rqc, Rpc,r,c,g) check_access_auth_fcn(Rqc, Rpc, r, c, g)
ULONG check_access_auth_fcn( CPRB_ptr     pRequestCprb,
                             CPRB_ptr     pReplyCprb,
                             role_id_t    rolelID,
                             USHORT       requested_fcn_code,
                             boolean     *pGranted)
```

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.

`rolelID` 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

Access is granted to role IDs using the *csuncnm* utility. 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 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 If the error code pointed to by *message_ptr* is S_OK, the function result is set to the length of the key. Otherwise, the function returns a value of ERROR (-1), and the value pointed to by *message_ptr* is a SAPI error code.

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

```
ULONG intel_long_reverse(ULONG long_val)
```

For portability reasons, the following macros have been conditionally defined for integer translation.

```
#ifdef BIG_ENDIAN
    #define xtohl(d) ((ULONG)d)
    #define htoxl(d) ((ULONG)d)
    #define atohl(d) intel_long_reverse((ULONG)d)
    #define htoal(d) intel_long_reverse((ULONG)d)
#else
    #define xtohl(d) intel_long_reverse((ULONG)d)
    #define htoxl(d) intel_long_reverse((ULONG)d)
    #define atohl(d) ((ULONG)d)
    #define htoal(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.

For portability reasons, the following macros have been conditionally defined for integer translation.

```
#ifdef BIG_ENDIAN
    #define xtohs(d) ((USHORT)d)
    #define htoxs(d) ((USHORT)d)
    #define atohs(d) intel_word_reverse((USHORT)d)
    #define htoas(d) intel_word_reverse((USHORT)d)
#else
    #define xtohs(d) intel_word_reverse((USHORT)d)
    #define htoxs(d) intel_word_reverse((USHORT)d)
    #define atohs(d) ((USHORT)d)
    #define htoas(d) ((USHORT)d)
#endif
```

where:

x External
h Host
a Adapter

Function Prototype

```
USHORT intel_word_reverse(USHORT intel_int)
```

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.

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.

```
#define TOKEN_IS_A_LABEL(keyid) \
         (( keyid[0] >= MIN_FOR_LABEL ) && ( keyid[0] <= MAX_FOR_LABEL ))
```

TOKEN_LABEL_CHECK - Determine whether 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.

```
#define TOKEN_LABEL_CHECK(keyid) \
         (( keyid >= MIN_FOR_LABEL ) && ( keyid <= MAX_FOR_LABEL ))
```

Appendix A. UDX Sample Code - Host Piece

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.

```
/**************************
**  Bring in the include files
**************************/
#include <stdlib.h>
#include <string.h>

#include "cmncryt2.h"            /* Cryptograhic types             */
#include "cmnerrcd.h"            /* Common error codes.            */
#include "cmnlibr.h"             /* Common library routines.       */

#include "safhead1.h"            /* External SAPI functions.       */
#include "safcextn.h"            /* External SAPI functions.       */

#include "csueextn.h"            /* CCA extensions for SAPI API    */
#include "cxt_cmds.h"            /* CCA extensions for SAPI, cmdproc*/

/*********************************************************************/
/*                  CCAXFCN1                                         */
/*-------------------------------------------------------------------*/
/* Purpose                                                           */
/*    This is the entry point for a sample CCA API extension verb.   */
/*                                                                   */
/*    This sample's input parameters are similiar to a CCA verb's    */
/*    parameters.  Following is an explanation of the parameters.    */
/*                                                                   */
/*      pReturnCode     -- Where to return the return code value.    */
/*      pReasonCode     -- Where to return the completion reason.    */
/*      pExitDataLength -- Unused parameter placeholder.             */
/*      pExitData       -- Unused parameter placeholder.             */
/*      pRuleArrayCount -- Pointer to number of 8 byte rule array    */
/*                         entries.                                  */
/*      pRuleArray      -- Pointer to the rule array values.         */
/*      pKeyId          -- Pointer to a 64 byte key or a key token.  */
/*                                                                   */
/*********************************************************************/
void SECURITYAPI CCAXFCN1
(
   long          * pReturnCode,
   long          * pReasonCode,
   long          * pExitDataLength,
   unsigned char * pExitData,
   long          * pRuleArrayCount,
   unsigned char * pRuleArray,
   unsigned char * pKeyId
)
{
  /*******************
  ** Local variables.
  *******************/
  long        ErrorMsg;              /* Error message of SAPI routines */
  CPRB_ptr    pCprb;                 /* CPRB pointer                   */
  UCHAR      *pReqBlk;               /* request parm blk pointer       */

  USHORT      ReqBlkLen;             /* request parm buffer length     */

  USHORT      KeyIdLength;           /* Length of key id/token.        */

  KEY_FIELD_HEADER KeyBlkHeader;     /* Key block header area.         */
  KEY_FIELD_HEADER KeyBlkReturn;     /* Key block header return area.  */

  REQUEST_REPLY_BUF *  pReqReplyBuf; /* Buffer area pointer for request*/
                                     /* and reply CPRB/parm areas.     */
  UCHAR   KeyLabel[64];              /* A key token with no data.      */
  boolean KeyIdIsLabel;              /* Truth value that the key id was*/
```

```
                                  /* passed as a label              */
key_data_structure *pKeyBlock;      /* Ptr. to generated key          */
key_data_structure *pNextKeyBlock;  /* Ptr. to generated key          */
generic_key_block_structure *pKeyField; /* Ptr. to start of a key field.*/

/*******************
** Assure we can return values without causing an address exception
** by checking for NULL pointers.
*******************/
if ( (pReturnCode == NULL) || (pReasonCode == NULL) )
   return;

/*******************
** Initialize return code and return code.
*******************/

*pReturnCode = 0;               /* reset return code           */
*pReasonCode = 0;               /* reset reason code           */

/*******************
** Assure we can use the passed pointers without causing an
** address exception by checking that required pointers are
** not NULL.
*******************/
if ( (pRuleArrayCount == NULL) || (pRuleArray == NULL) ||
     (pKeyId     == NULL) )
{
   CSUC_PROCRETC(pReturnCode, pReasonCode, E_NULL_PTR);
   return;
}

/*******************
** OPTIONAL ----- depends on whether your CCA extension has
**                rule array values.
**
** Assure that the rule count is reasonable.
*******************/
if ( (*pRuleArrayCount < RAC_MIN) || (*pRuleArrayCount > RAC_MAX) )
{
  CSUC_PROCRETC(pReturnCode, pReasonCode, E_RULE_ARRAY_CNT);
  return;
}

/*******************
** OPTIONAL ----- depends on whether your CCA extension uses
**                key tokens.
**
** Get the length of the key identifier field and check for validity.
*******************/
KeyIdLength = GetKeyLength( pKeyId, NULL, &ErrorMsg);

if( (ErrorMsg != S_OK) &&
    (CSUC_PROCRETC(pReturnCode, pReasonCode, ErrorMsg) == ERROR))
{
  return;
}

/*******************
**
**
** Add your own checking, etc. logic here.
**
**
*******************/

/*******************
** Build the 4 parts of the CPRB "T2" Request Parameter Block
** that is sent to the adapter for processing.
** The block consists of the following fields.
**
**  +---------+------+-------//-+------+------//-+------+-----//-+
**  |Sub-     |Rule Array       |Verb Unique     |Key Block      |
```

```
**  |Function |                |               |Fields         |
**  |Code     |      | Array   |      |        |      |        |
**  |         |Length| Elements |Length| Data   |Length| Fields |
**  +---------+------+-------//-+------+------//-+------+-----//-+
**  0         2      4          2+X    4+X       2+X+Y  4+X+Y
**
**  |<-- 2 -->|<-- X ---------->|<-- Y --------->|<-- Z -------->|
**
******************/

/*********************************
** Allocate working space for the sending CPRB, request
** parameter block, returning CPRB and the reply
** parameter block.
*********************************/
pReqReplyBuf = malloc( sizeof(REQUEST_REPLY_BUF) );

if ( pReqReplyBuf == NULL )
{
  CSUC_PROCRETC ( pReturnCode, pReasonCode, E_ALLOCATE_MEM ) ;
  return;
}

/*********************************
** Establish addressability to the CPRB area and the request
** parameter block.
**
** NOTE: The CPRB and the request parameter block must be built
**       in the allocated structure in the following manner.
**       +----------+--------------------+
**       |CPRB      | Request parameter  |
**       |structure | block              |
**       +----------+--------------------+
*********************************/
pCprb   = (CPRB_ptr) &(pReqReplyBuf->request_buf[0]) ;
pReqBlk = &(pReqReplyBuf->request_buf[0]) +
            sizeof(CPRB_structure);

/*******************
** Part 1 of 4.
**
** Put the 2-byte subfunction code at the beginning of the
** request parameter area.
**
** NOTE- Replace CCAXFNC1_ID with the subfunction code for your
**       CCA API extension.
*******************/
ReqBlkLen = 2;
*((USHORT *) pReqBlk) = htoas ( CCAXFNC1_ID ) ;

/*******************
** Part 2 of 4.
**
** Add the rule array block to the request parameter area.
**
** NOTE- If there are no rule array values, a 2 byte length field
**       with a value of 2 must be added.
*******************/
ReqBlkLen += BuildParmBlock(pReqBlk + ReqBlkLen, /* Target address*/
                            1,             /* # of blocks to add.*/
                            (USHORT) (8 * *pRuleArrayCount),
                            pRuleArray );

/*******************
** Part 3 of 4.
**
** Add the verb unique data block to the request parameter area.
**
** NOTE- If there is no verb unique data block, a 2 byte length field
**       with a value of 2 must be added.  This is an example of how
**       to just add a 2 byte length field when there is no
**       additional data.
```

```
******************/
ReqBlkLen += BuildParmBlock(pReqBlk + ReqBlkLen, /* Target address */
                            0,        /* # of blocks to add. */
                            0, NULL);

/******************
** Part 4 of 4.
**
** Add the key block to the request parameter area.  The
** key block can consist of key tokens or key labels.
**
** NOTE- If there is no key block, a 2 byte length field
**       with a value of 2 must be added.
******************/
/* Check first byte to determine if pKeyId is a label or a token    */
if ( (*pKeyId >= MIN_FOR_LABEL) &&
     (*pKeyId <= MAX_FOR_LABEL) )
{
  /******************
  ** The application specified a key label for the key token.
  ** The key label will be replaced by a key token from the key
  ** storage file prior to sending this request to the adapter.
  **
  ** NOTE- When a key token is going to be updated in the key
  **       storage file by this operation, the second key block
  **       must be specified for returning and updating of
  **       the key token in the key storage file.
  ******************/
  if ( saf_process_key_label( pKeyId, true, KeyLabel )
       == false )
  {
     /******************
     ** The format of the key label is invalid.
     ******************/
     free( pReqReplyBuf );
     CSUC_PROCRETC(pReturnCode, pReasonCode, E_KEY_LABEL_FMT);
     return;
  }

  KeyIdIsLabel = true;
  KeyBlkHeader.Flags  = htoas ( DES96_TYPE | ACTION_READ ) ;
  KeyBlkHeader.Length = htoas ( KEY_HDR_LEN + sizeof(KeyLabel) ) ;

  KeyBlkReturn.Flags  = htoas ( DES96_TYPE | ACTION_NOOP ) ;
  KeyBlkReturn.Length = htoas ( KEY_HDR_LEN + sizeof(KeyLabel) ) ;

  ReqBlkLen += BuildParmBlock(pReqBlk + ReqBlkLen, /* Target address*/
                           4,                /* # of blocks to add.*/
                           sizeof(KeyBlkHeader), &KeyBlkHeader,
                           sizeof(KeyLabel),     KeyLabel,
                           sizeof(KeyBlkReturn), &KeyBlkReturn,
                           sizeof(KeyLabel),     KeyLabel);
}
else
{
  /******************
  ** A key token was specified for the key identifier.
  ** An actual key token is being passed instead of a key label.
  ******************/
  KeyIdIsLabel = false;
  KeyBlkHeader.Flags  = htoas ( DES96_TYPE | ACTION_NOOP ) ;
  KeyBlkHeader.Length = htoas ( KEY_HDR_LEN + KeyIdLength ) ;

  ReqBlkLen += BuildParmBlock(pReqBlk + ReqBlkLen, /* Target address*/
                           2,                /* # of blocks to add.*/
                           sizeof(KeyBlkHeader), &KeyBlkHeader,
                           KeyIdLength, pKeyId );
}

/******************
** Build the CPRB
**
```

```
** NOTE- If your extension uses the data buffer, you must change
**       the request/reply data buffer parameters in the following
**       statement.
******************/
CSUC_BULDCPRB(pCprb,
              (UCHAR *) ESSS_FUNCTION_ID_S,
              ReqBlkLen, pReqBlk,           /* Request parm. buffer    */
              0, (UCHAR *) NULL,            /* Request data buffer     */
              sizeof( pReqReplyBuf->reply_buf ),
              pReqReplyBuf->reply_buf,      /* Reply parameter buffer  */
              0, (UCHAR *) NULL);           /* Reply data buffer       */

/******************
**
** SEND REQUEST TO THE COPROCESSOR FOR EXECUTION.
**
** The CPRB request is sent to the Security Server which in turn then
** forwards it to the device driver.
******************/
CSNC_SP_SCSRFBSS( pCprb, &ErrorMsg);


/******************
** Check the return/reason codes of the completed operation.
**
** NOTE- CSUC_PROCRETC returns ERROR if the error code in msg is higher
**       than the error code already in *pReturnCode and *pReasonCode.
**       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 ( (ErrorMsg != S_OK) &&
     (CSUC_PROCRETC(pReturnCode, pReasonCode, ErrorMsg) == ERROR) )
{
  free( pReqReplyBuf );
  return;
}

/******************
** Process the returned data, which is in the reply parameter block.
** The reply parameter block must have the same format as the
** request parameter block.
**
** Examine the reply parameter block to make sure it is OK.  If not,
** something went wrong in the adapter - it should return valid data.
******************/
if ( parm_block_valid( pCprb, SEL_REPLY_BLK ) == false)
{
  CSUC_PROCRETC( pReturnCode, pReasonCode, CP_DEV_HWERR );
  free( pReqReplyBuf );
  return;
}

/******************
** OPTIONAL ----- depends on your CCA extension.
**
** Find the first key in the key block of the reply parameter block.
** This will be the newly generated key, which we will pass back to the
** caller.  If there is no key, abort with an error.
******************/
if ( find_first_key_block( pCprb, &pKeyBlock, SEL_REPLY_BLK) == false)
{
  CSUC_PROCRETC( pReturnCode, pReasonCode, FT_INCONSIST_DATA);
  free( pReqReplyBuf );
  return;
}

/******************
** OPTIONAL ----- depends on your CCA extension.
**
** Assure that only one key block is returned.
```

```
  ******************/
  if ( find_next_key_block( pCprb, pKeyBlock,
                            &pNextKeyBlock, SEL_REPLY_BLK) == true)
  {
    CSUC_PROCRETC( pReturnCode, pReasonCode, FT_INCONSIST_DATA);
    free( pReqReplyBuf );
    return;
  }

  /*******************
  ** OPTIONAL ----- depends on your CCA extension.
  **
  ** We've successfully located the first key in the key block, and its
  ** address is in pKeyBlock.   Everything looks OK.
  *******************/
  pKeyField = (generic_key_block_structure *) pKeyBlock;
  if ( KeyIdIsLabel == false )
  {
    /*******************
    ** Since KeyId is not a label, we need to pass the re-enciphered
    ** key token back to the caller. pKeyBlock points to the start of the
    ** entire key field, including the key field header (length and flag
    ** bytes).
    *******************/
    memcpy( pKeyId, &(pKeyField->label_or_token), DES_TOKEN_LENGTH);
  }

  /*********************************
  ** Free the space allocated for the CPRB and the request/reply
  ** data areas and then return to the calling application.
  *********************************/
  free( pReqReplyBuf );
  return;

} /* end-of CCAXFCN1() */
```

Appendix B. UDX Sample Code - Coprocessor Piece

This appendix contains a listing of the sample file *cxt_samp.c*. This file is a skeleton for the design of the adapter piece of a CCA extension, including the command processor.

```
/***************************
**  Bring in the include files
**
**  If you are using any of the "C" functions provided by CPQ,
**  include the following CPQ files instead of the standard
**  "C" files ( stdlib.h, string.h, etc. ).
**
**    #include <clib.h>
**    #include <cpqlib.h>
**    #include <sys\stat.h>
**    #include <qrm_cnst.h>
**
**************************/

#include "cmncryt2.h"           /* CCA common definitions           */
#include "camacm.h"             /* Access manager definitions       */

#include "cmnerrcd.h"           /* Common error codes.              */
#include "cam_xtrn.h"           /* CCA managers                     */

#include "cassub.h"
#include "casfunct.h"           /* Common command processor funct's */
#include "cacdtkn.h"            /* PKA token subroutines.           */
#include "cacscacp.h"           /* Access control codes             */

#include "cxt_cmds.h"           /* CCA command extensions.          */


/*****************************************************************
** 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 } };


/*******************************
**
** Declare a variable which holds the number of CCA extension verbs
** defined in the ccax_cp_list table above.
**
*******************************/
ULONG ccax_cp_list_size = 0;      // NOTE:
                                  // Once you add your first entry to
                                  // "ccax_cp_list", delete this
                                  // statement and un-comment the
                                  // following statement.

// ULONG ccax_cp_list_size = ( sizeof(ccax_cp_list) / sizeof(CCAX_CP_DEF) );


/*******************************
** Local rule array value definitions.
*******************************/
#define  KEY_CLR    11
#define  KEY_ENC    12
#define  KEY_CLRD   13
```

```
#define  KEY_ENCD   14
#define  KEY_KM     15
#define  KEY_NKM    16
#define  KEY_OKM    17
#define  GENERATE   20
#define  VERIFY     21
#define  DFLT_CF    31
#define  ADAPTER    32


/*********************************************************************/
/*                     ccax_fcn_1                                    */
/*-------------------------------------------------------------------*/
/* Purpose:                                                          */
/*   This sample CCA extension shows the general structure of a      */
/*   command processor.                                              */
/*                                                                   */
/* Linkage:  see below                                               */
/*                                                                   */
/* Input:    See below                                               */
/*                                                                   */
/* Output:                                                           */
/*   The output CPRB and reply parameter blocks contains the         */
/*   results.                                                        */
/*                                                                   */
/* Return Code:                                                      */
/*   Passed back in the CPRB.                                        */
/*********************************************************************/
void ccax_fcn_1(
      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          */
{
  /*********************************
  ** Define the rule array for converting 8 character rules into
  ** integer values.  This example has 11 rules divided into 3 groups.
  ** On every request, there can be 3 choices, 1 from each group.
  *********************************/
  int RuleValue[ 3 ];      /* Variable to encrypt rule values.*/
  USHORT RuleMapCount = 11;
  static RULE_MAP RuleMap[] = {{"KEY-CLR ", 1 , KEY_CLR },
                               {"KEY-ENC ", 1 , KEY_ENC },
                               {"KEY-CLRD", 1 , KEY_CLRD},
                               {"KEY-ENCD", 1 , KEY_ENCD},
                               {"KEY-KM  ", 1 , KEY_KM  },
                               {"KEY-NKM ", 1 , KEY_NKM },
                               {"KEY-OKM ", 1 , KEY_OKM },
                               {"GENERATE", 2 , GENERATE},
                               {"VERIFY  ", 2 , VERIFY  },
                               {"DFLT-CF ", 3 , DFLT_CF },
                               {"ADAPTER ", 3 , ADAPTER },};

  /*********************************
  ** Define local variables.
  *********************************/
  long ReturnMsg;                /* Return code messages.              */

  RBFPTR        pReqBlk ;
  RBFPTR        pReplyBlk ;
  void          *pToken;

  UCHAR         *pKeyBlock ;
  UCHAR         *pLabel;
  USHORT        ReplyLength;

  des_key_token_structure     *pDesToken;
  verb_unique_data_structure  *pVerbData;
  generic_key_block_structure *pKeyBlockStruct;

  mk_status_var MstrKeyStatus;
```

```
boolean       KeyIdIsLabel;     /* Truth value that the key id was */
                                /* passed as a label               */

long   ReplyBlockLength;        /* length of the data added to the */
                                /*    reply block                  */
KEY_FIELD_HEADER KeyHeader;     /* header for key in reply block   */
boolean Authorized;

if ( RequestId == 0)       /* Do nothing statement to get rid of   */
  ReplyLength = 0;         /* compiler warning messages because    */
                           /* RequestId is not used.               */

/*********************************
** Copy input CPRB to the output area.
*********************************/
memcpy ( pCprbOut, pCprbIn, pCprbIn->CPRB_length );

Cas_proc_retc ( pCprbOut, S_OK );

/*********************************
** 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;
pReplyBlk = pCprbOut->reply_parm_block;

/*********************************
** Set the reply subfunction code early, because the Cas_proc_retc
** routine needs it set for negative return codes.
*********************************/
pReplyBlk->subfunction_code  = pReqBlk->subfunction_code;

/*********************************
** OPTIONAL ----- depends on your CCA extension.
**
** Make sure this service is authorized before we go any further
**
** NOTE: Replace SCA_COMMAND_BKTC with your own access point for
**       this extension.  CACSCACP.H defines what range of access
**       points are available for extension services.
*********************************/
CHECK_ACCESS_AUTH( pCprbIn,
                   pCprbOut,
                   roleID,
                   SCA_COMMAND_BKTC,   // Access control point.
                   &Authorized );

if ( Authorized == false )
{
    Cas_proc_retc ( pCprbOut, CP_NOT_AUTH ) ;
    return ;
}

/*********************************
** OPTIONAL ----- depends on your CCA extension.
**
** Make sure the current master key is valid before we go
** any further.
*********************************/
switch ( get_master_key_status ( &MstrKeyStatus ) )
{
   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_MGR_ERROR );
      return ;
}

/*********************************
** 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 ;
}

/*********************************
** OPTIONAL ----- depends on your CCA extension.
**
** Perform consistency check on the rule array - for this verb, the
** rule array always has one value.
*********************************/
if ( pReqBlk->rule_array_length != ( sizeof(pReqBlk->rule_array_length)
                                   + sizeof(rule_array_element) ) )
{
   Cas_proc_retc( pCprbOut, E_RULE_ARRAY_CNT );
   return ;
}

/*******************
** OPTIONAL ----- depends on your CCA extension.
**
** Compare for valid rule array values.
*******************/
RuleValue[0] = INVALID_RULE;  /* rule_check requires this initialization.*/
RuleValue[1] = INVALID_RULE;  /* rule_check requires this initialization.*/
RuleValue[2] = INVALID_RULE;  /* rule_check requires this initialization.*/

if ( rule_check ( (RULE_BLOCK *) &pReqBlk->rule_array_length,
                  RuleMapCount,
                  &RuleMap[0], &RuleValue[0], &ReturnMsg )
     == false )
{
    Cas_proc_retc ( pCprbOut, ReturnMsg ) ;
    return ;
}

/*******************
** OPTIONAL ----- depends on your CCA extension.
**
** Perform consistency check on the verb unique data - for this verb,
** the verb unique data is not used.
*******************/
pVerbData = (verb_unique_data_structure *)
            ((UCHAR *) &(pReqBlk->rule_array_length) +
                        pReqBlk->rule_array_length ) ;

if ( pVerbData->verb_unique_data_length != NO_VERBDATA )
{
   Cas_proc_retc ( pCprbOut, E_SIZE ) ;
   return ;
}

/*******************
** OPTIONAL ----- depends on your CCA extension.
**
** Perform consistency check on the key block - there should be
** 1 or 2 key block fields for this verb.
*******************/
```

```
if ( find_first_key_block( pCprbIn,
                           (key_data_structure **) &pToken,
                           SEL_REQ_BLK ) == false )
{
   Cas_proc_retc ( pCprbOut, RT_CONSISTENCY_ERROR ) ;     /* @p1c */
   return ;
}

if ( find_next_key_block( pCprbIn,
                          (key_data_structure *) pToken,
                          (key_data_structure **) &pKeyBlockStruct,
                          SEL_REQ_BLK ) == true )
{
   pLabel = &(pKeyBlockStruct->label_or_token);
   KeyIdIsLabel = true;
}
else
   KeyIdIsLabel = false;

/*******************
** OPTIONAL ----- depends on your CCA extension.
**
** Check for unexpected key label.  When SECY cannot find the requested
** key storage record, the key label is forwarded.
*******************/
pKeyBlock = (UCHAR *) &(( (generic_key_block_structure *)
                           pToken)->label_or_token );
if ( TOKEN_LABEL_CHECK( *pKeyBlock ) == TRUE )
{
   Cas_proc_retc ( pCprbOut, RT_KEYLABEL_NEXIST );
   return;
}

/*******************
** OK, the input looks pretty good, but will the reply buffer hold the
** output from this CCA extension.
*******************/
ReplyLength = pCprbIn->CPRB_length + sizeof(pReqBlk->subfunction_code) +
              ( 3 * sizeof(lthfield) ) + KEY_HDR_LEN + DES_TOKEN_LENGTH;

if ( pCprbIn->reply_parm_block_length < ReplyLength )
{
   Cas_proc_retc ( pCprbOut, BUFFER_LENGTH_ERROR );
   return ;
}

/*******************
**
**
** Add code to perform the requested function.
**
**
*******************/

/*******************
** Build 4 parts in the following Reply Parameter Block.
**
**  +---------+------+-------//-+------+------//-+------+-----//-+
**  |Sub-     |Rule Array       |Verb Unique     |Key Block      |
**  |Function |                 |                |Fields         |
**  |Code     |      | Array    |      |         |      |        |
**  |         |Length| Elements |Length| Data    |Length| Fields |
**  +---------+------+-------//-+------+------//-+------+-----//-+
**  0         2      4          2+X    4+X       2+X+Y  4+X+Y
**
**  |<-- 2 -->|<-- X ---------->|<-- Y --------->|<-- Z -------->|
**
*******************/

/*******************
** Build the response for returning the re-enciphered key token.
*******************/
```

```
  pCprbOut->reply_parm_block   = pReplyBlk ;
  pReplyBlk->rule_array_length = NO_RULEARRAY ;

  pVerbData = (verb_unique_data_structure *)
              ((UCHAR *) &pReplyBlk->rule_array_length +
                          pReplyBlk->rule_array_length ) ;
  pVerbData->verb_unique_data_length = NO_VERBDATA ;

  pKeyBlock = ((UCHAR *) &pVerbData->verb_unique_data_length +
                         pVerbData->verb_unique_data_length ) ;

  ReplyBlockLength = BLOCK_LENGTH + NO_RULEARRAY + NO_VERBDATA;

  if ( KeyIdIsLabel == true )
  {
     KeyHeader.Length = KEY_HDR_LEN + KEY_LABEL_LEN + DES_TOKEN_LENGTH;
     KeyHeader.Flags  = DES96_TYPE | ACTION_WRITE;

     ReplyBlockLength += BuildParmBlock( pKeyBlock,
                                         3,
                                         KEY_HDR_LEN, &KeyHeader,
                                         KEY_LABEL_LEN, pLabel,
                                         DES_TOKEN_LENGTH, pDesToken);
  }
  else
  {
     KeyHeader.Length = KEY_HDR_LEN + DES_TOKEN_LENGTH;
     KeyHeader.Flags  = DES96_TYPE | ACTION_NOOP;

     ReplyBlockLength += BuildParmBlock( pKeyBlock,
                                         2,
                                         KEY_HDR_LEN, &KeyHeader,
                                         DES_TOKEN_LENGTH, pDesToken);
  }

  /*******************
  ** Set the replied parameter block length and return to caller.
  *******************/
  pCprbOut->replied_parm_block_length = ReplyBlockLength;
  return;

} /* end-of ccax_fcn_1() */
```

Appendix C. Notices

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

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, North Castle Drive, Armonk, NY, 10504-1785, USA.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

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.

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

COPYRIGHT LICENSE: This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM’s application programming interfaces.

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

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

AIX
IBM
OS/2
S/390

Intel is a registered trademark of Intel Corporation in the United States, or other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

SET and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

UNIX is a registered trademark in the United States, or other countries, or both and is licensed exclusively through X/Open Company Limited.

Other company, product, and service names may be the trademarks or service marks of others.

List of Abbreviations and Acronyms

AIX	Advanced Interactive Executive (operating system)
API	application program interface
ASCII	American Standard Code for Information Interchange
BBRAM	battery-backed random access memory
CCA	Common Cryptographic Architecture
CDMF	Commercial Data Masking Facility
CMK	current master key
CP/Q	Control Program/Q
CPRB	Cooperative Processing Request Block
DES	Data Encryption Standard
DLL	dynamic load library
EPROM	erasable programmable read-only memory
FIPS	Federal Information Processing Standard
KEK	key encrypting key
IBM	International Business Machines
MAC	message authentication code
MKVP	master key verification pattern
NMK	new master key
OMK	old master key
OS/2	Operating System/2
PCI	peripheral component interconnect
PDF	portable document format
PIN	personal identification number
PKA	public key algorithm
PPD	program proprietary data
RAM	random access memory
RNG	random number generator
RSA	Rivest-Shamir-Adleman (algorithm)
SCC	secure cryptographic coprocessor
SET	Secure Electronic Transaction
SHA	Secure Hash Algorithm
SRDI	security relevant data item
TVV	token validation value
UDX	user-defined extensions
VUD	verb unique data

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.

Advanced Interactive Executive (AIX) operating system. The IBM implementation of the UNIX** operating system.

agent. (1) An application that runs within the IBM 4758 PCI Cryptographic Coprocessor. (2) Synonym for *secure cryptographic coprocessor application*.

AIX operating system. Advanced Interactive Executive operating system.

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

ANSI. American National Standards Institute.

API. Application program interface.

application program interface (API). A functional interface supplied by the operating system, or by a separate program, that allows an application program written in a high-level language to use specific data or functions of the operating system or that separate program.

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

authorization. (1) In computer security, the right granted to a user to communicate with or make use of a computer system. (T) (2) The process of granting a user either complete or restricted access to an object, resource, or function.

authorize. To permit or give authority to a user to communicate with or make use of an object, resource, or function.

B

battery-backed random access memory (BBRAM). Random access memory that uses battery power to retain data while the system is powered off. The IBM 4758 PCI Cryptographic Coprocessor uses BBRAM to store persistent data for SCC 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.

CBC. Cipher Block Chain.

CCA. Common Cryptographic Architecture.

CDMF algorithm. Commercial Data Masking Facility algorithm.

ciphertext. (1) Data that has been altered by any cryptographic process. (2) See *clear data*.

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

cleartext. (1) Data that has not been altered by any cryptographic process. (2) See *clear data*. (3) See also *ciphertext*.

clear data. Data that is not enciphered.

Commercial Data Masking Facility (CDMF) algorithm. An algorithm for data confidentiality applications; it is based on the DES algorithm and has an effective key strength of 40 bits.

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 PCI Cryptographic Coprocessor CCA Basic Services Reference and Guide*.

Control Program/Q (CP/Q). The operating system embedded within the IBM 4758 PCI Cryptographic Coprocessor. The version of CP/Q used by the coprocessor—including extensions to support cryptographic and security-related functions—is known as CP/Q++.

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.

CP/Q. Control Program/Q.

Cryptographic Coprocessor (IBM 4758). An expansion card that provides a comprehensive set of cryptographic functions to a workstation.

cryptography. (1) The transformation of data to conceal its meaning. (2) In computer security, the principles, means, and methods used to transform data.

D

data encrypting key. (1) A key used to encipher, decipher, or authenticate data. (2) Contrast with *key-encrypting key*.

Data Encryption Standard (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.

Federal Information Processing Standard (FIPS). A standard that is published by the US National Institute of Science and Technology.

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 IBM 4758 PCI Cryptographic Coprocessor, 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. (T) (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.

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 or 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

NT. See *Windows NT*.

O

Operating System/2 (OS/2). An IBM operating system for personal computers.

OS/2. Operating System/2.

P

PKA. Public key algorithm.

PPD. Program proprietary data.

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. (I) A procedure call usually includes an entry name and the applicable parameters.

program proprietary data (PPD). Persistent data stored within the IBM 4758 PCI Cryptographic Coprocessor flash memory or battery-backed RAM that is associated with a particular agent.

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 IBM 4758 PCI Cryptographic Coprocessor 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

SCC. Secure cryptographic coprocessor.

secure cryptographic coprocessor (SCC). An alternate name for the IBM 4758 PCI Cryptographic Coprocessor. The abbreviation “SCC” is used within the product software code.

secure cryptographic coprocessor (SCC) application. (1) An application that runs within the IBM 4758 PCI Cryptographic Coprocessor. (2) Synonym for *agent*.

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.(T)

V

verb. A function possessing an entry_point_name and a fixed-length parameter list. The procedure call for a verb uses the syntax standard to programming languages.

W

Windows NT. A Microsoft operating system for personal computers.

Numerics

IBM 4758. IBM 4758 PCI Cryptographic Coprocessor.

Index

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W