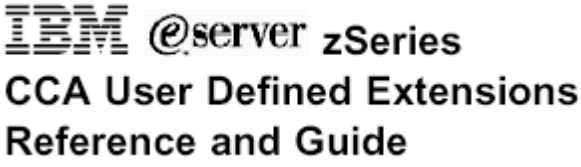
CCA User Defined Extensions

Reference and Guide



16-NOV-01, 14:51

CCA User Defined Extensions

Reference and Guide

Note!

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

First Edition (November, 2001)

IBM does not stock publications at the address given below. This and other publications related to the IBM 4758 Coprocessor can be obtained in PDF format from the Library page at *http://www.ibm.com/security/cryptocards*.

Reader’s comments can be communicated to IBM by using the Comments and Questions Form located on the product Web site at *http://www.ibm.com/security/cryptocards*, or you can respond by mail to:

Department VM9A, MG81/204-3
IBM Corporation
8501 IBM Drive
Charlotte, NC 28262-8563
U.S.A.

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

Contents

Figures

About This Book

The *IBM 4758 PCI Cryptographic Coprocessor CCA User Defined Extensions Reference and Guide*, Version 2: 4758-002 and 4758-023 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 PCI cryptographic coprocessor.
2. Functions that may be called by the portion of a UDX that runs on the host.
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 xiii and "Custom Software Publications" on page xiii.

Prerequisite Knowledge

The reader of this book should understand how to perform basic tasks (including editing, system configuration, file system navigation, and creating application programs) on the host machine and in the Windows NT environment, 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* and the *IBM 4758 PCI Cryptographic Coprocessor CCA Basic Services Reference and Guide*). The reader should also understand the OS/390 application environment (as described in the *OS/390 ICSF Application Programmer's Guide* and the *OS/390 ICSF System Programmer's 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 and Installing 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, “Cache Management Functions” describes the functions that a UDX can use to implement an on-board cache of secure data, and to track the contents of the cache on the host.

Chapter 12, “Miscellaneous Functions” describes several assorted utility functions available to a UDX.

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

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

Appendix F, “Moving a UDX from the Model 1 Card to the Model 2 Card” contains necessary changes to the UDX codewhen transferring code from a model 1 to a model 2 card.

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

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

Appendix I, “Notices” includes product and publication notices.

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

Typographic Conventions

This publication uses the following typographic conventions:

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

Related Publications

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

z/OS publications are available at *http://www.ibm.com/servers/eserver/zseries/zos/bkserv/*.

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

General Interest

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

- *IBM 4758 PCI Cryptographic Coprocessor General Information Manual* (version -01 or later)
- *IBM 4758 PCI Cryptographic Coprocessor Installation Manual*
- *z900 Support Element Operations Guide, SC28-6813, Version 1.7.2*

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*
- *z/OS Integrated Cryptographic Service Facility Application Programmer’s Guide*

Custom Software Publications

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

- *IBM 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 IBM Cryptographic Architecture (CCA) Application Program Interface (API).

- *IBM Transaction Security System General Information Manual*, GA34-2137
- *IBM Transaction Security System Basic CCA Cryptographic Services*, SA34-2362
- *IBM Transaction Security System 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
- *OS/390 Integrated Cryptographic Service Facility Overview*, GC23-3972
- *OS/390 Integrated Cryptographic Service Facility Application Programmer's Guide*, SC23-3976
- *OS/390 Integrated Cryptographic Service Facility System Programmer's Guide*, SC23-3974
- *OS/390 ICSF Trusted Key Entry Workstation User's Guide*, SC23-3978

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 2: 4758-002 and 4758-023. Revision bars (|) throughout this manual indicate updates.

Chapter 1. Understanding the UDX Environment

The *UDX Development Toolkit for S/390* The *UDX Toolkit* provides sample code, object modules, and macros that you can use to extend the IBM-developed Common Cryptographic Architecture (CCA) application program which employs the IBM zSeries 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 of ICSF and the zSeries cryptograpic engines, you should first become familiar with the information in the *OS/390 ICSF System Programmer’s Guide*, the *OS/390 ICSF Application Programmer’s Guide*, and the *IBM 4758 PCI Cryptographic Coprocessor CCA Basic Services Reference and Guide*.

The CCA architecture requires that security-sensitive functions be carried out in an environment where secret or private quantities can safely appear in the clear and where the design of the processing functions cannot be altered by an adversary. A coprocessor application program operates in such an environment. However, the confidentiality of secret or private quantities (for example, cryptographic keys or computational values) is also the responsibility of the application program design.

The CCA application operates as a request/response mechanism. Once initialized by CP/Q^{++} as a result of a PCI cryptographic 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, z/OS ICSF, 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”) is link-edited into a load module and installed in an APF authorized library. It executes in the ICSF address space along with the other ICSF callable services. The other (the “coprocessor piece”) is linked with a library containing IBM’s CCA coprocessor application modules, and loaded into the coprocessor. The host piece converts requests for service from the user’s application into messages to be sent to the coprocessor. These messages are received by the CCA application and routed to the appropriate (CCA or UDX) command processor.

[1] A few CCA verbs are implemented as subroutines in the top layer of CCA host code (z/OS ICSF) and do not send a request to the PCI Cryptographic Coprocessor.

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

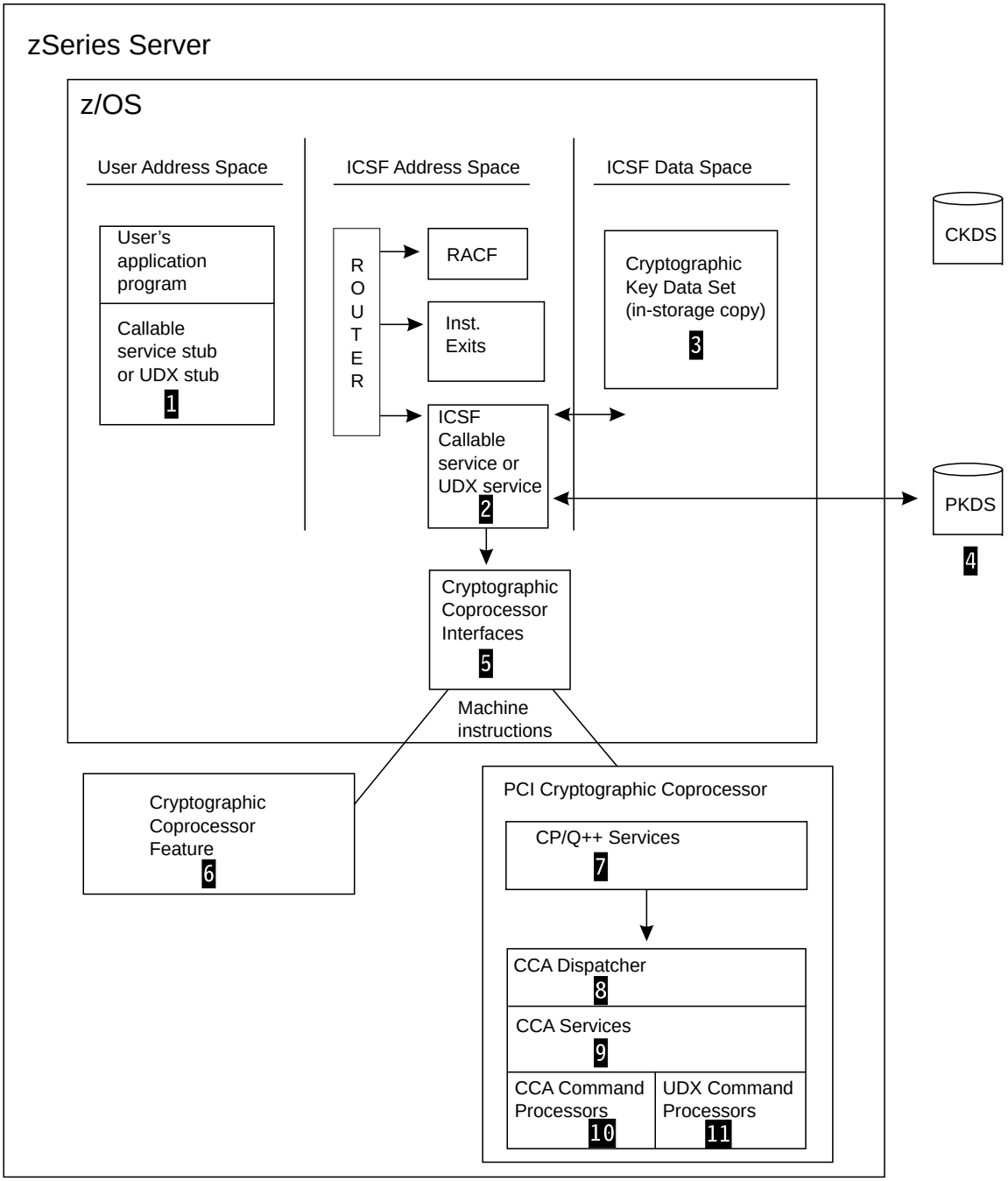


Figure 1-1. View of CCA with User-Defined Extensions

1 UDX service stub

The service stub connects the application program with the callable service. Each callable service which is part of ICSF has a service stub. Each UDX callable service must also have a related service stub. A callable service receives parameters from the application program when the program calls the service stub associated with the service. The service stub performs a space-switch PC (Program Call) operation to transfer control from the application's address space to ICSF's address space. The parameters that are associated with a callable service provide the only communication between the application program and ICSF.

2 Callable service

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

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

The fixed-length CPRB structure carries a primary function code, return and reason code values, and pointers to, and lengths of, the request and reply parameter blocks and data to be 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 tokens) in "key blocks"

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

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

The UDX callable services are analogous to the ICSF-provided CCA callable services. The UDX host-piece constructs and parses CPRB and request and reply parameter blocks using the same subroutines as employed by the ICSF callable service routines. Once the CPRB and request parameter block are constructed, the UDX callable service routine uses common subroutines to interface with the cryptographic coprocessor hardware (**5**). See Chapter 4, “Communications Functions” on page 4-1 for a description of the CSFADSPI (Communication interface between services and the coprocessor) function. See Appendix A, “UDX Sample Code - Host Piece - Service.”

3 Cryptographic Key Data Set (CKDS)

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

4 Public Key Data Set (PKDS)

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

5 Cryptographic Coprocessor Interfaces

ICSF provides service modules which interface with the cryptographic coprocessor hardware. The module which interfaces with the PCI cryptographic coprocessor is CSFACCPN. This module receives control from the callable service routines and examines the CPRB data passed as input to determine the nature of the call it will create to the PCI cryptographic coprocessor. The interface module makes use of machine instructions to cause zSeries Licensed Internal Code (LIC) to receive control to pass control via the PCI Bus to CP/Q^{++} in the PCI Cryptographic Coprocessor.

6 Cryptographic Coprocessor Feature

The Cryptographic Coprocessor Feature is a hardware feature available on the following zSeries servers: IBM zSeries Parallel Enterprise Server - Generation 3, IBM zSeries Multiprise 2000, zSeries G4 Enterprise Server, zSeries G5 Enterprise Server, zSeries G6 Enterprise Server, IBM @server zSeries. The Cryptographic Coprocessor Feature is secure, high-speed hardware which provides cryptographic functions. The Cryptographic Coprocessor Feature includes dual cryptographic coprocessor chips protected by tamper-detection circuitry and a cryptographic

battery unit. The callable services provided by ICSF utilize both the Cryptographic Coprocessor Feature (CCF) and the PCI cryptographic coprocessor to provide cryptographic functions to applications. The CCF, however, is not available for direct invocation by UDX callable services (other than by nested calls to other ICSF callable services). Cryptographic functions for UDX callable services are provided by the PCI cryptographic coprocessor.

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 cryptographic coprocessor interfaces layer and the CCA application. CCA host requests include the CPRB and request parameter block. The cryptographic coprocessor interfaces 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, that is, the default role.

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 command processor entry point. The dispatcher first checks the UDX entry point table for a match. If a match is not found, the dispatcher checks for a CCA command processor entry point. (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.

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 may be assigned, as required for security purposes, to authorize function within the command processor.

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 D, “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.
- Check that the caller is authorized to use this domain.
- Initialize the master key selector.
- 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.
- 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 are described in Appendix H of the *z/OS Integrated Cryptographic Service Facility Application Programmer's 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 host 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 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 utility routine `CSFACPRB`. This routine simplifies request/reply parameter block creation by accepting an arbitrary number of argument pairs (length + data pointer pairs) and constructs the sub-blocks in the previous table.

Similarly, while it is possible to extract data from the request/reply parameter blocks “by hand” using pointer arithmetic, it is recommended that the UDX developer instead use the utility routines `CSFAVLPB` (Validate a CPRB) and `CSFAPBLK` (Parse a CPRB).

Note: An example of the use of these functions is in Appendix A, “UDX Sample Code - Host Piece - Service” 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 CSFADSPI (Communication interface between services and the coprocessor) or the CSFACPRB (Build a CPRB) function followed by the CSFACCPN (Send a Request to the Coprocessor) function. 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_REQUEST_DATA or CPRB_REPLY_DATA as buffer indices.

On the host:

```
/* First, set the CPRB structures properly, with the Rule Array, Verb Unique Data, and Key Blocks */
/* using the CSFACPRB or CSFADSPI function.                                                       */
/* (Alternatively, you can use the CSFADSPI function, in which case you will not also use the     */
/* CSFACCPN function.                                                                             */

   CALL CSFACPRB( return_code,
                  flags,
                  subfunction_code,
                  rule_count,
                  rules,
                  number_vuds,
                  vud_list,
                  number_keys,
                  key_list,
                  request_data_block_length,
                  request_data_block,
                  reply_data_block_length,
                  reply_data_block,
                  request_CPRB_length,
                  request_CPRB,
                  reply_CPRB,
                  SPB);

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

       CCNPPTR = ADDR(INPUT_DATA_BLOCK);
       CCNP_ALET = PRIMALET;                          /* Primary ALET    */
       CCNP_ADDRESS = REQUEST_DATA)BLOCK;             /* Address of data */
       CCNP_LENGTH = REQUEST_DATA_BLOCK_LENGTH;       /* Length of data  */


       CCNPPTR = ADDR(OUTPUT_DATA_BLOCK);
       CCNP_ALET = PRIMALET;                          /* Primary ALET    */
       CCNP_ADDRESS = REPLY_DATA_BLOCK;               /* Address of data */
       CCNP_LENGTH = REPLY_DATA_BLOCK_LENGTH;         /* Length of data  */

       CALL   CSFACCPN( RETURN_CODE,
                        REASON_CODE,
                        REQUEST_CPRB,
                        REQUEST_CPRB_LENGTH,
                        REPLY_CPRB
                        REPLY_CPRB_LENGTH,
                        PRIMALET,
                        PCICC_INDEX,
                        PCICC_SERIAL_NUMBER,
                        ACCPN_BLANK_ID,
                        ACCPN_SHARED,
                        ACCPN_DOMAIN_NOT_APPLIC
                        SUBFUNCTION_CODE,
                        INPUT_DATA_BLOCK,
                        OUTPUT_DATA_BLOCK,
                        SPB);
```

```
//  First, set the CPRB structures properly, with the Rule Array, Verb Unique Data, and Key Blocks
using the CSFACPRB or CSFADSPI function.
// To set the Request Data Block:
   LocalRequestTextLength = *pTextLength;
   LocalReplyTextLength   = *pTextLength;

   CSUC_BULDCPRB( pCprb,
                  (UCHAR *) ESSS_FUNCTION_ID_S,

                  RequestBlockLength,              // Req.Parm
                  pRequestParmBlock,               //   block
                                                   //   len + adr

                  LocalRequestTextLength,          // Req.Data
                  (UCHAR *) pInpText,              //   block
                                                   //   len + adr

                  sizeof( pRequestReplyBuffer->reply_buf ),
                  pRequestReplyBuffer->reply_buf,

                  LocalReplyTextLength,            // Rep.Data
                  (UCHAR *) pOutText);             //   block
```

On the card:

```
// -----------------------------------------
   // Get the length of the bulk text first, from
   // the CPRB structure.
   // -----------------------------------------

   BulkBlockLength = pRequestCprb->req_data_block_length;

   // -----------------------------------------
   // Check that the length of the reply data block
   // in the CPRB is long enough (depends on your function)
   // -----------------------------------------

   if (BulkBlockLength > pRequestCprb->reply_data_block_length )
   {
      Cas_proc_retc( pReplyCprb, RT_CONSISTENCY_ERROR );
      return;
   }
   // -----------------------------------------
   // Get the InpText
   // -----------------------------------------

   // It is best to allocate these large blocks of data dynamically.
   // But don't forget to free them later!
   InpTxt = malloc(BulkBlockLength);
   if (InpTxt == NULL)
   {
      Cas_proc_retc(pReplyCprb, E_ALLOCATE_MEM);
      return;
   }
   memset(InpTxt,255,sizeof(InpTxt));

   // Get the data from the buffer.
   ReturnMsg = sccGetBufferData( RequestId,
                                 CPRB_REQUEST_DATA,
                                 InpTxt,
                                 BulkBlockLength);

   if (ReturnMsg != 0)
   {
      free(InpTxt);
      Cas_proc_retc( pReplyCprb,
                     RT_CONSISTENCY_ERROR  );
      return;
   }  // End if
   //-------------------------------------------------------------
   //
   //  Build the OutTxt
   //  after completing the function
   //  and filling the Reply CPRB with the correct information
   //-------------------------------------------------------------

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

```
for (iCnt=0 ; iCnt < BulkBlockLength ; iCnt++)
{
   OutTxt[iCnt] = InpTxt[BulkBlockLength-iCnt-1];
}  // End for

 free(InpTxt);

// The data we return is the same length as the data which
// was sent, for this function.
ReturnMsg = sccPutBufferData( RequestId,
                              CPRB_REPLY_DATA,
                              OutTxt,
                              BulkBlockLength);
free(OutTxt);
if (ReturnMsg != 0)
{
   Cas_proc_retc( pReplyCprb,
                  RT_CONSISTENCY_ERROR  );
   return;
}  // End if
//-------------------------------------------------------------
// Write the Length of OutTxt in the CPRB
// -------------------------------------------------------------

pReplyCprb->reply_data_block_length = BulkBlockLength;
//-------------------------------------------------------------
// Then return to the host function
//-------------------------------------------------------------
     return;
```

Refer to the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference* for more details on using the sccGetBufferData and sccPutBufferData functions.

Key Blocks

The key blocks portion of the request and reply parameter blocks is used to transport zero or more key identifiers (key tokens). A key block is a data structure consisting of a header and appended 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 which follows it. For zSeries, the *Flags* field should be set to binary zeroes.

Chapter 2. Building and Installing a CCA User-Defined Extension

Building a CCA UDX for zSeries is not a customer function. The steps followed during UDX development and test are outlined below for your information. See “Installing the Host Piece of a UDX” on page 2-3 and “Installing the Coprocessor Piece of a UDX” on page 2-9 for the steps required to install a zSeries UDX.

Host Piece of a UDX

Building the Host Piece of a UDX

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

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

The steps a developer performs in order to create the host piece of the UDX are as follows:

1. Define the UDX API.
2. Define the subfunction code for the UDX.
3. Define new completion codes for the UDX.
4. Update the appropriate macros supplied with the *UDX Development Toolkit for zSeries* with the UDX subfunction code and completion codes.
5. Design and code the logic of the UDX callable service.
6. Code the UDX service stub.
7. Compile the UDX callable service and service stub.

This section lists the steps an IGS developer performs in order to create the host piece of the zSeries UDX. More development detail is provided in this draft material than is provided in the customer version.

1. Define the UDX API.

   The definition of the UDX API will most likely be a joint effort between IGS developers and the customer’s technical team. IGS will work with the customer to understand the requested function, and to develop the API for the UDX

callable service. It is possible that the customer will defer the definition of the API totally to the IGS development team.

The UDX API may have any number of parameters, although to make use of ICSF's CSFVRGEN macro it is more convenient if the number of parameters is less than 20. Callable service parameters are positional, and must be specified even if not used. For consistency, it is recommended that all UDX functions include the following six parameters as the first six parameters of the API: `return_code`, `reason_code`, `exit_data_length`, `exit_data`, `rule_array_count`, and `rule_array`.

2. Define the subfunction code for the UDX.

   There will be one subfunction code associated with the UDX command processor (the coprocessor piece of the UDX). The following 2-character code points have been reserved for CCA extensions. You should not use other code points as they may conflict with existing CCA commands.

   WA - WZ, W0 - W9
   XA - XZ, X0 - X9 (reserved for customer-written UDXs)
   YA - YZ, Y0 - Y9 (reserved for customer-written UDXs)

   must be added to the macro CSFCPRB -- add the subfunction in the form

   ```
   CCP_FUNCTION_KEY_GENERATE          CONSTANT('4B47'X)     /* KG */
   ```

   (that is, in "big-endian" form). The CSFCPRB macro must be INCLUDEd (%INCLUDE SYSLIB(CSFCPRB);) in the host callable service module.

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 should be defined in the *cxt_cmds.h* header file that is `#include`d in the coprocessor portion of the UDX (the UDX command processor). Completion codes for ICSF code are defined in the macros CSFSDCL and CSFFDCL. In most cases there will be no need to add a new completion code to the ICSF macro(s). The only reason to add it would be to facilitate referencing the completion code via a variable name in the host portion of the UDX.

4. Update the appropriate macros with the UDX subfunction code and completion codes (if necessary).

   The UDX subfunction code is defined in macro CSFCPRB; completion codes are defined in macro CSFFDCL.

5. Design and code the logic of the UDX callable service.

   The host piece of a UDX is typically straightforward - it essentially constructs a request block, sends the block to the coprocessor, and parses the result. See Appendix A, "UDX Sample Code - Host Piece - Service" on page A-1 for a sample (*UDXKEN1.PLX*). 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.

6. Code the UDX service stub.

   Besides writing the callable service itself, you must write a service stub, which is the connection between the application program and the UDX callable service. The application program calls the service stub, which accesses the callable service. The service stub can be identified by any name you choose to call it. All callable service stubs for callable services which execute in ICSF's address space look identical. The ICSF group will provide a sample stub which may be copied to create a new UDX service stub.

7. Compile the UDX callable service and service stub.

   The UDX callable service and service stub are both coded in PL/X, and should be compiled with the PL/X compiler at the ???.?? level. The macro library concatenation for the compile and assemble steps will be provided by the ICSF group. (The macro library concatenation is dependent upon the level of OS/390 on the zSeries system where the UDX is to be installed.)

End IGS Information

This ends the development information.

Installing the Host Piece of a UDX

You will receive several files from IBM which must be installed on the zSeries host and the ICSF Installation Options data set must be customized in order to use your UDX. The files and the steps to be followed are specified below. (The *OS/390 ICSF System Programmer's Guide* may also provide valuable information about steps required for UDX installation.)

1. The OBJ file for the UDX callable service must be link-edited into a load module and installed into an APF authorized library. ICSF uses the normal OS/390 search order to locate the service:

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

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

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

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

   where service_stub_name is the name of the service stub for the UDX callable service and service_parameters are the parameters you want to pass to the UDX callable service. You supply the parameters according to the syntax of the programming language that you use to write the application program.

3. You must identify the UDX service in the ICSF Installation Options Data Set using the UDX keyword. For information about the specification of the UDX keyword, refer to the *OS/390 ICSF System Programmer's Guide*. You will specify information including the UDX subfunction code, a service number, and a load module name.

4. If you received a post-processing exit for the CSFPCI callable service (because you requested that your UDX have an access control point), the exit must be installed. Link-edit the OBJ file into a load module, and install the load module into an APF-authorized library. ICSF uses the normal OS/390 search order to locate the service:

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

   The ICSF Installation Options Data Set must be updated to define the exit. Use the EXIT keyword, specifying “CSFPCI” for the ICSF name of the callable service exit. For information about the specification of the EXIT keyword, refer to the *OS/390 ICSF System Programmer’s Guide*.

5. You will receive a password associated with the UDX which has been created for you. You use the password to authorize the UDX on one or more PCI cryptographic coprocessors. Use the ICSF “Authorize a UDX” panel which is selectable from the “User Defined Extension Management” panel.

Coprocessor Piece of a UDX

Building the Coprocessor Piece of a UDX

The coprocessor piece of a UDX is a command processor that is linked with IBM's CCA coprocessor application modules to create an executable that is loaded into the coprocessor. The coprocessor piece of a UDX may invoke any of the CCA services and can also invoke CP/Q++ functions.

The steps a developer must complete in order to create the coprocessor piece of a UDX are as follows:

1. Define the UDX command processor API.
2. Define access control points for the UDX.
3. Define new completion codes for the UDX.
4. Define the subfunction code for the UDX.
5. Add the UDX command processor to the command decoding array.
6. Design and code the logic of the coprocesor piece of the UDX.
7. Build the UDX coprocessor executable.

This section lists the steps an IGS developer performs in order to create the coprocessor piece of the zSeries UDX. More development detail is provided in this draft material than is provided in the customer version.

1. Define the UDX command processor API.

   A prototype for each command processor the coprocessor piece of the UDX makes available to the host piece of the UDX must be placed in a header file (for example, *cxt_cmds.h*) that is `#included` by the command processor. The prototype must have the same parameters and return type as the example shown in Figure 2-1 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) 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 2-1. 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 created by the host piece of the UDX which 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 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 `#included` by the command processor piece of the UDX. Figure 2-2 on page 2-7 contains an example of such a definition.

   A command processor can use access control points in conjunction with the role identifier supplied as an argument to the command processor to determine

[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.

whether or not a particular operation is authorized. See “check_access_auth_fcn - Verify User Authority” on page 12-2 for details.

```
/*****************************************************************
** 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 2-2. Example UDX Access Control Points

Notes:

a. When writing UDXs for zSeries customers, IGS should define access control points from the range 0xF000-0xFFFF. The range 0x8000-0xEFFF should be reserved for customer-written UDXs.

b. For zSeries, the access control point for a UDX should never be enabled by IGS in the DEFALTxx role. Enablement of a UDX access control point requires a TKE.

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 should be defined in a header file (for example, *cxt_cmds.h*) that is `#included` in the coprocessor portion of the UDX (the UDX command processor).

4. Define the subfunction code for the UDX.

   There will be one subfunction code associated with the UDX command processor (the coprocessor piece of the UDX). The following 2-character code points have been reserved for CCA extensions. You should not use other code points as they may conflict with existing CCA commands.

   WA - WZ, W0 - W9
   XA - XZ, X0 - X9 (reserved for customer-written UDXs)
   YA - YZ, Y0 - Y9 (reserved for customer-written UDXs)

   The subfunction code must be defined in a header file (for example, *cacuatm.h*). The definition must be in “little-endian” form:

   ```
   #define  SSKEYG_ID       0x474B   /* 'KG', Generate Key            */
   ```

5. Add the UDX command processor to the command decoding array.

   IBM’s CCA coprocessor application modules use 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 the coprocessor piece 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 2-3 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 2-3. Example UDX Command Decoding Array Definition

6. 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 (*udx_ken1.c*) is in Appendix D, “UDX Sample Code - Coprocessor Piece” on page D-1. It can be used as a skeleton and customized to meet the requirements of most UDXs.

7. Construct the UDX Authority Table (UAT)

   Generate a password associated with the UDX. You can use a random number generator to generate an 8-byte random number. (This password must be saved and communicated to the UDX owner.) Hash the password using SHA-1. To add a new UDX to the UDX Authority Table, modify the *cacuatm.h* part. Add a UDX entry consisting of the UDX subfunction code, the SHA-1 hash of the UDX password, and the default authority state ('N') to *cacuatm.h*. Increment the number of entries constant (UAT_ENTRY_NUMBER). When there are any changes to the UAT, the UDX list version number (UDX_LIST_VERSION) must be incremented once for the set of changes.

8. Build the UDX coprocessor executable.

   The *UDX Development Toolkit for zSeries* includes a sample makefile (*S390NT.mak*) for Windows NT. Statements should be added to compile the source files that contain the coprocessor piece of the UDX. The UDX files should be compiled with the `S390` option and with debug options.

End IGS Information

This ends the development information.

(*S390NT.mak*)

Installing the Coprocessor Piece of a UDX

The UDX coprocessor executable file will be incorporated into the coprocessor segment three image and signed by IBM. The image will be made available via the zSeries LIC patch and driver process. To install the UDX command processor on your coprocessor, install the appropriate level of zSeries LIC. The segment three image containing the UDX command processor will be loaded into the PCI cryptographic coprocessor at the next coprocessor reset.

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 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 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*.

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.
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 on the coprocessor, your program must include the following header files.

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

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

```
%INCLUDE SYSLIB(CSFGSVT);        /* Generic service vector table    */
```

Summary of Functions

Request and reply processing includes the following functions.

On the zSeries Host

CSFACKDS Access the in-storage ICSF Cryptographic Keys Data Set.

CSFAPKDS Access the ICSF Public Key Data Set.

CSFACCPN Send a request to the coprocessor.

CSFACPRB Build a CPRB.

CSFADSCP Destroy a CPRB.

CSFAVLPB Validate a CPRB.

CSFAPBLK Parse a CPRB.

CSFAPKTV Validate/initialize an RSA/DSS key token.

CSFADSPI Communication interface between services and the coprocessor.

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

On the Coprocessor

BuildParmBlock Build a parameter block.

Cas_proc_retc Prioritizes a return code in the reply CPRB.

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.

Refer to the *z/OS Integrated Cryptographic Service Facility Application Programmer's Guide*, SC23-3976 located on the OS/390 publications Web site (*http://www.ibm.com/servers/eserver/zseries/zos/bkserv/*) for general information about ICSF callable services and common parameters.

CSFACKDS - Access ICSF Cryptographic Keys Data Set

Note: This function is available on the host.

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

Function Prototype

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

Input

On entry to this routine:

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

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

`entry_code` is an integer that represents the 4-byte hexadecimal value containing the entry code. Possible values are:

X'00000001' (TOKEN)
Retrieve a token from the in-storage CKDS. (If the token is not found, return a return code rather than abending.)

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

`key_type` is a character string containing 8 EBCDIC characters specifying the type of key record to be processed.

ANY for generic retrieval.

Otherwise, allowable key types are:

DATA
DATAXLAT
EXPORTER
IMPORTER
IPINENC
MAC
MACVER
NULL
OPINENC
PINGEN
PINVER

CV

`output_area` is a character string which is to contain the key record returned.

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

Output

On successful exit from this routine:

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

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

`output_area` is a character string containing the actual key record returned.

Return and Reason Codes

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

Return Code (dec)	Reason Code	Meaning
0	0	The operation was successful.
8	10012	Key not found.
8	16004	Request failed by RACF.
8	16020	Function not allowed for system key.
12	0	CSF not active.
12	12	Exit has failed.
12	10020	MAC failed.
12	10024	Key failed by installation exit.
12	10036	Label not unique.
12	10052	No space in CKT for dynamic adds.
16	4	Your call to an ICSF callable service resulted in an abnormal ending.

Abend Code	Reason Code	Meaning
x'18F'	160 256 258	Invalid entry code Invalid return code from exit Invalid return code from CSFPCMF

CSFAPKDS - Access ICSF Public Key Data Set

Note: This function is available on the host.

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

Function Prototype

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

Input

On entry to this routine:

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

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

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

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

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

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

READ	Read record
CREATE	Create record
UPDTENUL	Replace null token
UPDATE	Update token
DELLABEL	Delete record
DELTOKEN	Replace token with nulls

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

Output

On successful exit from this routine:

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

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

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

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

Return and Reason Codes

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

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

Abend Code	Reason Codes	Meaning
x'18F'	174	Unknown FUNCTION code in the parameter list.

CSFACCPN - Send a Request to the Coprocessor

Note: This function is available on the host.

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

Function Prototype

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

Input

On entry to this routine:

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

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

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

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

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

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

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

`identifier` is a character string containing 8 bytes of binary zeroes.

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

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

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

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

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

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

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

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

Output

On successful exit from this routine:

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

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

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

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

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

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

`output_data_block` is a character string containing 20 bytes of zeroes if there is no reply data block, or else it is a character string containing the reply data block.

Return and Reason Codes

In general, the return and reason codes from this function will have been generated by the coprocessor, in the course of completing the request identified by *pcifunction*.

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

Return Code (dec)	Reason Code	Meaning
0	0	The operation was successful.
8		Application error.
12	0	CSF not active.
12	X'2B28'	Service has failed.
12	X'2B34'	Crytographic coprocessor is not available.
16	4	Unrecoverable failure in this routine.
16	4	Your call to an ICSF callable service resulted in an abnormal ending.

Abend Code	Reason Code	Meaning
x'18F'	80 165 258 267 268 408	NQ incomplete. Error in the interface to the coprocessor. Bad internal parameters between internal service calls Invalid condition code from the CSF (instruction) macro Full CCP queue CSFACCPS Bad response code DQ/NQ/PQ AP PSMID in returned message does not match sent message

CSFACPRB - Build a CPRB

Note: This function is available on the host.

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

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

Function Prototype

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

Input

On entry to this routine:

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

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

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

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

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

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

Offset	Length	Description
0	4	length of verb unique data field
4	2	*flag1* may be used to indicate the type of data
6	2	*flag2* may be used to indicate the type of data
8	4	Address of the data item to be added

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

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

Offset	Length	Description
0	4	Length of key token
4	2	*flag* used to indicate the type of data
6	2	*reserved*
8	4	Address of key token to be added

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

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

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

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

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

Output

On successful exit from this routine:

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

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

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

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

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

`reply_data_block` is a character string containing the address of the reply data block.

Return and Reason Codes

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

Return Code (dec)	Reason Code	Meaning
0		The operation was successful.
4		Too much data for the request parameter block.
16	4	Your call to an ICSF callable service resulted in an abnormal ending.

Abend Code	Reason Code	Meaning
x'18F'	178	Unable to obtain storage for CPRB

CSFADSCP - Destroy a CPRB

Note: This function is available on the host.

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

Function Prototype

```
call CSFADSCP
( return_code,
  SPB )
```

Input

On entry to this routine:

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

Output

On successful exit from this routine:

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

Return and Reason Codes

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

Return Code (dec)	Reason Code	Meaning
0		The operation was successful.
4		No address in the SPB.
16	4	Your call to an ICSF callable service resulted in an abnormal ending.

Abend Code	Reason Code	Meaning
X'18F'	180	Invalid pointer in SPB for CPRB

CSFAVLPB - Validate a CPRB

Note: This function is available on the host.

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

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

Function Prototype

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

Input

On entry to this routine:

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

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

Output

On successful exit from this routine:

`return_code` is the general result of the callable service.

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

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

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

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

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

Return and Reason Codes

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

Return Code (dec)	Reason Code	Meaning
0		The operation was successful.
2		The CPRB is not valid.
4		The parameter block is not valid.
6		No CPRB found at this address.
16	4	Your call to an ICSF callable service resulted in an abnormal ending.

Abend Code	Reason Code	Meaning
X'18F'	179 180 183	Domain in CPRB doesn't match CCVE Invalid pointer in SPB for CPRB Reply CPRB or parameter block is bad

CSFAPBLK - Parse a CPRB

Note: This function is available on the host.

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

Function Prototype

```
call CSFAPBLK
( return_code,
  block_address,
  element_ptr,
  element_length,
  element_flag,
  element_data_ptr,
  SPB )
```

Input

On entry to this routine:

`block_address` is a character string containing the address of the block to be parsed.

`element_ptr` is a character string containing the address of the last record found in the block, or null if the first record in the block is to be found.

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

Output

On successful exit from this routine:

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

`element_ptr` is a character string containing the address of the next record found in the block.

`element_length` is an integer that represents the length of the data returned at the address specified by the *element_data_ptr* parameter.

`element_flag` is a character string containing the flag from the record found.

`element_data_ptr` is a character string containing the address of the data from the record found.

Return and Reason Codes

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

Return Code (dec)	Reason Code	Meaning
0	0	The operation was successful.
1		The operation was successful. There are no more records in the block.
4		The last record address is not valid.
6		The block is not valid.
16	4	Your call to an ICSF callable service resulted in an abnormal ending.

Abend Code	Reason Code	Meaning
X'18F'	180	Invalid pointer in SPB for CPRB

CSFAPKTV - Validate/Initialize an RSA or DSS Key Token

Note: This function is available on the host.

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

Function Prototype

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

Input

On entry to this routine:

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

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

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

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

Output

On successful exit from this routine:

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

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

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

Return and Reason Codes

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

Return Code (dec)	Reason Code	Meaning
0	0	The operation was successful.
8	2040	Invalid first byte (must be X'1E' or X'1F').
8	11012	Invalid flags in the token.
8	11016	Invalid hash value.
8	11020	Token too old—invalid RMK or SMK hash.

Return Code (dec)	Reason Code	Meaning
8	11024	Token missing one or more required parts.
8	11044	Invalid PKA token.
8	11092	Invalid PKA subsection or TLV.

CSFADSPI - Communication Between Services and Coprocessor

Note: This function is available on the host.

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

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

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

Function Prototype

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

Input

On entry to this routine:

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

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

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

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

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

invoking_alet is an integer that represents the ALET that is used in the call to CSFACCPN and also when building the Data Block parameter for the call to CSFACCPN. Specify zero.

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

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

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

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

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

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

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

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

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

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

`request_data_block_length` is an integer that represents the length of the request data block. storage allocated for the reply.

`request_data_block` is a character string containing the Request Data Block. Specify a blank character if the *request_data_block_length* specifies a length of zero.

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

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

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

Output

On successful exit from this routine:

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

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

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

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

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

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

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

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

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

Return and Reason Codes

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

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

Abend Code	Reason Code	Meaning
X'18F'	165	Bad internal parameters between internal service calls

CSFASEC - Check Authorization

Note: This function is available on the host.

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

Function Prototype

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

Input

On entry to this routine:

`resource` is a character string containing the name of the resource to be authority-checked.

`resource_length` is an integer that represents the length of the resource name.

`resource_class` is a character string containing 8 EBCDIC characters, CSFKEYS or CSFSERV.

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

Output

On successful exit from this routine:

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

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

Return and Reason Codes

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

Return Code (dec)	Reason Code	Meaning
0	0	The operation was successful.
8	16000	Authorization failed.

BuildParmBlock - Build a Parameter Block

Note: This function is available on 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.

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.

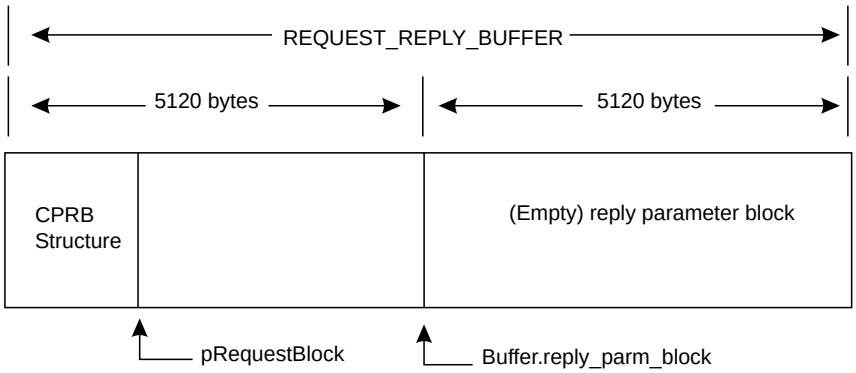
`Data`*i*`_length` is the length of the *i*th. item, in bytes.

`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.

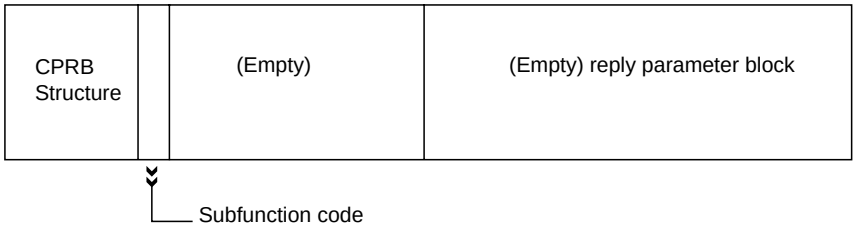
If 2 or more items of verb unique data are to be added, each item should be preceded by a short field containing the length of the individual item +2. This will allow the function FindNextDataBlock to parse the result.

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



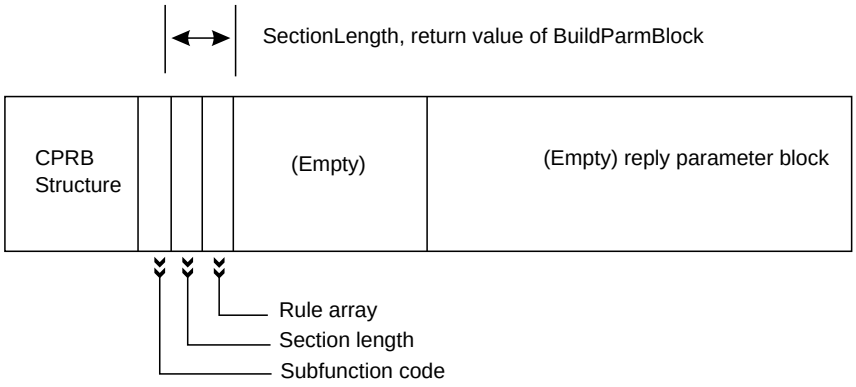
Step one: add the subfunction code

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



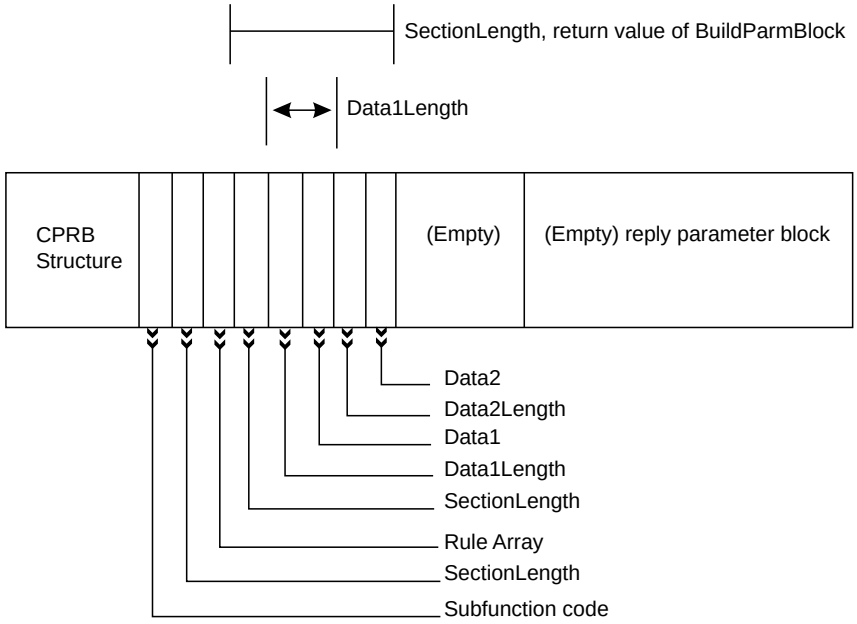
Step two: add the rule array

```
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), &Data1Length, /* length of 1st item, including this field */
Data1Size, pData1,
sizeof(short), &Data2Length, /* length of 2nd item, including this field */
Data2Size, pData2);
```



Step four: add the key blocks

```
KeyHeaderI.Length = KeyTokenLength + sizeof(KEY_FIELD_HEADER);
KeyHeaderI.Flags = storageOptions;

BlockLength = BuildParmBlock(pRequestBlock + BlockLength,
                        2,   /*adding a key block header and a key token*/
                        sizeof(KEY_FIELD_HEADER), &KeyHeaderI,
                        KeyTokenLength, &KeyToken):
```

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 and Reason Codes

This function has no return codes.

Notes

Building the Parameter Blocks

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

The rule array is a byte array, with 8 bytes for each rule present. Each rule is 8 bytes long, padded on the right with spaces. It is important to note that the entire 8 bytes are compared - these are not strings as C and C++ define them. No allowance is made for a null terminator, so be careful when copying rule data into the array. 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 1-12.

See Appendix A, UDX Sample Code - Host Piece - Service 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 and Reason Codes

Common return codes generated by this routine are:

OK The operation was successful.

ERROR The return code in `msg` was greater than Warning level (level 4).

FindFirstDataBlock - Search for Address of First Data Block

Note: This function is available on 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 and Reason Codes

This function has no return codes.

FindNextDataBlock - Search for Address of Next Data Block

Note: This function is available on 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 and Reason Codes

This function has no return codes.

find_first_key_block - Search for First Key Data Block

Note: This function is available on 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 and Reason Codes

This function has no return codes.

find_next_key_block - Find Address of Next Key Data Block

Note: This function is available on 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 and Reason Codes

This function has no return codes.

InitCprbParmPointers - Initialize CPRB Parameter Pointers

Note: This function is available on the coprocessor.

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

Function Prototype

```
void InitCprbParmPointers
(
  CPRB_structure     *pInputCprb,
  CPRB_structure     *pOutputCprb
)
```

Input

On entry to this routine:

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

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

Output

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

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

Return and Reason Codes

This function has no return codes.

keyword_in_rule_array - Search for Rule Array Keyword

Note: This function is available on 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 4-36 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 and Reason Codes

This function has no return codes.

parm_block_valid - Examine and Verify a Parameter Block

Note: This function is available on the coprocessor.

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

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

Function Prototype

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

Input

On entry to this routine:

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

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

Output

On successful exit from this routine:

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

Return and Reason Codes

This function has no return codes.

rule_check - Verify Rule Array

Note: This function is available on 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 and Reason Codes

Common return codes generated by this routine are:

E_RULE_ARRAY_KWD	Indicates that a required rule array keyword was missing. This also applies if only one keyword must be present out of a group of keywords, but none from the group are in your rule array.
E_RULE_ARRAY_COMBINE	Indicates that a rule array keyword appears more than one time in the input rule array. It can also indicate that more than one keyword appears from a group, where only one from the group is supposed to be present.

Examples

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

Checking the Rule Array for Verb CSNBPKI

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

- FIRST
- MIDDLE
- LAST

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

```
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 */
```

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.

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

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

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

Overview of the Coprocessor 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 two sets of master keys, one set for PKA keys (ASYM_MK) and a second set for DES keys (SYM_MK). Each set consists of three master key values.

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

zSeries supports sixteen cryptographic domains. There is a separate set of master keys for each domain (that is, associated with each domain is an old, current, and new master key value for PKA keys (ASYM_MK) and for DES keys (SYM_MK)). The *mk_set* parameter of the *mk_selectors* variable type identifies the set of master keys to be processed by a function.

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

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

  For the SYM_MK master key, if the first and third key parts are the same, this value is calculated using the z/OS ICSF algorithm. Otherwise, this value is calculated using SHA-1. For the ASYM_MK master key, two verification values are stored. One value is calculated using SHA-1 and one value is calculated using MDC-4. (For zSeries, only the MDC-4 verification value is stored.)

- The **status** of the key. For the CMK and the OMK, two status values are possible.

  – The register contains a valid key value.
  – The register does not contain a valid key value.

  For the NMK register, three status values are possible.

  – The register is empty. It does not contain any portion of a new master key value.
  – The register is partially full. The last key part has not yet been combined into the value in the register.
  – The NMK register is full. All key parts have been combined to form the final key value.

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

Location of the Master Keys

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

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

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 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, specified by the mk_selectors variable. (See below.) If no error occurs opening the SRDI, then error code *mk_NO_ERROR* is returned to the caller. Otherwise, error code *mk_SRDI_OPEN_ERROR* is returned.

Required Variables

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

- The master key set (`mk_set`) that specifies which set of master keys is to be accessed, for environments where more than one set of master keys may exist. If more than one set of master keys exist, the mk_set is identified by the cryptographic domain. The domain number serves as an index to the Master Key SRDI, which is an array of sixteen sets of master keys. In environments which support more than one master key set, the cryptographic domain is passed by the host to the coprocessor in the CPRB. Where there is only one set of master keys, mk_set 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 master key type (`type_mks`) which defines the type of key to be encrypted with this set of master keys. This variable can be any of ASYM_MK, SYM_MK, or BOTH_MK.

The following functions are summarized in this chapter.

Functions to Check Master Key Values and Status

Summary of Functions

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

mkmGetMasterKeyStatus - Get Master Key Status

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

Function Prototype

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

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

Input

On entry to this routine:

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

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

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 to be queried.
- `type_mks` should be set to ASYM_MK if the request is to get the verification pattern of the asymmetric-keys master key or to SYM_MK if the request is to get the verification pattern of the symmetric-keys master key.

Output

On successful exit from this routine:

`ver_pattern` contains the verification pattern.

Return Codes

Common return codes generated by this routine are:

mk_NO_ERROR	The operation was successful.
mk_INVALID_KEY_SELECTOR	The input parameters are not valid.
mk_SEM_CLAIM_FAILED	Unable to access the SRDI Manager.
mk_KEY_NOT_VALID	The selected key was not in a valid state.

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.
TDESDecryptUnderMasterKey	Triple-DES decrypts data using the master key.
TDESEncryptUnderMasterKey	Triple-DES encrypts data under the master key.
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 to be used for decryption.
- `type_mks` should be set to ASYM_MK if the data is to be decrypted with the asymmetric-keys master key or to SYM_MK if the data is to be decrypted with the symmetric-keys master key.

`cleartext` is a pointer to a buffer large enough to store the `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 to be used for encryption.
- `type_mks` should be set to ASYM_MK if the data is to be encrypted with the asymmetric-keys master key or to SYM_MK if the data is to be encrypted with the symmetric-keys master key.

`cleartext` is a pointer to a buffer containing the data to be enciphered.

`ciphertext` is a pointer to a buffer large enough to store the `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.

TDESDecryptUnderMasterKey

TDESDecryptUnderMasterKey decrypts data which has been encrypted with a master key using the Triple DES algorithm. The master key which was used for encryption must be specified, and the cipher text provided must be a multiple of 8 bytes long.

Function Prototype

```
long mkmTDESDecryptUnderMasterKey(mk_selectors  MKSelector,
                                  UCHAR        *cipher_text,
                                  UCHAR        *clear_text,
                                  ULONG         text_length);
```

Input

On entry to this routine:

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

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- `mk_register` is set to either old_mk, current_mk, or new_mk, representing the key to be used for decryption.
- `type_mks` should be set to ASYM_MK if the data is to be decrypted with the asymmetric-keys master key or to SYM_MK if the data is to be decrypted with the symmetric-keys master key.

`cipher_text` is a pointer to a buffer which contains the encrypted text. This text must be a multiple of 8 bytes long.

`clear_text` is a pointer to a buffer to hold the decrypted text. This buffer must be as long as `text_length`.

`text_length` is the number of bytes of data in the `cipher_text` buffer. This is also the number of bytes of encrypted text returned in the `clear_text` buffer.

Output

On successful exit from this routine:

`clear_text` contains `text_length` bytes of data decrypted from the `cipher_data` using the Triple DES algorithm.

Return Codes

Common return codes generated by this routine are:

mk_NO_ERROR	The operation was successful.
mk_INVALID_DATA_LENGTH	The data length is not a multiple of eight.
mk_SRDI_OPEN_ERROR	Could not open the master key SRDI.
mk_SEM_CLAIM_FAILED	Unable to access the SRDI Manager.
mk_KEY_NOT_VALID	The designated master key is not valid.
mk_INVALID_KEY_SELECTOR	The input parameters are not valid.

TDESEncryptUnderMasterKey

TDESEncryptUnderMasterKey takes a variable amount of data in a multiple of 8 bytes and encrypts it using the Triple DES algorithm.

Function Prototype

```
long mkmTDESEncryptUnderMasterKey(mk_selectors  MKSelector,
                                  UCHAR        *clear_text,
                                  UCHAR        *cipher_text,
                                  ULONG         text_length);
```

Input

On entry to this routine:

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

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- `mk_register` is set to either old_mk, current_mk, or new_mk, representing the key to be used for encryption.
- `type_mks` should be set to ASYM_MK if the data is to be encrypted with the asymmetric-keys master key or to SYM_MK if the data is to be encrypted with the symmetric-keys master key.

`clear_text` is a pointer to the text which you wish to encipher. This text must be a multiple of 8 bytes long.

`cipher_text` is a pointer to a buffer in which to store the encrypted text. This buffer must be at least as long as the `clear_text`.

`text_length` is the number of bytes of data in the `clear_text` buffer. This is also the number of bytes of encrypted text returned in the `cipher_text` buffer.

Output

On successful exit from this routine:

`cipher_text` contains `text_length` bytes of data encrypted from the `clear_data` using the Triple DES algorithm.

Return Codes

Common return codes generated by this routine are:

mk_NO_ERROR	The operation was successful.
mk_INVALID_DATA_LENGTH	The data length is not a multiple of eight.
mk_SRDI_OPEN_ERROR	Could not open the master key SRDI.
mk_SEM_CLAIM_FAILED	Unable to access the SRDI Manager.
mk_KEY_NOT_VALID	The designated master key is not valid.
mk_INVALID_KEY_SELECTOR	The input parameters are not valid.

triple_decrypt_under_master_key

triple_decrypt_under_master_key triple decrypts eight bytes of data with the EDE algorithm, using the specified master key register.

Function Prototype

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

`ciphertext` is a pointer to a buffer containing the data to be deciphered.

`cleartext` is a pointer to a buffer 8 bytes in length. This may be the same as the `ciphertext` buffer.

Output

On successful exit from this routine:

`cleartext` contains 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 to be used for decryption.
- `type_mks` should be set to ASYM_MK if the data is to be decrypted with the asymmetric-keys master key or to SYM_MK if the data is to be decrypted with the symmetric-keys master key.

`cv` is a pointer to a double-length CCA control vector, which is exclusive-ORed with the specified key value before the key is used.

`ciphertext` is a pointer to a buffer containing the data to be deciphered.

`cleartext` is a pointer to a buffer 8 bytes in length. This may be the same as the `ciphertext` buffer.

Output

On successful exit from this routine:

`cleartext` is a pointer to the buffer where the deciphered data is placed. This buffer may be the same as the `ciphertext` buffer, if desired.

Return Codes

Common return codes generated by this routine are:

mk_NO_ERROR	The operation was successful.
mk_SRDI_OPEN_ERROR	Could not open the master key SRDI.
mk_SEM_CLAIM_FAILED	Unable to access the SRDI Manager.
mk_INVALID_KEY_SELECTOR	The input parameters are not valid.

triple_encrypt_under_master_key

triple_encrypt_under_master_key triple encrypts eight bytes of data with the EDE algorithm, using the specified master key register.

Function Prototype

```
long triple_encrypt_under_master_key(mk_selectors *mk_selector,
                                     UCHAR        *cleartext,
                                     UCHAR        *ciphertext);
```

Input

On entry to this routine:

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

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- `mk_register` is set to either old_mk, current_mk, or new_mk, representing the key to be used for encryption.
- `type_mks` should be set to ASYM_MK if the data is to be encrypted with the asymmetric-keys master key or to SYM_MK if the data is to be encrypted with the symmetric-keys master key.

`cleartext` is a pointer to a buffer containing the data to be enciphered.

`ciphertext` is a pointer to a buffer which is 8 bytes in length. This may be the same as the `cleartext` buffer.

Output

On successful exit from this routine:

`ciphertext` is a pointer to the buffer where the enciphered data is placed. This buffer may be the same as the `cleartext` buffer, if desired.

Return Codes

Common return codes generated by this routine are:

mk_NO_ERROR	The operation was successful.
mk_SRDI_OPEN_ERROR	Could not open the master key SRDI.
mk_SEM_CLAIM_FAILED	Unable to access the SRDI Manager.
mk_KEY_NOT_VALID	The master key (OMK, CMK, or NMK) is not a valid key.
mk_INVALID_KEY_SELECTOR	The input parameters are not valid.

triple_encrypt_under_master_key_with_CV

triple_encrypt_under_master_key_with_CV triple encrypts eight bytes of data with the EDE algorithm, using a control vector with the specified master key.

Note: This function does not check the validity of the control vector.

Function Prototype

```
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 to be used for encryption.
- `type_mks` should be set to ASYM_MK if the data is to be encrypted with the asymmetric-keys master key or to SYM_MK if the data is to be encrypted with the symmetric-keys master key.

`cv` is a pointer to a double-length CCA control vector, which is exclusive-ORed with the specified key value before the key is used.

`cleartext` is a pointer to a buffer containing the data that is enciphered.

`ciphertext` is a pointer to a buffer which can hold 8 bytes of data. This may be the same as the `cleartext` buffer.

Output

On successful exit from this routine:

`ciphertext` is a pointer to the buffer where the enciphered data is placed. This buffer may be the same as the `cleartext` buffer, if desired.

Return Codes

Common return codes generated by this routine are:

mk_NO_ERROR	The operation was successful.
mk_SRDI_OPEN_ERROR	Could not open the master key SRDI.
mk_SEM_CLAIM_FAILED	Unable to access the SRDI Manager.

Chapter 7. SHA-1 Functions

The functions described in this chapter allow a UDX to compute the hash of a block of data using the Secure Hash Algorithm (SHA-1) as defined in FIPS Publication 180-1.

Note: 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:

computeHMAC_SHA1	Compute a keyed-hash for message authentication code (HMAC) for a given set of data using the Secure Hash Algorithm (SHA-1).
do_sha_hash_message	Compute the hash of a block of data using the SHA-1 algorithm.
do_sha_hash_msg_to_bfr	“wrapper” for sha_hash_message.
hw_sha_hash_message	Compute a SHA-1 hash of the requested data on the hashing hardware.
sha_hash_message	Compute the hash of a block of data using the SHA-1 algorithm.
sha_hash_msg_to_bfr	“wrapper” for sha_hash_message.

computeHMAC_SHA1 - Compute HMAC using SHA-1 Algorithm

computeHMAC_SHA1 computes a keyed-hash for message authentication code (HMAC) for a given set of data, using the SHA-1 algorithm and a provided key.

Function Prototype

```
void computeHMAC_SHA1 (char       *pBuffer, int buffer_length,
                       char       *pKey,    int key_length,
                       char       *pHmac);
```

Input

On entry to this routine:

`pBuffer` is a pointer to an array which holds the data to be hashed for message authentication codes.

`buffer_length` is the number of bytes of data in `pbuffer`.

`pKey` is a pointer to a key (a random string of bits, preferably 64 bytes long).

`key_length` is the number of bytes of data in `pKey`.

`pHmac` is a pointer to a buffer 20 bytes long, which will hold the returned data.

Output

On successful exit from this routine:

`pHmac` contains the HMAC-SHA1 of the data in `pBuffer`.

Return Codes

This function has no return codes.

do_sha_hash_message - Calculate SHA-1 Hash Hardware/Software

do_sha_hash_message calculates the SHA-1 hash of the data using the coprocessor hardware if the data is longer than 8192 bits, or the software if the data is shorter.

Function Prototype

```
ULONG do_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 7-4 for details.

`middle` The input data constitutes an additional portion of a block of data to be hashed. See “Chained Operations” on page 7-4 for details.

`final` The input data constitutes the final portion of a block of data to be hashed. See “Chained Operations” on page 7-4 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 7-4 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 do_sha_hash_message with `MsgPart` set to `first` and the first piece of the block of data to hash identified by `pBlock` and `*pBitCount`. On return, `*pContext` contains context information that must be preserved and passed to sha_hash_message when the next piece of the block of data to hash is processed.

Subsequent pieces of the block are processed by calling sha_hash_message with `MsgPart` set to `middle` (or to `final` if the piece in question is the last) and the location and length of the piece identified by `pBlock` and `*pBitCount`. `*pContext` must contain the value returned in that structure by the call to sha_hash_message that processed the previous piece of the block. The function hashes the piece and updates `*pContext` and `pHash` appropriately.

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) );

do_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 */
    do_sha_hash_message(employee[i].name, &Hash, &BitCount, &Context, middle);
}
/* hash the final portion */
sha_hash_message(employee[MAX_EMP-1].name, &Hash, &BitCount, &Context,final);
/* at this point, the value in Hash is the SHA-1 hash of the names */
```

do_sha_hash_msg_to_bfr - SHA-1 Hash

do_sha_hash_msg_to_bfr is a wrapper for sha_hash_message that simplifies the interface when chained operations (see page 7-4) are not necessary.

Function Prototype

```
void do_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 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.

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));
  do_sha_hash_message(pBlock,pHash,pBitCount,&Context,only);
}
```

Return Codes

This function has no return codes.

hw_sha_hash_message - Compute SHA-1 Hash in Hardware

hw_sha_hash_message computes a SHA-1 hash of the requested data on the hashing hardware in the coprocessor.

Function Prototype

```
void hw_sha_hash_message(UCHAR       *pText,
                         long         text_length,
                         sha_context *chain_vector,
                         long         RequestId,
                         ULONG        options,
                         UCHAR       *final_data,
                         long        *pMsg);
```

Input

On entry to this routine:

`pText` is a pointer to the data which is to be hashed.

`text_length` is the number of bytes of data at `pText`.

`chain_vector` is the context of this request. If this is the first block of data, or the only block, this variable should be initialized to zeros. On subsequent calls, this variable should be returned to the next call without change.

`RequestId` is the ID of the caller.

`options` may include SHA_MSGPART_FIRST, SHA_MSGPART_MIDDLE, SHA_MSGPART_FINAL, or SHA_MSGPART_ONLY, as well as SHA_INTERNAL_INPUT or SHA_EXTERNAL_INPUT. These constants are defined in the **scc_int.h** header file.

`final_data` is a pointer to a buffer at least 20 bytes long, to hold the result of the hash.

`pMsg` is a pointer to a 4-byte block.

Output

On successful exit from this routine:

`chain_vector` contains the chaining information for the next call to the function, unless this call was with the final or only block of data. This variable should be passed unchanged to the next call.

`final_data` contains the final hash value, if this call was with the final or only data block.

`pMsg` contains the return code for the function.

Return Codes

Common return codes generated by this routine are:

OK	The operation was successful.
SHA1_DATA64_ERROR	The options argument specified SHA_MSGPART_FIRST or SHA_MSGPART_MIDDLE but the length of the data to process is not a multiple of 64.
SHA1_DATA32MB_ERROR	The length of the data to process is 32M or greater.
SHA1_FINAL_ERROR	An error occurred while attempting to pad the input data as directed by the SHA-1 algorithm or while attempting to hash the pad bytes.

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 7-10 for details.

`middle` The input data constitutes an additional portion of a block of data to be hashed. See “Chained Operations” on page 7-10 for details.

`final` The input data constitutes the final portion of a block of data to be hashed. See “Chained Operations” on page 7-10 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 7-10 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 7-10) 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.

You should understand the use of the CCA control-vector before using the functions in this chapter. Control vectors are explained and described in Appendix C of the *CCA Basic Services Reference and Guide*. Three bits in the basic control vector have been reserved for UDX developers. Setting Bit 61 will prevent a key token from being used in any CCA standard verb except the import and export verbs. Bits 4 and 5 of the control vector will be checked by any standard CCA code. This allows developers to use these three bits to indicate their own, UDX specific keys, which can be used only by UDX verbs. (These verbs must be written to test the required bits.)

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

Header Files for DES Utility Functions

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

```
#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.

CasBuildCv	Builds a default control vector. See page 8-5.
CasBuildToken	Builds a default DES key token. See page 8-6.
CasCurrentMkvp	Returns the current master key verification pattern. See page 8-8.
CasOldMkvp	Returns the old master key verification pattern. See page 8-9.
CasMasterKeyCheck	Performs a master key version check. See page 8-14.
cas_adjust_parity	Adjusts the parity of a DES key token. See page 8-4.
cas_des_key_token_check	Verifies the integrity of a DES key token. See page 8-10.
cas_get_key_type	Returns the type of DES key token. See page 8-11.
cas_key_length	Returns the length of a DES key. See page 8-12.

cas_key_tokentvv_check	Verifies a DES key token validation value. See page 8-13.
cas_parity_odd	Determines whether a DES key has odd parity. See page 8-16.
RecoverDesDataKeyWithMK	Recovers the cleartext form of a DES importer data key. See page 8-17.
RecoverDesKekImporterWithMK	Recovers the cleartext form of a DES key encrypting key (KEK). See page 8-19.

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.

CasBuildCv - Build a Default Control Vector

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

Function Prototype

```
boolean CasBuildCv          ( KEY_TYPES   KeyType,
                              CV_LENGTHS  CV_Length,
                              UCHAR      *pCV )
void    cas_build_default_cv( KEY_TYPES   KeyType,
                              UCHAR      *pCV )
```

`cas_build_default_cv` has the same effect as calling `CasBuildCv` after setting the CV_Length parameter to CV_DEFAULT.

Input

On entry to this routine:

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

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

`CV_Length` is one of CV_DOUBLE, CV_SINGLE, or CV_DEFAULT, depending on what length of key you are building a control vector for.

Output

On successful exit from this routine:

`CasBuildCv` returns true if the build was successful, and false otherwise. (For example, if the length requested was not legal for the key type.)

`pCV` contains the new control vector.

Return Codes

This function has no return codes.

CasBuildToken - Build a Default Token

CasBuildToken builds a default key token, of the type specified by parameter *KeyType* for the mk_set being processed.

Function Prototype

```
boolean CasBuildToken       ( UCHAR                    TokenFlag,
                              KEY_TYPES                KeyType,
                              CV_LENGTHS               CV_Length,
                              mk_selectors            *pMKSelector,
                              des_key_token_structure *pKeyToken )
void cas_build_default_token( UCHAR                    TokenFlag,
                              KEY_TYPES                KeyType,
                              des_key_token_structure *pKeyToken )
```

`cas_build_default_token` has the same effect as calling `CasBuildToken` with CV_Length set to CV_DEFAULT and MKSelector set to {MK_SET_DEFAULT, current_mk, SYM_MK}.

Input

On entry to this routine:

`TokenFlag` is the token flag used in constructing the new key token. Legal values for this field are INTERNAL_TOKEN_FLAG and EXTERNAL_TOKEN_FLAG.

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

`CV_Length` is one of CV_DOUBLE, CV_SINGLE, or CV_DEFAULT, depending on what length of key you are building a control vector for.

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

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- `mk_register` must be set to current_mk.
- `type_mks` should be set to SYM_MK.

`pKeyToken` is a pointer to a 64-byte buffer which can store a key token.

Output

On successful exit from this routine:

`pKeyToken` contains the token constructed by the function.

Return Codes

This function has no return codes.

CasCurrentMkvp - Current Master Key Verification Pattern

CasCurrentMkvp returns the 20-byte master key verification pattern (MKVP) for the current master key for the mk_set being processed. The MKVP is a cryptographically calculated checksum on the master key value. It is used in all internal (master-key encrypted) DES key tokens, to indicate which master key was used to encrypt the key.

Function Prototype

```
boolean CasCurrentMkvp  ( mk_selectors  *pMKSelector,
                          UCHAR         *pMKVP)
boolean cas_current_mkvp( UCHAR         *pMKVP )
```

`cas_current_mkvp` has the same effect as calling `CasCurrentMkvp` with the pMKSelector parameter set to {MK_SET_DEFAULT, current_mk, SYM_MK}.

Input

On entry to this routine:

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

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- `mk_register` must be set to new_mk or current_mk.
- `type_mks` should be set to SYM_MK.

`pMKVP` must contain the address of a variable in which a 20-byte master key verification pattern can be stored.

Output

On successful exit from this routine:

`pMKVP` contains the current master key verification pattern.

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

Return Codes

This function has no return codes.

CasOldMkvp - Old Master Key Verification Pattern

CasOldMkvp returns the 20-byte master key verification pattern (MKVP) for the old master key for the mk_set being processed. The MKVP is a cryptographically calculated checksum on the master key value. It is used in all internal (master-key encrypted) DES key tokens, to indicate which master key was used to encrypt the key.

Function Prototype

```
boolean CasOldMkvp  ( mk_selectors  *pMKSelector,
                      UCHAR         *pMKVP)
boolean cas_old_mkvp( UCHAR         *pMKVP )
```

`cas_old_mkvp` has the same effect as calling `CasOldMkvp` with the pMKSelector parameter set to {MK_SET_DEFAULT, current_mk, SYM_MK}.

Input

On entry to this routine:

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

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- `mk_register` must be set to new_mk or current_mk.
- `type_mks` should be set to SYM_MK.

`pMKVP` must contain the address of a variable in which a 20-byte master key verification pattern can be stored.

Output

On successful exit from this routine:

`pMKVP` contains the current master key verification pattern.

`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.

Either version 0 or version 1 DES key tokens may be validated.

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.

CasMasterKeyCheck - Master Key Version Check

CasMasterKeyCheck determines which version of the master key was used to encrypt the specified key token for the mk_set being processed. The response indicates whether the key token is encrypted using the current master key, the old master key, or a master key that is no longer available.

Function Prototype

```
UNDER_MASTER_KEY  CasMasterKeyCheck   ( mk_selectors            *pMKSelector,
                                        des_key_token_structure *pKeyToken )
UNDER_MASTER_KEY  cas_master_key_check( des_key_token_structure *pKeyToken )
```

`cas_master_key_check` has the same effect as calling `CasMasterKeyCheck` with the pMKSelector parameter set to {MK_SET_DEFAULT, current_mk, SYM_MK}.

Input

On entry to this routine:

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

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- `mk_register` must be set to new_mk or current_mk.
- `type_mks` should be set to SYM_MK.

`pKeyToken` is a pointer to the key token which is to be examined.

Output

On successful exit from this routine:

`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.

RecoverDesDataKeyWithMK - Recover DES Data Key

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

Function Prototype

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

RecoverDesDataKey is the equivalent of calling RecoverDesDataKeyWithMK after setting the MKSelector parameter to {MK_SET_DEFAULT, current_mk, SYM_MK}.

Input

On entry to this routine:

pDesToken is a pointer to the input key token.

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

- mk_set is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- mk_register must be set to new_mk or current_mk.
- type_mks should be set to SYM_MK.

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.

RecoverDesKekImporterWithMK - Recover DES Importer KEK

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

Function Prototype

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

`RecoverDesKekImporter` is the equivalent of calling `RecoverDesKekImporterWithMK` with the MKSelector parameter set to {MK_SET_DEFAULT, current_mk, SYM_MK}.

Input

On entry to this routine:

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

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

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- `mk_register` must be set to new_mk or current_mk.
- `type_mks` should be set to SYM_MK.

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"
#include "cacmkld.h"                   /* functions for generating RSA
                                       /* signatures and registered keys */
```

Summary of Functions

RSA keys and key tokens include the following functions.

CalculatenWordLength	Returns the word length. See page 9-6.
CreateInternalKeytokenWithMK	Receives a clear key token and creates the internal form. See page 9-7.
CreateRsaInternalSectionWithMK	Creates the RSA internal section. See page 9-8.
delete_KeyToken	Remove a registered public or private key from storage. See page 9-9.
GenerateCcaRsaToken	Generates a CCA RSA key token from an internal format key token and a CCA PKA skeleton token. See page 9-10.
GenerateRsaInternalToken	Creates an internal RSA token from a CCA RSA key token. See page 9-11.
generate_dSig	Receives an RSA key token (in operational format) and a buffer of data (with the length and the digital signature). See page 9-12.
GeteLength	Returns the RSA public exponent byte length. See page 9-14.

getKeyToken	Retrieves a PKA token from the SRDI where it is stored. See page 9-15.
GetModulus	Extracts and copies the RSA modulus. See page 9-16.
GetnBitLength	Returns the bit length of the RSA modulus. See page 9-17.
GetnByteLength	Returns the byte length of the RSA modulus. See page 9-18.
GetPublicExponent	Extracts and copies the RSA key public exponent. See page 9-19.
GetRsaPrivateKeySection	Returns a pointer to the private key section of an RSA key token. See page 9-20.
GetRsaPublicKeySection	Returns a pointer to the public key section of an RSA key token. See page 9-21.
GetTokenLength	Returns the length of the specified token. See page 9-22.
IsPrivateExponentEven	Verifies whether the RSA private exponent is an even valued integer. See page 9-23.
IsPrivateKeyEncrypted	Verifies whether the private key section of the specified key token is encrypted. See page 9-24.
IsPublicExponentEven	Verifies whether the RSA public exponent is an even valued integer. See page 9-25.
IsRsaToken	Verifies whether the key token contains an RSA key. See page 9-26.
IsTokenInternal	Identifies whether the key token is in internal format. See page 9-27.
PkaHashQueryWithMK	Returns the master key version used to encrypt the specified token. See page 9-28.
PkaMkvpQueryWithMK	Returns a value indicating which master key was used to encrypt the specified key token. See page 9-29.
pka96_tvvgen	Calculates the token validation value (TVV) for the specified key token. See page 9-30.

RecoverPkaClearKeyTokenUnderMkWithMK	Recovers the PKA clear key token under the master key. See page 9-31.
RecoverPkaClearKeyTokenUnderXport	Recovers the PKA clear key token under the DES export key. See page 9-33.
ReEncipherPkaKeyTokenWithMK	Re-enciphers an internal PKA key token from the old master key to the current master key. See page 9-34.
RequestRSACrypto	Performs an RSA operation. See page 9-35.
store_KeyToken	Saves a public or private key to the SRDI. See page 9-36.
TokenMkvpMatchMasterKey	Tests whether the key token was encrypted using a specified version of the master key. See page 9-37.
ValidatePkaToken	Verifies that the RSA key token is valid for use in the system. See page 9-38.
VerifyKeyTokenConsistency	Verifies the consistency of a key token. See page 9-39.
verify_dSig	Verifies the RSA key token signature. See page 9-40.

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.
RecoverPkaClearKeyTokenUnderMkWithMK	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.

CreateInternalKeyTokenWithMK - Create Internal Key Token

CreateInternalKeyTokenWithMK receives a clear CCA RSA key token and creates the internal form by encrypting the private key areas under the master key for the mk_set being processed.

Function Prototype

```
long CreateInternalKeyTokenWithMK ( RsaKeyTokenHeader  *pTokenIn,
                                    mk_selectors       *pMKSelector,
                                    RsaKeyTokenHeader  *pTokenOut )
long CreateInternalKeyToken       ( RsaKeyTokenHeader  *pTokenIn,
                                    RsaKeyTokenHeader  *pTokenOut )
```

`CreateInternalKeyToken` has the same effect as calling `CreateInternalKeyTokenWithMK` after setting the MKSelector parameter to {MK_SET_DEFAULT, current_mk, ASYM_MK}.

Input

On entry to this routine:

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

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

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- `mk_register` must be set to current_mk
- `type_mks` should be set to ASYM_MK.

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.

CreateRsaInternalSectionWithMK - Create RSA Internal Section

CreateRsaInternalSectionWithMK receives an SCC complete token and creates an internal CCA RSA key token by calculating the validation values and encrypting under the master key for the mk_set being processed.

Function Prototype

```
long CreateRsaInternalSectionWithMK ( RsaKeyTokenHeader *pTokenOut,
                                      mk_selectors      *pMKSelector,
                                      sccRSAKeyToken_t  *pRsaTokenIn )
long CreateRsaInternalSection       ( RsaKeyTokenHeader *pTokenOut,
                                      sccRSAKeyToken_t  *pRsaTokenIn )
```

`CreateRsaInternalSection` has the same effect as calling `CreateRsaInternalSectionWithMK` after setting the MKSelector parameter to {MK_SET_DEFAULT, current_mk, ASYM_MK}.

Note: This function only works with version 0 tokens.

Input

On entry to this routine:

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

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

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- `mk_register` must be set to current_mk.
- `type_mks` should be set to ASYM_MK.

`pRsaTokenIn` is a pointer to the internal SCC key structure.

Output

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

The internal section of the CCA RSA key token is created.

`pTokenOut` contains the 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.

generate_dSig - Receives RSA Key Token

generate_dSig receives an RSA key token in operational format and a buffer of data, with the length of the data and the expected length of the digital signature. The key token is deciphered and the input data is hashed with SHA-1, then the data is formatted according to the requested Type before signing with the clear key. The format may be one of ISO-9796, PKCS #1 block type 0 or 1, or zero-padded.

Function Prototype

```
long generate_dSig ( RsaKeyTokenHeader *pTokenIn,
                     UCHAR             *pDataIn,
                     long               DataLength,
                     UCHAR             *pSignatureOut,
                     USHORT            *pSignatureBitLength,
                     UCHAR              Type )
```

Input

On entry to this routine:

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

`pDataIn` is a pointer to the data which is to be signed.

`DataLenth` is the length of the data to be signed, in bytes.

`pSignatureOut` is a pointer to a buffer which is to hold the returned signature.

`pSignatureBitLength` is a pointer to the length of the buffer pSignatureOut, in bits.

`Type` is one of the following:

- `M_ISO9796` if the data is to be formatted according to the ISO-9796 standard before signing.
- `M_PKCS10` if the data is to be formatted as specified in the RSADataSecurity, Inc., *Public Key Cryptography Standards #1* block type 00 before signing.
- `M_PKCS11` if the data is to be formatted as specified in the RSADataSecurity, Inc., *Public Key Cryptography Standards #1* block type 01 before signing.
- `M_ZEROPAD` if the Data is to be placed in the low-order bits of a bit-string of the same length as the modulus with all other bit-positions set to zero before signing.

Output

On successful exit from this routine:

`pSignatureBitLength` contains the length (in bits) of the calculated digital signature.

`pSignatureOut` contains the digital signature.

Return Codes

Common return Codes generated by this routine are:

OK	The operation was successful.
E_SIZE	The provided buffer was not large enough to contain the signature.
PKABadAddr	The key token is not valid.

GeteLength - Return RSA Public Exponent Byte Length

GeteLength returns the byte length of the RSA public exponent field, as contained in the member field of the key token.

Note: The member field is a 16-bit field and is in zSeries (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 zSeries (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

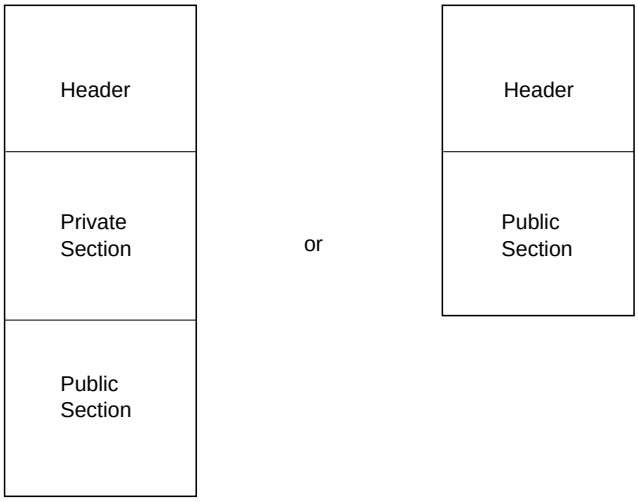
This function returns no output. On successful exit from this routine:

`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 zSeries (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.

PkaHashQueryWithMK - Return Master Key Version

PkaHashQueryWithMK returns a value indicating which master key was used to encrypt the specified key token for the mk_set being processed.

Function Prototype

```
MK_VERSION PkaHashQueryWithMK ( RsaKeyTokenHeader *pToken,
                                mk_selectors      *pMKSelector )
```

Input

On entry to this routine:

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

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

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- `mk_register` must be set to new_mk, current_mk, or old_mk.
- `type_mks` should be set to ASYM_MK.

Output

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

Common return codes generated by this routine are:

OK	The operation was successful.
mk_KEY_NOT_VALID	The current master key is not valid.
mk_SEM_CLAIM_FAILED	Could not access the master keys.

PkaMkvpQueryWithMK - Return Master Key Version

PkaMkvpQueryWithMK returns a value indicating which master key was used to encrypt the specified key token for the mk_set being processed.

Function Prototype

```
MK_VERSION PkaMkvpQueryWithMK ( RsaKeyTokenHeader *pToken,
                                mk_selectors      *pMKSelector
MK_VERSION PkaMkvpQuery       ( RsaKeyTokenHeader *pToken )
```

`PkaMkvpQuery` has the same effect as calling `PkaMkvpQueryWithMK` after setting the MKSelector parameter to {MK_SET_DEFAULT, current_mk, ASYM_MK}.

This function only works with version 0 tokens.

Input

On entry to this routine:

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

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

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- `mk_register` must be set to current_mk
- `type_mks` should be set to ASYM_MK.

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.

RecoverPkaClearKeyTokenUnderMkWithMK

RecoverPkaClearKeyTokenUnderMkWithMK receives a PKA key token which is encrypted under the master key for the mk_set which is currently in use. If the key is in the on-board cache of decrypted keys, this key is returned to the calling function. Otherwise, the clear form of the key is recovered by decrypting the private areas of the key and verifying the SHA-1 hashes of those sections. The clear key is then added to the on-board cache before being returned to the calling function.

Function Prototype

```
long RecoverPkaClearTokenUnderMkWithMK( RsaKeyTokenHeader  *pTokenIn,
                                        RsaKeyTokenHeader  *pTokenOut,
                                        mk_selectors       *pMKSelector,
                                        long               *pMsg )
long RecoverPkaClearKeyTokenUnderMk   ( RsaKeyTokenHeader  *pTokenIn,
                                        RsaKeyTokenHeader  *pTokenOut,
                                        long               *pMsg )
```

`RecoverPkaClearKeyTokenUnderMk` has the same effect as calling `RecoverPkaClearTokenUnderMkWithMK` after setting the MKSelector parameter to {MK_SET_DEFAULT, current_mk, ASYM_MK}.

Input

On entry to this routine:

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

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

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- `mk_register` must be set to new_mk or current_mk
- `type_mks` should be set to ASYM_MK.

Output

On successful exit from this routine:

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

`pMsg` is the error code.

Notes

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

This function does not change the value of byte 28 of the private key, the Key format and Security byte. If you are planning to store this key in clear form, you should change this byte to the appropriate value before storing. Refer to Appendix B of the *CCA Basic Services Reference and Guide* for the appropriate values for different RSA key token formats.

Return Codes

Common return codes generated by this routine are:

OK	The operation was successful.
ERROR	`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.

ReEncipherPkaKeyTokenWithMK - Re-Encipher PKA Key Token

ReEncipherPkaKeyTokenWithMK re-enciphers an internal PKA key token from the old master key to the current master key for the mk_set being processed.

Function Prototype

```
long ReEncipherPkaKeyTokenWithMK( RsaKeyTokenHeader  *pToken,
                                  mk_selectors       *pMKSelector,
                                  UCHAR              *pWorkArea )
long ReEncipherPkaKeyToken      ( RsaKeyTokenHeader  *pToken,
                                  UCHAR              *pWorkArea )
```

`ReEncipherPkaKeyToken` has the same effect as calling `ReEncipherPkaKeyTokenWithMK` after setting the PMKSelector parameter to {MK_SET_DEFAULT, current_mk, ASYM_MK}.

Input

On entry to this routine:

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

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

- `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- `mk_register` must be set to new_mk or current_mk
- `type_mks` should be set to ASYM_MK.

`pWorkArea` is a pointer to a variable which can hold a private key. This is used as a work area when decrypting.

Output

On successful exit from this routine:

`pToken` contains the key token, which has been enciphered under the 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 RecoverPkaClearKeyTokenUnderMkWithMk().

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. This number is presumed to be equal to the length of the output data buffer, in bits. If this number is *larger* than the modulus length in bits, the data which will be operated on is in the rightmost modulusLength bits of the input data buffer, and the result will be placed in the rightmost modulusLength bits of the output data buffer.

`RsaOperation` is the requested RSA operation, such as RSA_ENCRYPT (public key operation) or RSA_DECRYPT (private key operation).

Output

On successful exit from this routine:

`pOutput` is a pointer to a buffer that receives the results of the requested RSA operation.

Return Codes

Common return codes generated by this routine are:

OK	The operation was successful.
ERROR	Could not create a buffer to receive the RSA key token.
E_SIZE	The data is smaller 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.
- `type_mks` should be set to ASYM_MK.

`pToken` is a pointer to the key token that you want to test.

Output

On successful exit from this routine:

`TokenMkvpMatchMasterKey` returns TRUE if the MKVP of the specified key token is equal to the MKVP for the specified master key, and FALSE if it is not.

Return Codes

This function has no return codes.

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.

verify_dSig - Verify RSA Key Token Signature

verify_dSig receives an RSA key token in operational form, a buffer of data (with the length of the data), a digital signature and the length of the digital signature (in bytes), as well as the format of the digital signature. The data is hashed with SHA-1 and formatted according to the Type variable before being compared with the encrypted signature. The return code indicates whether the signature was verified.

Function Prototype

```
long verify_dSig ( RsaKeyTokenHeader *pTokenIn,
                   UCHAR             *pDataIn,
                   long               DataLength,
                   UCHAR             *pDigitalSignature,
                   USHORT             SignatureLength,
                   UCHAR              Type)
```

Input

On entry to this routine:

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

`pDataIn` is a pointer to the data which is to be hashed and compared with the encrypted signature.

`DataLenth` is the length of the data to be signed, in bytes.

`pSignatureOut` is a pointer to a buffer which contains the signature to be verified.

`SignatureLength` is the length of the buffer pSignatureOut, in bytes.

`Type` is one of the following:

- `M_ISO9796` if the data is to be formatted according to the ISO-9796 standard before signing.
- `M_PKCS10` if the data is to be formatted as specified in the RSADataSecurity, Inc., *Public Key Cryptography Standards #1* block type 00 before signing.
- `M_PKCS11` if the data is to be formatted as specified in the RSADataSecurity, Inc., *Public Key Cryptography Standards #1* block type 01 before signing.
- `M_ZEROPAD` if the Data is to be placed in the low-order bits of a bit-string of the same length as the modulus with all other bit-positions set to zero before signing.

Return Codes

Common return Codes generated by this routine are:

OK	The operation was successful.
DSIG_NOT_VERIFIED	The digital signature was not verified.
E_SIZE	The provided buffer was not large enough to contain the signature.
PKABadAddr	The key token is not valid.

Chapter 10. CCA SRDI Manager Functions

This 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.

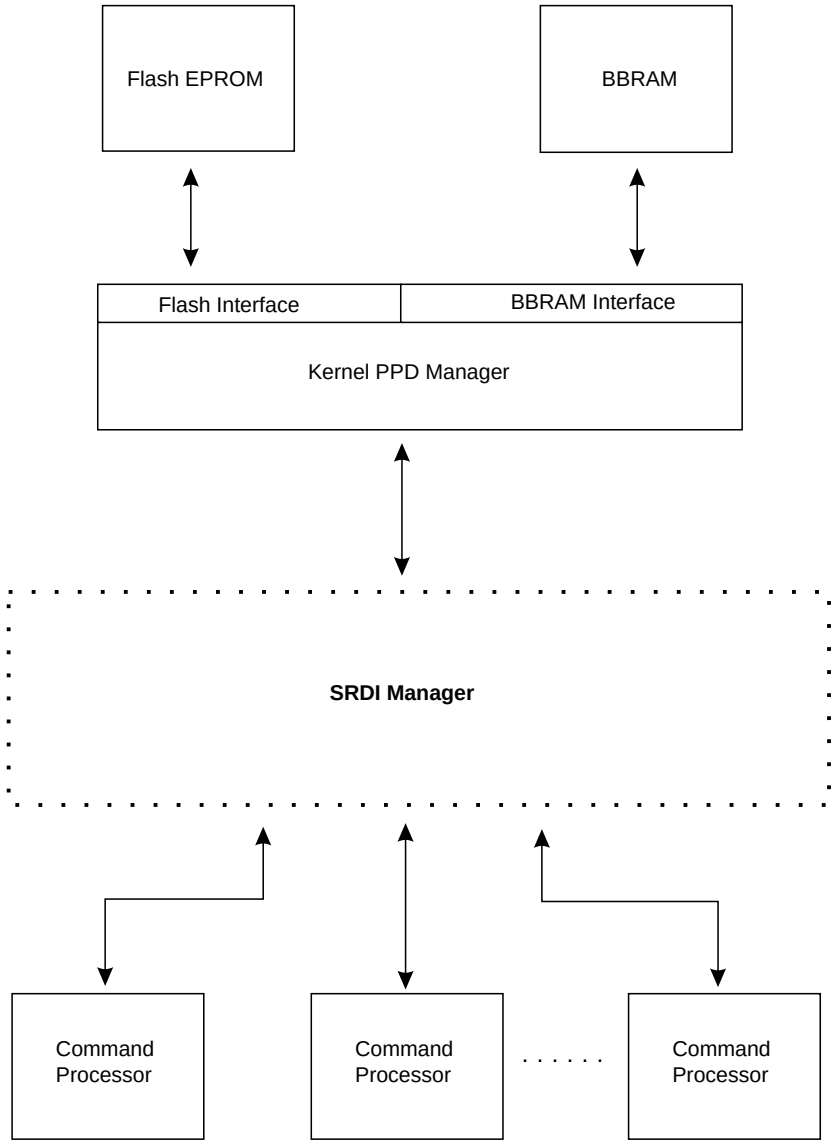


Figure 10-1. Master SRDI Manager Overview

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.

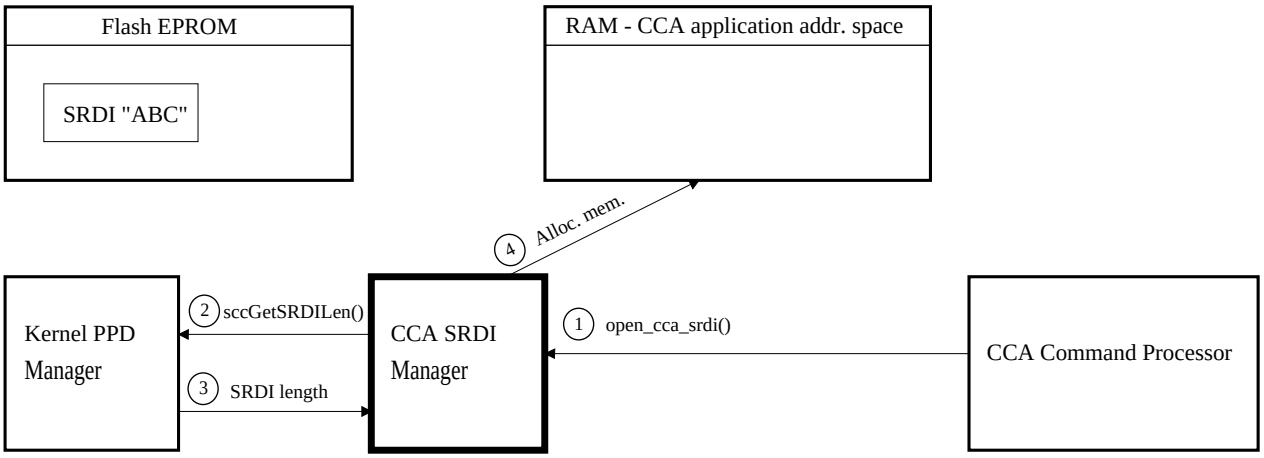


Figure 10-2. Master SRDI Read Illustration, Part 1

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.

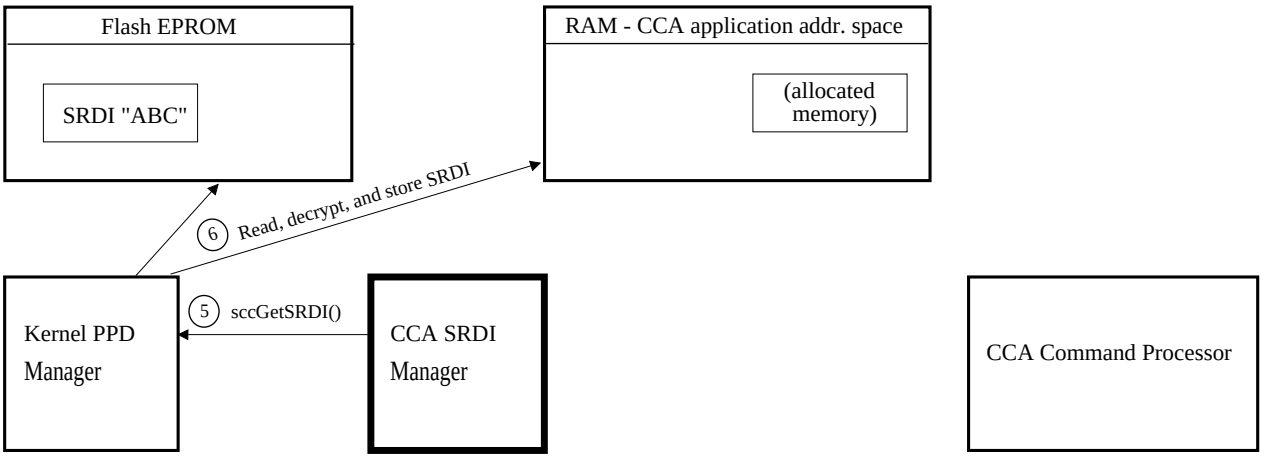


Figure 10-3. Master SRDI Read Illustration, Part 2

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*.

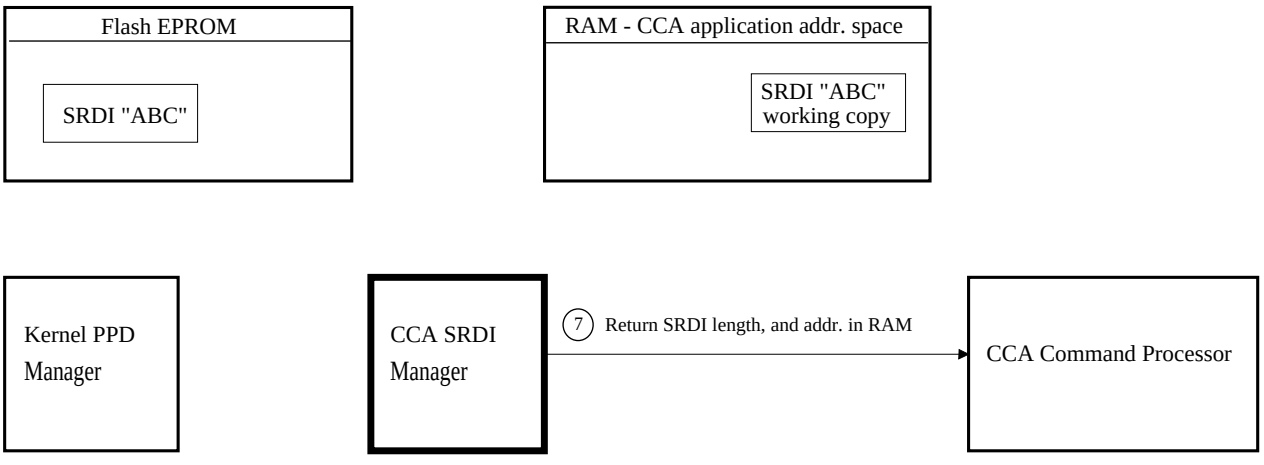


Figure 10-4. Master SRDI Read Illustration, Part 3

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.
create4update_cca_srdi	Creates an SRDI in BBRAM.
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.
update_cca_srdi	Updates an SRDI with the provided 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.

create4update_cca_srdi - Create CCA SRDI for Update Only

create4update_cca_srdi creates an SRDI in BBRAM. The SRDI may not be resized after creation. The SRDI is stored in unencrypted form.

Function Prototype

```
long create4update_cca_srdi(char *pSrdiName,
                            char *pSrdiAddr,
                            ULONG SrdiLength);
```

Input

On entry to this routine:

`pSrdiName` is a pointer to an eight character ASCII string containing the name of the desired SRDI. This string does not contain a null terminator, and should be padded on the right with blanks.

`pSrdiAddr` is a pointer to the data which should be written to the SRDI.

`SrdiLength` contains the length of the data pointed to by pSrdiAddr, and the permanent length of this SRDI.

Output

This function returns no output. On successful exit from this routine, the data has been stored in BBRAM under the requested name.

Return Codes

Common return codes generated by this routine are:

srdi_NO_ERROR	The operation was successful.
srdi_EXISTS	The SRDI item already exists.
srdi_GENERAL_ERROR	The SRDI Manager was unable to create the SRDI.

delete_cca_srdi - Delete CCA SRDI

delete_cca_srdi deletes an SRDI from the persistent memory area where it is stored (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.

update_cca_srdi - Update an SRDI Item

update_cca_srdi updates an SRDI with provided data.

Function Prototype

```
long update_cca_srdi(char *pSrdiName,
                     char *pDatabuffer,
                     unsigned long Datalength,
                     unsigned long Dataoffset);
```

Input

On entry to this routine:

`pSrdiName` is a pointer to an eight character string containing the name of the SRDI to be changed. This string does not contain a null terminator and should be padded on the right with blanks. This SRDI must have been opened using the `open_cca_srdi` call.

`pDatabuffer` is a pointer to a buffer containing the data with which to update the SRDI.

`Datalength` contains the length of the data to be changed, in bytes.

`Dataoffset` contains the offset of the first byte of data to change from `pDatabuffer`.

Output

This function returns no output. On successful exit from this routine, the SRDI item has been changed.

Return Codes

Common return codes generated by this routine are:

srdi_NO_ERROR	The operation was successful.
srdi_WRITE_ERROR	The operation could not be completed.

Example Code

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

```
#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. Cache Management Functions

This section describes functions used to maintain an on-board cache of security relevant data. For example, the CCA API currently uses these functions to maintain a cache of decrypted private keys.

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

Header Files for Caching Functions

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

```
#include "cache.h"   /* Cache management functions*/
```

Overview of Cache Management Functions

The cache management functions allow the operation of a cache of data in the DRAM of the coprocessor. Each entry in the cache consists of a data item and its unique identifier. The cache is indexed on two bytes of data supplied by the user, for example a predefined label or two chosen bytes of a hash of the item or two bytes of the unique identifier. The cache uses a Least Recently Used (LRU) replacement system, eliminating the item which has been unused the longest when more space is needed in the cache.

Data in the cache is stored and accessed using a two-level lookup process. An item is referenced using a “tag”, which is any arbitrary-length byte string that uniquely identifies an item. In addition, every access to an item passes a 2-byte “short” tag, which should also be as unique as possible to the item being accessed. It is up to the user to decide how to create the short tags - examples might include a hash of the full tag, the first two bytes of the full tag, or two fixed-position bytes of ciphertext for items that contain encrypted data.

The short tag is broken into two separate one-byte tags, referred to as tag1 and tag2. Tag1 is used as an index into a 256-entry array, where each element either contains a pointer to a list of items that are further addressed using tag2, or NULL if the cache does not contain any items with the indexed tag1 value.

The tag2 lists are linked-lists, where each element contains the data of a cached item. The linked-list, once addressed through the tag1 array, is searched linearly for items with tag2 entries matching the passed tag2 value. For each tag2 match, the entire full tag is compared with that stored in the item, to find the one that is the desired object. (Note that it is possible for multiple items to share identical tag1 and tag2 values, although it is unlikely for good choices of tag1/tag2 computation methods, unless the cache holds a very large number of items.)

Summary of Functions

These functions are used by the cache manager to manage the cache.

cache_clear	Remove all entries from a cache.
cache_delete	Remove an existing cache.
cache_delete_item	Remove a specific item from the cache.
cache_get_item	Retrieve an item from the cache, copying it into a user-supplied buffer.
cache_get_item_b	Retrieve an item from the cache after allocating memory to hold it.
cache_init	Open a new cache.
cache_status	Check the status of an existing cache.
cache_write_item	Store an item in the cache.

cache_clear

cache_clear clears all data from the specified cache.

Function Prototype

```
cacheError cache_clear(cacheHandle handle);
```

Input

On entry to this routine:

`handle` must be set to a valid cacheHandle.

Output

This function returns no output. On successful exit, all data has been cleared from the specified cache.

Return Codes

Common return codes generated by this routine are:

ce_OK	The function completed successfully.
ce_BAD_HANDLE	The specified cache does not exist.

cache_delete

cache_delete will remove an existing cache, recovering all memory.

Function Prototype

```
cacheError cache_delete(cacheHandle handle);
```

Input

On entry to this routine:

`handle` must be set to a valid cacheHandle.

Output

This function returns no output. On successful exit, all data has been cleared from the specified cache, the cache has been removed, and all associated memory objects have been freed.

Return Codes

Common return codes generated by this routine are:

ce_OK	The function completed successfully.
ce_BAD_HANDLE	The specified cache does not exist.

cache_delete_item

cache_delete_item deletes the specified item from the cache.

Function Prototype

```
cacheError cache_delete_item(cacheHandle  handle,
                             short_tag_t  short_tag,
                             ULONG        full_tag_length,
                             UCHAR       *full_tag);
```

Input

On entry to this routine:

`handle` contains the handle of the specified cache.

`short_tag` is the two-byte tag used for indexing the cache items.

`full_tag_length` is the length of the unique identifier of the item to be deleted.

`full_tag` is a buffer which contains the unique identifier of the item to be deleted.

Output

This function returns no output. On successful completion, the data item referenced by the short_tag and the full_tag has been removed from the cache.

Return Codes

Common return codes generated by this routine are:

ce_OK	The function completed successfully.
ce_BAD_HANDLE	The specified cache does not exist.
ce_ITEM_NOT_FOUND	The specified item was not found in the cache.

cache_get_item

cache_get_item retrieves a specific item from the cache and copies it into a buffer supplied by the user.

Function Prototype

```
cacheError cache_get_item(cacheHandle  handle,
                          ULONG       *bfrSize,
                          UCHAR       *bfr,
                          short_tag_t  short_tag,
                          ULONG        full_tag_length,
                          UCHAR       *full_tag);
```

Input

On entry to this routine:

`handle` contains the handle for the specified cache.

`bfrSize` is a pointer to the length of the available space in the supplied buffer.

`bfr` is a pointer to the buffer in which the retrieved item will be returned.

`short_tag` is the two-byte tag which is used to index the item in the cache.

`full_tag_length` is the length of the unique identifier of the item in the cache.

`full_tag` is the unique identifier of the item to be retrieved from the cache.

Output

On successful exit to this routine:

`bfrSizeis` a pointer to the length of the item which was retrieved from the cache.

`bfrcontains` the item which was retrieved.

Return Codes

Common return codes generated by this routine are:

ce_OK	The function completed successfully.
ce_BAD_HANDLE	The specified cache does not exist.
ce_ITEM_NOT_FOUND	The specified item was not found in the cache.
ce_BFR_TOO_SMALL	The provided buffer was not large enough to hold the item.

cache_get_item_b

cache_get_item retrieves a specific item from the cache and copies it into a buffer which is allocated by the function. This function is useful when the cache holds items with very different sizes.

Note: The calling function is responsible for freeing the memory allocated for the item when the item is no longer needed. However, if the function fails to complete successfully, the buffer is not allocated.

Function Prototype

```
cacheError cache_get_item_b(cacheHandle   handle,
                            ULONG        *bfrSize,
                            UCHAR       **bfr,
                            short_tag_t   short_tag,
                            ULONG         full_tag_length,
                            UCHAR        *full_tag);
```

Input

On entry to this routine:

`handle` is the handle which specifies the required cache.

`short_tag` is the two-byte tag used to index the item within the cache.

`full_tag_length` is the length of the unique identifier for the item.

`full_tag` is the unique identifier of the item.

Output

On successful exit from this routine:

`bfrSize` is a pointer to the size of the buffer returned.

`bfr` is a pointer to a buffer containing the recovered item. This buffer has been allocated by the function.

Return Codes

Common return codes generated by this routine are:

ce_OK	The function completed successfully.
ce_BAD_HANDLE	The specified cache does not exist.
ce_ITEM_NOT_FOUND	The specified item was not found in the cache.
ce_MEM_ALLOC_ERROR	The function was unable to allocate memory for the buffer to return the item.

cache_init

cache_init opens a new cache, returning a handle with which to access the cache. The caller specifies the maximum size of the cache and the maximum size of the items within the cache.

Function Prototype

```
cacheError cache_init(ULONG       maxBytes,
                      ULONG       maxItemBytes,
                      cacheHandle *handle);
```

Input

On entry to this routine:

`maxBytes` contains the maximum number of bytes to be used in the cache. This number must be large enough to hold at least one item of `maxItemBytes`, along with extra room for control structures. This number determines at which point the LRU algorithm will come into play, replacing old items with new ones.

`maxItemBytes` contains the maximum number of bytes which an item will occupy. This value should include both the maximum size of the item and the maximum size of the unique identifier.

`handle` is a pointer to an item of cacheHandle type, in which the function will return the handle for accessing the cache.

Output

On successful exit from this routine:

`handle` contains the cacheHandle used to access the new cache.

Return Codes

Common return codes generated by this routine are:

ce_OK	The function completed successfully.
ce_ITEM_TOO_LARGE	The `maxItemBytes` parameter is too large for the `maxBytes`.
ce_MEM_ALLOC_ERROR	The function was unable to allocate memory for the cache overhead.

cache_status

cache_status returns the status of the specified cache, that is bytes used by each item (including overhead), bytes used altogether, number of items currently in the cache.

Function Prototype

```
cacheError cache_status(cacheHandle  handle,
                        ULONG       *maxBytes,
                        ULONG       *maxItemBytes,
                        ULONG       *bytesUsed,
                        ULONG       *itemsStored) ;
```

Input

On entry to this routine:

`handle` is the handle of the cache to be queried.

`maxBytes` is a pointer to a long data item.

`maxItemBytes` is a pointer to a long data item.

`bytesUsed` is a pointer to a long data item.

`itemsStored` is a pointer to a long data item.

Output

On successful exit from this routine:

`maxBytes` is a pointer to the maximum number of bytes to be used in the cache, including overhead.

`maxItemBytes` is a pointer to the maximum number of bytes each entry will use, including the data item, the unique identifier, and the overhead.

`bytesUsed` is a pointer to the number of bytes currently used in the cache, including data and overhead.

`itemsStored` is a pointer to the number of items which are currently stored in the cache.

Return Codes

Common return codes generated by this routine are:

ce_OK	The function completed successfully.
ce_NULL_POINTER	One of the parameters was a null pointer.
ce_BAD_HANDLE	The specified cache does not exist.

cache_write_item

cache_write_item writes an item to the specified cache. If the cache is full, the function will delete the least recently used item from the cache before storing the new item.

Function Prototype

```
cacheError cache_write_item(cacheHandle  handle,
                            ULONG        item_length,
                            UCHAR       *item,
                            short_tag_t  short_tag,
                            ULONG        full_tag_length,
                            UCHAR       *full_tag) ;
```

Input

On entry to this routine:

`handle` is the handle which identifies the cache.

`item_length` is the length of the item to be stored. This length plus the `full_tag_length` must be smaller than the `maxItemBytes` specified when the cache was initialized.

`item` is a pointer to a buffer containing the item to be stored.

`short_tag` is the two-byte tag to be used to index this item within the cache.

`full_tag_length` is the length of the unique identifier for the item. This length plus `item_length` must be less than the maxItemBytes specified when the cache was initialized.

`full_tag` is the unique identifier for this data item.

Output

This function returns no output. On successful completion, the item has been added to the cache.

Return Codes

Common return codes generated by this routine are:

ce_OK	The function completed successfully.
ce_NULL_POINTER	One of the parameters was a null pointer.
ce_BAD_HANDLE	The specified cache does not exist.
ce_ITEM_TOO_LARGE	The sum of the full_tag_length and the item_length is greater than the maximum item size for the cache.

Chapter 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 if Key Identifier is a Label

This macro has a value of TRUE when the character input is valid for a key label. All key labels have a first byte between 0x20 and 0xFE. TOKEN_LABEL_CHECK should be used when only one byte is available for checking.

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

Appendix A. UDX Sample Code - Host Piece - Service

This appendix contains a listing of the sample file *zudxsvc.bal*. This file is a skeleton for the design of the host piece of a CCA extension.

```
         TITLE 'ZUDXSVC: Sample UDX - PIN Block Processing Service'
ZUDXSVC  CSECT
ZUDXSVC  AMODE 31
ZUDXSVC  RMODE ANY
*************************************************************************
*                                                                       *
*   Sample callable service for UDX.                                    *
*                                                                       *
*   (C) Copyright IBM Corp. 2001                                        *
*                                                                       *
*   Function: This program will process an encrypted PIN block          *
*             (assume a proprietary block form) and return the          *
*             block encrypted under the original or a second key.       *
*                                                                       *
*   Inputs:   Rule Array Count  Number of keywords passed in rule       *
*                               array                                   *
*                                                                       *
*             Rule Array        Keywords (0, 1 or 2 keywords may        *
*                               be passed)                              *
*                                                                       *
*             Input Pin Key Id  Input PIN encrypting key identifier.*
*                               The input PIN block is enciphered       *
*                               under this key. The identifier may      *
*                               be a token or label.                    *
*                                                                       *
*             Input Pin Block   Enciphered PIN block to be              *
*                               processed.                              *
*                                                                       *
*             Output Pin Key Id Output PIN encrypting key identifi- *
*                               er or a null token. The identifier      *
*                               may be a token or label. If the key *
*                               is not used, a null token is            *
*                               supplied.                               *
*                                                                       *
*             Extra Data        Extra data to be used in processing *
*                               the PIN block (always 8 bytes).         *
*                                                                       *
*   Processing: 1. Copy the parameter address block to the ICSF         *
*                  address space.                                       *
*                                                                       *
*               2. Copy all input parameters to the ICSF address        *
*                  space.                                               *
*                                                                       *
*               3. Validate the rule array count.                       *
*                                                                       *
*               4. Check that the caller is authorized to use this      *
*                  protected resource (RACF check).                     *
*                                                                       *
*               5. Process the InputPinKeyId. If the identifier is      *
*                  a label, retrieve the token from the CKDS.           *
*                                                                       *
*               6. Process the OutputPinKeyId. If the identifier is *
*                  a label, retrieve the token from the CKDS.           *
*                  Set a flag if the identifier is a  null token.       *
*                                                                       *
*               7. Call CSFADSPI to perform the following:              *
*                   - Create the  request CPRB for the function.        *
*                   - Submit the request CPRB to the PCICC              *
*                   - Validate the reply CPRB.                          *
*                                                                       *
*               8. Parse the PIN block from the verb unique data        *
*                  block and copy it to the output parameter.           *
*                                                                       *
*               9. If the return/reason code indicates that a token *
```

```
*                 was enciphered under the old master key, parse   *
*                 the key block and copy the reenciphered keys     *
*                 to the appropriate identifiers. Labels are not   *
*                 updated.                                         *
*                                                                  *
*             10. Return to caller.                                *
*                                                                  *
*   Outputs:  Return Code       Return code from processing        *
*                                                                  *
*             Reason Code       Reason code from processing        *
*                                                                  *
*             Output Pin Block  Processed enciphered PIN block     *
*                                                                  *
*             Input Pin Key Id  Reenciphered token if the key was  *
*                               enciphered under the old master key.*
*                                                                  *
*             Output Pin Key Id Reenciphered token if the key was  *
*                               enciphered under the old master key.*
*                                                                  *
*   External routines:                                             *
*                                                                  *
*     CSFACKDS - Retrieve a DES key from the CKDS                  *
*     CSFADSPI - Invoke PCICC                                      *
*     CSFASEC  - Check authorization to a RACF-protected           *
*                or security-exit-protected resource               *
*                                                                  *
*   External executable macros:                                    *
*                                                                  *
*     CSFAGET  - Obtain dynamic storage                            *
*     CSFAFREE - Release dynamic storage                           *
*                                                                  *
*   Register usage:                                                *
*                                                                  *
*     R0      = Work register                                      *
*     R1      = Address of parameter lists                         *
*     R2-R6   = Work registers                                     *
*     R7      = CCVT address                                       *
*     R8      = SPB  address                                       *
*     R9      = GSVT address and work register                     *
*     R10     = CCVE address and work register                     *
*     R11     = Dynamic data area address                          *
*     R12     = Module base register                               *
*     R13     = unused                                             *
*     R14-R15 = Linkage registers                                  *
*                                                                  *
*   Change History                                                 *
*                                                                  *
*   Date      Programmer  Description                              *
*   --------  ----------  -----------                              *
*   09/26/99  kbk         Created                                  *
*   01/31/01  mce         Converted to Assembler language          *
*                                                                  *
********************************************************************
        EJECT
        MACRO
&LABEL  COPYPARMS  &SOURCE=,&SRCALET=,&COPYLEN=,&TARGET=,&TGTALET=,   *
              &COPYDIR=
        L     R10,&SOURCE
        ST    R10,SOURCE_ADDR
        LA    R10,&SRCALET
        ST    R10,SOURCE_ALET
        L     R10,&COPYLEN
        ST    R10,COPY_LENGTH
        L     R10,&TARGET
        ST    R10,TARGET_ADDR
        LA    R10,&TGTALET
        ST    R10,TARGET_ALET
        L     R10,&COPYDIR
        ST    R10,COPY_DIRECTION
        BAS   R14,COPY
        MEND
```

```
         EJECT
*
*********************************************************************
*     Main entry for module
*********************************************************************
*
MAINENT  DS    0H
         USING *,R15
         B     PROLOG
         DC    AL1(18)
         DC    C'ZUDXSVC   2001.031'
         DROP  R15
PROLOG   BSM   R14,0
         BAKR  R14,0
         LAE   R12,0
         LR    R12,R15
PSTART   EQU   ZUDXSVC
         USING PSTART,R12
         LAE   R10,0(,R1)
         L     R0,DYNDATA_SIZE
         LA    R15,0
         CPYA  AR1,AR12
         SAC   512
         CSFAGET  OBTAIN,LENGTH=(R0),SP=(R15),LINKAGE=SYSTEM
         LAE   R11,0(,R1)
         USING DATD,R11
         EREG  R14,R1
         CPYA  AR1,AR10
         MVC   PARAMETER_LIST(PARMLLEN),0(R1)
         LR    R8,R0
         CPYA  AR8,AR12
         USING SPB,R8
*********************************************************************
*     Save the address of the caller's parameter block              *
*********************************************************************
         LR    R6,R1
*
*********************************************************************
*     Clear the processing flags and local variables
*********************************************************************
*
         XC    PROCESSING_FLAGS(1),PROCESSING_FLAGS
         SLR   R7,R7
         ST    R7,LOCAL_REASON_CODE
         ST    R7,LOCAL_RETURN_CODE
         ST    R7,SERVICE_RC
         ST    R7,SERVICE_RS
*
MAINPROC DS    0H
*********************************************************************
*     Check the environment and copy the parameters to local storage.*
*********************************************************************
*
         BAS   R14,VALIDATE_AND_COPY
         L     R9,LOCAL_RETURN_CODE
         LTR   R9,R9
         BP    ENDMAIN
*
*********************************************************************
*     Process the key identifiers.                                  *
*********************************************************************
*
         BAS   R14,PROCESS_KEYS
         L     R10,LOCAL_RETURN_CODE
         LTR   R10,R10
         BP    ENDMAIN
*
*********************************************************************
*     Process the request.                                          *
*********************************************************************
*
         BAS   R14,PROCESS_REQUEST
```

```
         L     R9,LOCAL_RETURN_CODE
         LTR   R9,R9
         BP    ENDMAIN
*
*************************************************************************
*     Copy the output PIN block to the output parameter.                *
*                                                                       *
*     Call the COPY routine with parameters as follows:                 *
*           Address of returned encrypted PIN block (PIN_BLOCK@)        *
*           ICSF's ALET                                                 *
*           Length of LOCAL_OUTPUT_PIN_BLOCK                            *
*           OUTPUT_PIN_BLOCK@                                           *
*           Application ALET                                            *
*           TO_CALLER                                                   *
*************************************************************************
*
         COPYPARMS  SOURCE=PIN_BLOCK@,SRCALET=ICSF_ALET,               *
               COPYLEN=PIN_BLOCK_LEN,TARGET=OUTPUT_PIN_BLOCK@,         *
               TGTALET=APPL_ALET,COPYDIR=TO_CALLER
*************************************************************************
*     Check return and reason code to see if a token was enciphered     *
*     under the old master key. (RC=0, RS=10001)                        *
*     (We assume any necessary reencipherment was performed by          *
*     the UDX code in the PCICC.)                                       *
*      If so, copy the reenciphered tokens back to the caller.          *
*************************************************************************
*
         CLC   LOCAL_REASON_CODE(4),OMK_TOKEN_USED
         BNE   ENDMAIN
*
*************************************************************************
*     Process the input pin key id first. (Only process the key         *
*     if the input key identifier is not a label.)                      *
*************************************************************************
*
         TM    PROCESSING_FLAGS,INPUT_KEY_IS_LABEL
         BNZ   CHKOUTKY
*
*************************************************************************
*     Copy the (possibly) reenciphered input PIN key back to the        *
*     caller's parameter area.                                          *
*                                                                       *
*     Call the COPY routine with parameters as follows:                 *
*           Address of LOCAL_INPUT_KEY_TOKEN                            *
*           ICSF's ALET                                                 *
*           Length of LOCAL_INPUT_KEY_TOKEN                             *
*           INPUT_PIN_KEY_ID@                                           *
*           Application ALET                                            *
*           TO_CALLER                                                   *
*************************************************************************
*
         LA    R10,LOCAL_INPUT_KEY_TOKEN
         ST    R10,SOURCE_ADDRESS
         COPYPARMS  SOURCE=SOURCE_ADDRESS,SRCALET=ICSF_ALET,           *
               COPYLEN=TOKEN_LENGTH,TARGET=INPUT_PIN_KEY_ID@,          *
               TGTALET=APPL_ALET,COPYDIR=TO_CALLER
*************************************************************************
*     Now process the output pin key id. (Only process the key          *
*     if the key identifier is not a label and the output key is        *
*     not null.)                                                        *
*************************************************************************
*
CHKOUTKY TM    PROCESSING_FLAGS,OUTPUT_KEY_IS_LABEL
         BNZ   ENDMAIN
         TM    PROCESSING_FLAGS,OUTPUT_KEY_IS_NULL
         BNZ   ENDMAIN
*
*************************************************************************
*     Copy the (possibly) reenciphered output PIN key back to the       *
*     caller's parameter area.                                          *
*                                                                       *
*     Call the COPY routine with parameters as follows:                 *
```

```
*           Address of LOCAL_OUTPUT_KEY_TOKEN                       *
*           ICSF's ALET                                             *
*           Length of LOCAL_OUTPUT_KEY_TOKEN                        *
*           OUTPUT_PIN_KEY_ID@                                      *
*           Application ALET                                        *
*           TO_CALLER                                               *
*********************************************************************
*
         LA    R10,LOCAL_OUTPUT_KEY_TOKEN
         ST    R10,SOURCE_ADDRESS
         COPYPARMS  SOURCE=SOURCE_ADDRESS,SRCALET=ICSF_ALET,          *
               COPYLEN=TOKEN_LENGTH,TARGET=OUTPUT_PIN_KEY_ID@,        *
               TGTALET=APPL_ALET,COPYDIR=TO_CALLER
*
ENDMAIN  EQU   *
*
*********************************************************************
*     Return to the caller with return reason in register 0.        *
*********************************************************************
*
*   Set RETURN_CODE equal to LOCAL_RETURN_CODE
         L     R14,LOCAL_RETURN_CODE
         LA    R10,APPL_ALET
         SAR   AR15,R10
         L     R15,RETURN_CODE@
         ST    R14,0(,R15)
*   Set REASON_CODE equal to LOCAL_REASON_CODE
         L     R15,LOCAL_REASON_CODE
         SAR   AR10,R10
         L     R10,REASON_CODE@
         ST    R15,0(,R10)
*   Free dynamic storage and return to caller with return code
*   in register 15.
         LR    R10,R14               Save return code around call
         LR    R3,R15                Save reason code around call
         CPYA  AR3,AR0
         L     R0,DYNDATA_SIZE       Size of area to free
         LA    R15,0
         LR    R1,R11
         CPYA  AR1,AR11
         CSFAFREE RELEASE,LENGTH=(R0),ADDR=(R1),SP=(R15),LINKAGE=SYSTEM
         LR    R0,R3                 Reason code into register 0
         CPYA  AR0,AR3
         LR    R15,R10               Return code into register 10
         PR
         EJECT
*
*********************************************************************
*     Subroutines                                                   *
*********************************************************************
*
*********************************************************************
*     Validate and Copy                                             *
*                                                                   *
*     Explicit Inputs: none                                         *
*                                                                   *
*     Implicit Inputs: CCVT     Crypto communication vector table   *
*                                                                   *
*                      PARAMETER_LIST                               *
*                                                                   *
*                      All input parameters                         *
*                                                                   *
*     Process: 1. Validate that ICSF is running on CMOS hardware.   *
*                                                                   *
*              2. Check that at least one PCICC is active.          *
*                                                                   *
*              3. Call RACF (or equivalent) to see if the caller is *
*                 authorized to run this program.                   *
*                                                                   *
*              4. Copy the parameter address list into ICSF storage *
*                 using the caller's key.                           *
*                                                                   *
```

```
*              5. Copy the input parameters to local storage using   *
*                 the caller's key.                                  *
*                                                                    *
*     Explicit Outputs: None                                         *
*                                                                    *
*     Implicit Outputs: LOCAL_RETURN_CODE                            *
*                                                                    *
*                       LOCAL_REASON_CODE                            *
*                                                                    *
*                       LOCAL_RULE_ARRAY_COUNT                       *
*                                                                    *
*                       LOCAL_RULE_ARRAY                             *
*                                                                    *
*                       LOCAL_INPUT_KEY_ID                           *
*                                                                    *
*                       LOCAL_INPUT_PIN_BLOCK                        *
*                                                                    *
*                       LOCAL_EXTRA_DATA                             *
*                                                                    *
*                       LOCAL_OUTPUT_KEY_ID                          *
*                                                                    *
**********************************************************************
*
VALIDATE_AND_COPY EQU   *
         STM   R14,R12,SAVEAREA
*
         STAM  AR14,AR12,SAVEAREA+60
*
**********************************************************************
* Check the environment.                                             *
*                                                                    *
*   Check that ICSF is active (Bit CCVTMK in the CCVT is equal to 1) *
**********************************************************************
*
         L     R7,SPBCCVT
         USING CCVT,R7
         CPYA  AR7,AR12
         TM    CCVTSFG1,CCVTMK
         BO    CHKCCP
*        If ICSF is not active, set LOCAL_RETURN_CODE to 12 and
*           LOCAL_REASON_CODE to 0
         LA    R10,RC_CSF_ERROR
         ST    R10,LOCAL_RETURN_CODE
         SLR   R10,R10
         ST    R10,LOCAL_REASON_CODE
         B     ENDVAL
*
**********************************************************************
*      Check that at least one CCP is available for the service.     *
*       (Bit CCVTCCP in the CCVT is equal to 1)                      *
**********************************************************************
*
CHKCCP   EQU   *
         TM    CCVTFLAG,CCVTCCP
         BO    CHKAUTH
*        If no CCP is available, set LOCAL_RETURN_CODE to 12 and
*           LOCAL_REASON_CODE to 11060
         LA    R10,RC_CSF_ERROR
         ST    R10,LOCAL_RETURN_CODE
         MVC   LOCAL_REASON_CODE(4),RS_12_CCP_NOT_AVAILABLE
         B     ENDVAL
*
**********************************************************************
*      Call CSFASEC to see if the caller is authorized to use this   *
*      program.                                                      *
*                                                                    *
*     Call CSFASEC with parameters as follows:                       *
*           LOCAL_RETURN_CODE                                        *
*           LOCAL_REASON_CODE                                        *
*           RESOURCE_NAME                                            *
*           RESOURCE_LENGTH                                          *
*           ASEC_CSFSERV                                             *
```

```
*           SPB                                                     *
*********************************************************************
*
CHKAUTH  EQU   *
         XC    PARMS_FOR_CALL,PARMS_FOR_CALL
         LA    R7,LOCAL_RETURN_CODE
         ST    R7,PARMS_FOR_CALL
         LA    R9,LOCAL_REASON_CODE
         ST    R9,PARMS_FOR_CALL+4
         LA    R10,RESNAME
         ST    R10,PARMS_FOR_CALL+8
         LA    R7,RESLEN
         ST    R7,PARMS_FOR_CALL+12
         LA    R9,RESCLASS
         ST    R9,PARMS_FOR_CALL+16
         ST    R8,PARMS_FOR_CALL+20          SPB pointer
         L     R7,SPBCCVT
         L     R10,CCVTCCVE
         USING CCVE,R10
         SLR   R9,R9
         SAR   AR10,R9
         L     R9,CCVEGSVT
         CPYA  AR9,AR10
         USING GSVT,R9
         L     R15,GSVT_ASEC
         LAE   R1,PARMS_FOR_CALL
         BALR  R14,R15               Branch to CSFASEC
*     If CSFASEC returns a return code greater than zero,
*       return to caller with CSFASEC's return code.
         L     R7,LOCAL_RETURN_CODE
         LTR   R7,R7
         BP    ENDVAL
*
*********************************************************************
*     Copy the parameter address list into ICSF's address space.    *
*********************************************************************
*
         SLR   R1,R1
         IC    R1,SPBPSWKY             PSW key into register 1
         LA    R2,PARAMETER_LIST       Target address
         SLR   R3,R3
         SAR   AR2,R3                  Target space
         LR    R4,R6                   Source address (address of
*                                      caller's parameter block)
         LA    R5,APPL_ALET            Caller's ALET
         SAR   AR4,R5                  Target space
         LA    R0,PARMLLEN             Length in register 0
         BCTR  R0,0
         OI    SPBF1,SPBTERM           Set recovery flag
         MVCSK 0(R2),0(R4)
         NI    SPBF1,X'FF'-SPBTERM     Reset recovery flag
*
*********************************************************************
*     Copy the parameters to local storage.                         *
*                                                                   *
*     Call the COPY routine with parameters as follows:             *
*           Address of caller's parameter to be copied              *
*           Application ALET                                        *
*           Length of local area to hold copied parameter           *
*           Address of local area to hold the copied parameter      *
*           ICSF's ALET                                             *
*           Direction to copy: TO_ICSF                              *
*********************************************************************
*
*********************************************************************
*     Copy the Rule Array Count                                     *
*********************************************************************
*
         LA    R10,LOCAL_RULE_ARRAY_COUNT
         ST    R10,TARGET_ADDRESS
         COPYPARMS  SOURCE=RULE_ARRAY_COUNT@,SRCALET=APPL_ALET,       *
               COPYLEN=WORDSIZE,TARGET=TARGET_ADDRESS,                *
```

```
               TGTALET=ICSF_ALET,COPYDIR=TO_ICSF
*
*********************************************************************
*     Validate the rule array count.                                *
*        LOCAL_RULE_ARRAY_COUNT must be 0, 1, or 2                  *
*********************************************************************
*
         L     R10,LOCAL_RULE_ARRAY_COUNT
         LTR   R10,R10
         BM    BADCOUNT
         LA    R9,MAX_RULE_COUNT
         CR    R10,R9
         BNH   COPYRA
BADCOUNT DS    0H
* Return to caller with return code = 8, reason code = 2012
         LA    R10,RC_APPLICATION_ERROR
         ST    R10,LOCAL_RETURN_CODE
         LA    R10,RS_8_IV_RA_COUNT
         ST    R10,LOCAL_REASON_CODE
         B     ENDVAL
*
*********************************************************************
*     Copy the Rule Array                                           *
*********************************************************************
*
COPYRA   EQU   *
         L     R9,LOCAL_RULE_ARRAY_COUNT
         LTR   R9,R9                 Are there rules to copy?
         BZ    COPYIPIN              No rules
         SLA   R9,3                  Multiply LOCAL_RULE_ARRAY_COUNT
*                                     by 8
         ST    R9,RULES_LENGTH
         LA    R10,LOCAL_RULE_ARRAY
         ST    R10,TARGET_ADDRESS
         COPYPARMS  SOURCE=RULE_ARRAY@,SRCALET=APPL_ALET,              *
               COPYLEN=RULES_LENGTH,TARGET=TARGET_ADDRESS,             *
               TGTALET=ICSF_ALET,COPYDIR=TO_ICSF
*
*********************************************************************
*     Input PIN Key Id                                              *
*********************************************************************
*
COPYIPIN EQU   *
         LA    R10,LOCAL_INPUT_KEY_ID
         ST    R10,TARGET_ADDRESS
         COPYPARMS  SOURCE=INPUT_PIN_KEY_ID@,SRCALET=APPL_ALET,        *
               COPYLEN=TOKEN_LENGTH,TARGET=TARGET_ADDRESS,             *
               TGTALET=ICSF_ALET,COPYDIR=TO_ICSF
         MVC   LOCAL_INPUT_KEY_TOKEN,LOCAL_INPUT_KEY_ID
*
*********************************************************************
*     Input PIN Block                                               *
*********************************************************************
*
COPYIPBK EQU   *
         LA    R10,LOCAL_INPUT_PIN_BLOCK
         ST    R10,TARGET_ADDRESS
         COPYPARMS  SOURCE=INPUT_PIN_BLOCK@,SRCALET=APPL_ALET,         *
               COPYLEN=PIN_BLOCK_LEN,TARGET=TARGET_ADDRESS,            *
               TGTALET=ICSF_ALET,COPYDIR=TO_ICSF
*
*********************************************************************
*     Extra Data                                                    *
*********************************************************************
*
         LA    R10,LOCAL_EXTRA_DATA
         ST    R10,TARGET_ADDRESS
         COPYPARMS  SOURCE=EXTRA_DATA@,SRCALET=APPL_ALET,              *
               COPYLEN=EXTRA_DATA_LEN,TARGET=TARGET_ADDRESS,           *
               TGTALET=ICSF_ALET,COPYDIR=TO_ICSF
*
*********************************************************************
```

```
*     Output PIN Key Id                                             *
*********************************************************************
*
COPYOPIN EQU   *
         LA    R10,LOCAL_OUTPUT_KEY_ID
         ST    R10,TARGET_ADDRESS
         COPYPARMS  SOURCE=OUTPUT_PIN_KEY_ID0,SRCALET=APPL_ALET,       *
               COPYLEN=TOKEN_LENGTH,TARGET=TARGET_ADDRESS,             *
               TGTALET=ICSF_ALET,COPYDIR=TO_ICSF
         MVC   LOCAL_OUTPUT_KEY_TOKEN,LOCAL_OUTPUT_KEY_ID
*
ENDVAL   DS    0H
         LM    R14,R12,SAVEAREA
         LAM   AR14,AR12,SAVEAREA+60
         BR    R14                     Return to caller
*
*
*********************************************************************
*     Process Keys                                                  *
*                                                                   *
*     Function: Check the input key identifiers. Retrieve the token  *
*               from the CKDS if the identifier is a label.         *
*                                                                   *
*     Explicit Inputs: None                                         *
*                                                                   *
*     Implicit Inputs: INPUT_PIN_KEY_IDENTIFIER                     *
*                                                                   *
*                      OUTPUT_PIN_KEY_IDENTIFIER                    *
*                                                                   *
*     Process: 1. Check the INPUT_PIN_KEY_IDENTIFIER.               *
*                                                                   *
*              2. Check the OUTPUT_PIN_KEY_IDENTIFIER.              *
*                                                                   *
*     Explicit Outputs: None                                        *
*                                                                   *
*     Implicit Outputs: LOCAL_RETURN_CODE                           *
*                                                                   *
*                       LOCAL_REASON_CODE                           *
*                                                                   *
*                       LOCAL_INPUT_KEY_TOKEN                       *
*                                                                   *
*                       LOCAL_OUTPUT_KEY_TOKEN                      *
*                                                                   *
*                       OUTPUT_KEY_IS_LABEL                         *
*                                                                   *
*                       INPUT_KEY_IS_LABEL                          *
*                                                                   *
*                       OUTPUT_KEY_IS_NULL                          *
*                                                                   *
*********************************************************************
*
PROCESS_KEYS EQU   *
         STM   R14,R12,SAVEAREA
*
         STAM  AR14,AR12,SAVEAREA+60
*
*********************************************************************
*                                                                   *
*    Process the input PIN encrypting key. Check to see if the      *
*    identifier is a label. The first character must be greater     *
*    than a blank if it is a label.                                 *
*                                                                   *
*********************************************************************
*
         LA    R9,LOCAL_INPUT_KEY_ID
         SLR   R10,R10
         SAR   AR9,R10
         CLI   0(R9),FIRST_LABEL_CHAR
         BNH   NOTLABEL
         OI    PROCESSING_FLAGS,INPUT_KEY_IS_LABEL  Set flag
*********************************************************************
*     Identifier is a label, call CSFACKDS to retrieve the key.     *
```

```
*                                                                     *
*     Call CSFACKDS with parameters as follows:                       *
*          LOCAL_RETURN_CODE                                          *
*          LOCAL_REASON_CODE                                          *
*          NULL_EXIT_LENGTH                                           *
*          NULL_EXIT_DATA                                             *
*          ACKDS_ENTRY_TOKEN                                          *
*          LOCAL_INPUT_KEY_ID                                         *
*          ACKDS_TYPE_ANY                                             *
*          LOCAL_INPUT_KEY_TOKEN                                      *
*          SPB                                                        *
***********************************************************************
*
         XC    PARMS_FOR_CALL,PARMS_FOR_CALL
         LA    R10,LOCAL_RETURN_CODE
         ST    R10,PARMS_FOR_CALL
         LA    R10,LOCAL_REASON_CODE
         ST    R10,PARMS_FOR_CALL+4
         LA    R10,NULL_EXIT_LENGTH
         ST    R10,PARMS_FOR_CALL+8
         LA    R10,NULL_EXIT_DATA
         ST    R10,PARMS_FOR_CALL+12
         LA    R10,ACKDS_ENTRY_TOKEN
         ST    R10,PARMS_FOR_CALL+16
         ST    R9,PARMS_FOR_CALL+20
         LA    R9,TYPEANY
         ST    R9,PARMS_FOR_CALL+24
         LA    R10,LOCAL_INPUT_KEY_TOKEN
         ST    R10,PARMS_FOR_CALL+28
         ST    R8,PARMS_FOR_CALL+32
         L     R7,SPBCCVT
         L     R10,CCVTCCVE
         SLR   R9,R9
         SAR   AR10,R9
         L     R9,CCVEGSVT
         CPYA  AR9,AR10
         L     R15,GSVT_ACKDS
         LAE   R1,PARMS_FOR_CALL
         BALR  R14,R15                Call CSFACKDS
*     If CSFACKDS returns a return code greater than zero,
*       return to caller with CSFACKDS's return code.
NOTLABEL L     R9,LOCAL_RETURN_CODE
         LTR   R9,R9
         BP    ENDPRKEY
*
***********************************************************************
*    Process the output PIN encrypting key. Check to see if the       *
*    identifier is a null token. The first character will be '00'x.   *
***********************************************************************
*
         LA    R9,LOCAL_OUTPUT_KEY_ID
         CLI   0(R9),X'00'
         BNE   NOTNULL
         OI    PROCESSING_FLAGS,OUTPUT_KEY_IS_NULL   Set flag
         B     ENDPRKEY
*
NOTNULL  DS    0H
***********************************************************************
*      Check to see if the identifier is a label. The first           *
*      character must be greater than a blank.                        *
***********************************************************************
*
         CLI   0(R9),FIRST_LABEL_CHAR
         BNH   ENDPRKEY
         OI    PROCESSING_FLAGS,OUTPUT_KEY_IS_LABEL   Set flag
***********************************************************************
*     Identifier is a label, call CSFACKDS to retrieve the key.       *
*                                                                     *
*     Call CSFACKDS with parameters as follows:                       *
*          LOCAL_RETURN_CODE                                          *
*          LOCAL_REASON_CODE                                          *
*          NULL_EXIT_LENGTH                                           *
```

```
*           NULL_EXIT_DATA                                      *
*           ACKDS_ENTRY_TOKEN                                   *
*           LOCAL_OUTPUT_KEY_ID                                 *
*           ACKDS_TYPE_ANY                                      *
*           LOCAL_OUTPUT_KEY_TOKEN                              *
*           SPB                                                 *
************************************************************************
*
         XC    PARMS_FOR_CALL,PARMS_FOR_CALL
         LA    R10,LOCAL_RETURN_CODE
         ST    R10,PARMS_FOR_CALL
         LA    R10,LOCAL_REASON_CODE
         ST    R10,PARMS_FOR_CALL+4
         LA    R10,NULL_EXIT_LENGTH
         ST    R10,PARMS_FOR_CALL+8
         LA    R10,NULL_EXIT_DATA
         ST    R10,PARMS_FOR_CALL+12
         LA    R10,ACKDS_ENTRY_TOKEN
         ST    R10,PARMS_FOR_CALL+16
         ST    R9,PARMS_FOR_CALL+20
         LA    R9,TYPEANY
         ST    R9,PARMS_FOR_CALL+24
         LA    R10,LOCAL_OUTPUT_KEY_TOKEN
         ST    R10,PARMS_FOR_CALL+28
         ST    R8,PARMS_FOR_CALL+32
         L     R7,SPBCCVT
         L     R10,CCVTCCVE
         SLR   R9,R9
         SAR   AR10,R9
         L     R9,CCVEGSVT
         CPYA  R9,R10
         L     R15,GSVT_ACKDS
         LAE   R1,PARMS_FOR_CALL
         BALR  R14,R15
ENDPRKEY DS    0H
         LM    R14,R12,SAVEAREA
         LAM   AR14,AR12,SAVEAREA+60
         BR    R14
*
*
************************************************************************
*     Process Request                                                  *
*                                                                      *
*     Explicit Inputs: none                                            *
*                                                                      *
*     Implicit Inputs: LOCAL_RULE_ARRAY_COUNT                          *
*                                                                      *
*                      LOCAL_RULE_ARRAY                                *
*                                                                      *
*                      LOCAL_EXTRA_DATA                                *
*                                                                      *
*                      LOCAL_INPUT_PIN_BLOCK                           *
*                                                                      *
*                      LOCAL_INPUT_KEY_TOKEN                           *
*                                                                      *
*                      LOCAL_OUTPUT_KEY_TOKEN                          *
*                                                                      *
*     Process: Call CSFADSPI to perform the following:                 *
*                                                                      *
*              1. Create the CPRB for the request.                     *
*                                                                      *
*              2. Submit the request to the PCICC.                     *
*                                                                      *
*              3. Validate the reply CPRB.                             *
*                                                                      *
*              4. Parse the reply parameter block.                     *
*                                                                      *
*     Explicit Outputs: None                                           *
*                                                                      *
*     Implicit Outputs: LOCAL_RETURN_CODE                              *
*                                                                      *
*                       LOCAL_REASON_CODE                              *
```

```
*                                                                     *
*                       PIN_BLOCK@                                    *
*                                                                     *
*                       LOCAL_INPUT_KEY_TOKEN                         *
*                                                                     *
*                       LOCAL_OUTPUT_KEY_TOKEN                        *
*                                                                     *
***********************************************************************
*
PROCESS_REQUEST EQU   *
         STM   R14,R12,SAVEAREA
         STAM  AR14,AR12,SAVEAREA+60
*
         SLR   R9,R9
         ST    R9,PIN_BLOCK@  Clear variable to hold address of
*                             returned PIN Block
***********************************************************************
*     Set up parameters for CSFADSPI invocation                       *
***********************************************************************
*
*     Set flags to zero
         XC    ADSPI_FLAGS,ADSPI_FLAGS
*
*     Set PCICC_INDEX to -1 (ANY PCICC)
         SLR   R10,R10
         BCTR  R10,0
         ST    R10,PCICC_INDEX
*
*     Set PCICC_SERIAL_NUMBER to "NOT APPL"
         MVC   PCICC_SERIAL_NUMBER(8),SERIAL_NUMBER_NOT_APPLICABLE
*
*     Set PCICC_DOMAIN to -1 (Not Applicable)
         SLR   R10,R10
         BCTR  R10,0
         ST    R10,PCICC_DOMAIN
*
*     Set ALET to ICSF's ALET
         LA    R10,ICSF_ALET
         ST    R10,SOURCE_ALET
*
*     Set up VUD List
         LA    R10,VUD_ELEMENT_LENGTH
         ST    R10,VUD1_LENGTH
         ST    R10,VUD2_LENGTH
         L     R10,VUD_FLAGS
         ST    R10,VUD1_FLAG
         ST    R10,VUD2_FLAG
         LA    R10,LOCAL_INPUT_PIN_BLOCK
         ST    R10,VUD1_STRING
         LA    R10,LOCAL_EXTRA_DATA
         ST    R10,VUD2_STRING
*
*     Set up KEY List
         L     R10,TOKEN_LENGTH
         ST    R10,KEY1_LENGTH
         ST    R10,KEY2_LENGTH
         L     R10,KEY_FLAGS
         ST    R10,KEY1_FLAG
         ST    R10,KEY2_FLAG
         LA    R10,LOCAL_INPUT_KEY_TOKEN
         ST    R10,KEY1_STRING
         LA    R10,LOCAL_OUTPUT_KEY_TOKEN
         ST    R10,KEY2_STRING
*
         LA    R10,UDX_KEYNUM
         ST    R10,NUMBER_OF_KEYS
         XC    RETURNED_KEY_BLOCK(138),RETURNED_KEY_BLOCK
         LA    R9,RETURNED_KEY_BLOCK
         ST    R9,RETURNED_KEY_BLOCK_ADDRESS
*
         LA    R10,UDX_VUDNUM
         ST    R10,NUMBER_OF_VUDS
```

```
         XC    RETURNED_VUD_BLOCK(10),RETURNED_VUD_BLOCK
         LA    R10,RETURNED_VUD_BLOCK
         ST    R10,RETURNED_VUD_BLOCK_ADDRESS
*
*************************************************************************
*     Submit the request.                                               *
*************************************************************************
*
         LA    R10,SERVICE_RC
         ST    R10,ADSPI_RC_@
         LA    R10,SERVICE_RS
         ST    R10,ADSPI_RS_@
         LA    R10,ADSPI_FLAGS
         ST    R10,ADSPI_FLAGS_@
         LA    R10,FUNCTION_CODE_UDX390
         ST    R10,ADSPI_SUBFUNC_@
         LA    R10,PCICC_INDEX
         ST    R10,ADSPI_CCP_INDX_@
         LA    R10,PCICC_SERIAL_NUMBER
         ST    R10,ADSPI_CCP_SN_@
         LA    R10,PCICC_DOMAIN
         ST    R10,ADSPI_CCP_DOM_@
         LA    R10,SOURCE_ALET
         ST    R10,ADSPI_ALET_@
         LA    R10,LOCAL_RULE_ARRAY_COUNT
         ST    R10,ADSPI_RULES_COUNT_@
         LA    R10,LOCAL_RULE_ARRAY
         ST    R10,ADSPI_RULES_@
         LA    R10,NUMBER_OF_VUDS
         ST    R10,ADSPI_NUM_VUDS_@
         LA    R10,VUD_LIST
         ST    R10,ADSPI_VUD_LIST_PARM_@
         LA    R10,NUMBER_OF_KEYS
         ST    R10,ADSPI_NUM_KEYS_@
         LA    R10,KEY_LIST
         ST    R10,ADSPI_KEY_LIST_PARM_@
         LA    R10,RETURNED_VUD_BLOCK_LENGTH
         ST    R10,ADSPI_VUD_BLOCK_LEN_@
         LA    R10,RETURNED_VUD_BLOCK_ADDRESS
         ST    R10,ADSPI_VUD_BLOCK_PTR_@
         LA    R10,RETURNED_KEY_BLOCK_LENGTH
         ST    R10,ADSPI_KEY_BLOCK_LEN_@
         LA    R10,RETURNED_KEY_BLOCK_ADDRESS
         ST    R10,ADSPI_KEY_BLOCK_PTR_@
         LA    R10,NULL_LENGTH
         ST    R10,ADSPI_REQ_DATA_BLK_LEN_@
         LA    R10,NULL_POINTER
         ST    R10,ADSPI_REQ_DATA_PTR_LEN_@
         LA    R10,NULL_LENGTH
         ST    R10,ADSPI_REP_DATA_BLK_LEN_@
         LA    R10,NULL_POINTER
         ST    R10,ADSPI_REP_DATA_PTR_LEN_@
         ST    R8,ADSPI_SPB_@
         L     R7,SPBCCVT
         L     R10,CCVTCCVE
         SLR   R9,R9
         SAR   AR10,R9
         L     R9,CCVEGSVT
         CPYA  R9,R10
         L     R15,GSVT_ADSPI
         LAE   R1,ADSPI_PARMS
         BALR  R14,R15
*
*     Check return code from CSFADSPI
         L     R10,SERVICE_RC
         LTR   R10,R10
         BNZ   ADSPIERR
*
*************************************************************************
*                                                                       *
*    Process the output if the service completed successfully.          *
*    Parse the output PIN block from the VUD.                           *
```

```
*                                                                     *
*    VUD structure                                                    *
*    ------------------------                                         *
*    | overall | output pin |   The PIN block is eight bytes long.   *
*    | length  | block      |   The overall length should be 10.     *
*    ------------------------                                         *
*     2 bytes    8 bytes                                             *
*                                                                     *
***********************************************************************
*
*     Get address of returned encrypted PIN Block from VUD
*
         LA    R10,RET_ENCRYPTED_PIN_BLOCK
         ST    R10,PIN_BLOCK@
*
***********************************************************************
*                                                                     *
*    If either key was enciphered under the old master key,           *
*    the reenciphered key will be returned in the key block,          *
*    with the input PIN key first and the output PIN key second.      *
*    Check to see if the input PIN key was updated. If the token      *
*    returned is a null token, then the key was not updated.          *
*                                                                     *
***********************************************************************
*
         CLI   RET_KEY1,X'00'
         BE    CHKOPKEY
         MVC   LOCAL_INPUT_KEY_TOKEN(64),RET_KEY1
*
***********************************************************************
*                                                                     *
*    Check to see if the output PIN key was updated. If the           *
*    token returned is a null token, then the key was not             *
*    updated.                                                         *
* ********************************************************************
*
CHKOPKEY DS    0H
         CLI   RET_KEY2,X'00'
         BE    KEYSDONE
         MVC   LOCAL_OUTPUT_KEY_TOKEN(64),RET_KEY2
*
ADSPIERR DS    0H
KEYSDONE DS    0H
***********************************************************************
*                                                                     *
*    Save the return and reason codes.                                *
*                                                                     *
***********************************************************************
*
         L     R10,SERVICE_RC
         ST    R10,LOCAL_RETURN_CODE
         L     R10,SERVICE_RS
         ST    R10,LOCAL_REASON_CODE
*
***********************************************************************
*    Return to caller                                                 *
***********************************************************************
*
ENDPRREQ DS    0H
         LM    R14,R12,SAVEAREA
         LAM   AR14,AR12,SAVEAREA+60
         BR    R14
*
*
***********************************************************************
*     Copy                                                            *
*                                                                     *
*     Function: Copy storage between the caller's address space and   *
*               ICSF's address space in either direction. This        *
*               routine can copy any length of storage. The 'move     *
*               with source key (MVCSK)' and 'move with destination   *
*               key (MVCDK)' instructions are used.                   *
```

```
*                                                                     *
*     Notes: When copying back to the caller, the target address is   *
*            the original address copied from the parameter block.    *
*                                                                     *
*     Explicit Inputs: SOURCE_ADDRESS   Address of the source data    *
*                                                                     *
*                      SOURCE_ALET      ALET of the source data       *
*                                                                     *
*                      COPY_LENGTH      Length of the data            *
*                                                                     *
*                      TARGET_ADDRESS   Address of the target data    *
*                                                                     *
*                      TARGET_ALET      ALET of the target data       *
*                                                                     *
*                      COPY_DIRECTION   Flag indicating which dir-    *
*                                       ection to copy the data.      *
*                                                                     *
*     Implicit Inputs: SPB              Secondary parameter block.    *
*                                                                     *
*     Process: Copy the data from the source ALET to the target ALET  *
*              256 bytes at a time. If copying to the caller's ALET,  *
*              the parameters are compared. If they are equal, then   *
*              the parameter is NOT moved back to user storage. This  *
*              allows the user to pass in write-protected, read-only  *
*              storage as a parameter.                                *
*                                                                     *
*     Explicit Outputs: None                                          *
*                                                                     *
*     Implicit Outputs: None                                          *
*                                                                     *
***********************************************************************
*
COPY     EQU   *
         STM   R14,R12,COPYSAVE
         STAM  AR14,AR12,COPYSAVE+60
         OI    DO_COPY,COPY_YES       Initialize flag to require a
*                                     copy. (A copy is always
*                                     required if the copy direction
*                                     is TO_ICSF.)
         L     R9,COPY_LENGTH         Length to copy
         SLR   R1,R1
         IC    R1,SPBPSWKY             PSW key in register 1
         L     R2,TARGET_ADDR
         L     R5,TARGET_ALET
         SAR   AR2,R5
         L     R4,SOURCE_ADDR
         L     R10,SOURCE_ALET
         SAR   AR4,R10
*
***********************************************************************
*   If copying back to caller ALET, compare the storage               *
*    (There is no need to perform the copy if the area has            *
*     not changed.)                                                   *
***********************************************************************
*
*   See if direction to copy is "TO_CALLER"
*
         L     R10,COPY_DIRECTION
         LTR   R10,R10
         BNZ   ENDCOMP
         LR    R3,R9                  Length to compare
         LR    R5,R9                  Length to compare
         NI    DO_COPY,COPY_NO        Initialize flag
         CLCL  R2,R4                  Compare operands
         BC    8,ENDCOMP              No copy necessary if equal
         OI    DO_COPY,COPY_YES       Set copy flag if not equal
ENDCOMP  DS    0H
*
*   Check if a copy operation is required. A copy must be
*   performed if the DO_COPY flag is on.
*
         TM    DO_COPY,COPY_YES
```

```
         BNO   ENDCOPY
         B     TESTLEN
*
MOVELOOP DS    0H
*********************************************************************
*      Set amount to move. The amount is the lesser of              *
*      the LOCAL_COPY_LENGTH or the MAXIMUM_MOVE amount (256 bytes). *
*      If the LOCAL_COPY_LENGTH is greater than 256 bytes, we will  *
*      loop doing the move for 256 bytes at a time until the        *
*      entire amount has been copied.                               *
*********************************************************************
*
         LA    R0,MAXIMUM_MOVE
         CR    R0,R9
         BNH   USEMAX
         LR    R0,R9
USEMAX   BCTR  R0,0
*********************************************************************
*     MVCxK instruction only moves 'MAXIMUM_MOVE' bytes at a        *
*     time. The instructions require the following input:           *
*     Register 0 is the length. Register 1 is the storage key       *
*     (xxxxxxKx). Register 2 is the target storage. Register 4      *
*     is the source storage.                                        *
*********************************************************************
*
         OI    SPBF1,SPBTERM           Set recovery flag
         LA    R3,COPY_DIRECTION
         CLC   0(4,R3),TO_ICSF
         BNE   TOCALLER
TOICSF   DS    0H             Copy to ICSF's storage
         MVCSK 0(R2),0(R4)
         B     MVCDONE
TOCALLER DS    0H             Copy to caller's storage
         MVCDK 0(R2),0(R4)
MVCDONE  NI    SPBF1,X'FF'-SPBTERM     Reset recovery flag
*
*        Set up to do another MVCxK if necessary
*
         LA    R5,MAXIMUM_MOVE
         ALR   R2,R5                   Next target address for move
         ALR   R4,R5                   Next address to move
         LA    R3,MAXIMUM_MOVE
         LCR   R3,R3
         ALR   R9,R3                   Subtract MAXIMUM_MOVE amount
*                                      from LOCAL_COPY_LENGTH
TESTLEN  LTR   R9,R9                   Is there more to move?
         BP    MOVELOOP
*
ENDCOPY  DS    0H
         LM    R14,R12,COPYSAVE
         LAM   AR14,AR12,COPYSAVE+60
         BR    R14                     Return to caller
*
         DS    0H
         EJECT
*
*********************************************************************
*   DECLARES                                                        *
*********************************************************************
*
DYNDATA_SIZE DS    0A
             DC    A(DYNSIZE)
             DS    0D
*
*********************************************************************
*            LOCAL NON-VARIABLE FIELDS                              *
*********************************************************************
*
TO_ICSF                     DC    F'1'
TO_CALLER                   DC    F'0'
PIN_BLOCK_LEN               DC    F'8'
EXTRA_DATA_LEN              DC    F'8'
```

```
TOKEN_LENGTH                DC    F'64'
WORDSIZE                    DC    F'4'
RESNAME                     DC    CL8'ZUDXSVC '
RESLEN                      DC    F'7'
RESCLASS                    DC    CL8'CSFSERV '
RS_12_CCP_NOT_AVAILABLE     DC    F'11060'
ACKDS_ENTRY_TOKEN           DC    F'1'  CSFACKDS is to return a token
TYPEANY                     DC    CL8'ANY     '
FUNCTION_CODE_UDX390        DC    XL2'5852'
OMK_TOKEN_USED              DC    F'10001'
RETURNED_VUD_BLOCK_LENGTH   DC    F'10'
RETURNED_KEY_BLOCK_LENGTH   DC    F'138'
NULL_POINTER                DC    F'0'
NULL_LENGTH                 DC    F'0'
NULL_EXIT_LENGTH            DC    F'0'
NULL_EXIT_DATA              DC    F'0'
SERIAL_NUMBER_NOT_APPLICABLE DC   CL8'NOT APPL'
VUD_FLAGS                   DC    XL4'FFFFFFFF'
KEY_FLAGS                   DC    XL4'00000000'
*
*********************************************************************
*           REGISTER EQUATES                                        *
*********************************************************************
*
R0    EQU     0
R1    EQU     1
R2    EQU     2
R3    EQU     3
R4    EQU     4
R5    EQU     5
R6    EQU     6
R7    EQU     7
R8    EQU     8
R9    EQU     9
R10   EQU     10
R11   EQU     11
R12   EQU     12
R13   EQU     13
R14   EQU     14
R15   EQU     15
*
*********************************************************************
*           ACCESS REGISTER EQUATES                                 *
*********************************************************************
*
AR0   EQU     0
AR1   EQU     1
AR2   EQU     2
AR3   EQU     3
AR4   EQU     4
AR5   EQU     5
AR6   EQU     6
AR7   EQU     7
AR8   EQU     8
AR9   EQU     9
AR10  EQU     10
AR11  EQU     11
AR12  EQU     12
AR13  EQU     13
AR14  EQU     14
AR15  EQU     15
*
*********************************************************************
*           CONSTANT EQUATES                                        *
*********************************************************************
*
RC_APPLICATION_ERROR    EQU 8
RC_CSF_ERROR            EQU 12
RS_12_SERV_NOTAVAIL     EQU 8
RS_8_IV_RA_COUNT        EQU 2012
APPL_ALET               EQU 1
ICSF_ALET               EQU 0
```

```
MAX_RULE_COUNT          EQU 2
COPY_NO                 EQU B'01111111'
COPY_YES                EQU B'10000000'
MAXIMUM_MOVE            EQU 256
FIRST_LABEL_CHAR        EQU X'40'
OUTPUT_KEY_IS_NULL      EQU X'80'
INPUT_KEY_IS_LABEL      EQU X'40'
OUTPUT_KEY_IS_LABEL     EQU X'20'
UDX_VUDNUM              EQU 2
UDX_KEYNUM              EQU 2
VUD_ELEMENT_LENGTH      EQU 8
*
         LTORG
*
         EJECT
*
*********************************************************************
*            Dynamic Data definitions                               *
*********************************************************************
*
DATD     DSECT
         DS    0F
**
***          Saveareas
**
SAVEAREA DS    30F
COPYSAVE DS    30F
**
***          Parameter lists
**
*
COPY_PLIST DS  6F                Parameter list for COPY_PARMS
         ORG   COPY_PLIST
SOURCE_ADDR    DS A
SOURCE_ALET    DS A
COPY_LENGTH    DS A
TARGET_ADDR    DS A
TARGET_ALET    DS A
COPY_DIRECTION   DS A
*
PARMS_FOR_CALL DS 17A            Parameter list used by internal calls
*
ADSPI_PARMS              DS  0F   Parameter list to call CSFADSPI
ADSPI_RC_@               DS  A
ADSPI_RS_@               DS  A
ADSPI_FLAGS_@            DS  A
ADSPI_SUBFUNC_@          DS  A
ADSPI_CCP_INDX_@         DS  A
ADSPI_CCP_SN_@           DS  A
ADSPI_CCP_DOM_@          DS  A
ADSPI_ALET_@             DS  A
ADSPI_RULES_COUNT_@      DS  A
ADSPI_RULES_@            DS  A
ADSPI_NUM_VUDS_@         DS  A
ADSPI_VUD_LIST_PARM_@    DS  A
ADSPI_NUM_KEYS_@         DS  A
ADSPI_KEY_LIST_PARM_@    DS  A
ADSPI_VUD_BLOCK_LEN_@    DS  A
ADSPI_VUD_BLOCK_PTR_@    DS  A
ADSPI_KEY_BLOCK_LEN_@    DS  A
ADSPI_KEY_BLOCK_PTR_@    DS  A
ADSPI_REQ_DATA_BLK_LEN_@ DS  A
ADSPI_REQ_DATA_PTR_LEN_@ DS  A
ADSPI_REP_DATA_BLK_LEN_@ DS  A
ADSPI_REP_DATA_PTR_LEN_@ DS  A
ADSPI_SPB_@              DS  A
**
***  Caller's parameter address list (input to ZUDXSVC)
**
PARAMETER_LIST     DS CL36
         ORG   PARAMETER_LIST
RETURN_CODE@       DS AL4
```

```
REASON_CODE@       DS AL4
RULE_ARRAY_COUNT@  DS AL4
RULE_ARRAY@        DS AL4
INPUT_PIN_KEY_ID@  DS AL4
INPUT_PIN_BLOCK@   DS AL4
EXTRA_DATA@        DS AL4
OUTPUT_PIN_KEY_ID@ DS AL4
OUTPUT_PIN_BLOCK@  DS AL4
         ORG   PARAMETER_LIST+36
PARMLEND           DS   0X
PARMLLEN  EQU  (PARMLEND-PARAMETER_LIST) Length of list of parameter
*                                        addresses
*
*** Local copies of caller's parameters
*
LOCAL_RETURN_CODE      DS F
LOCAL_REASON_CODE      DS F
LOCAL_RULE_ARRAY_COUNT DS F
LOCAL_RULE_ARRAY       DS CL16
LOCAL_INPUT_KEY_ID     DS CL64
LOCAL_INPUT_PIN_BLOCK  DS CL8
LOCAL_EXTRA_DATA       DS CL8
LOCAL_OUTPUT_KEY_ID    DS CL64
LOCAL_INPUT_KEY_TOKEN  DS CL64
LOCAL_OUTPUT_KEY_TOKEN DS CL64
*
** Other variables
*
SOURCE_ADDRESS    DS A
TARGET_ADDRESS    DS A
SERVICE_RC        DS F
SERVICE_RS        DS F
PCICC_INDEX       DS F
PCICC_DOMAIN      DS F
PIN_BLOCK@        DS A
RULES_LENGTH      DS F
ADSPI_FLAGS       DS BL32
NUMBER_OF_VUDS    DS F
NUMBER_OF_KEYS    DS F
DO_COPY                   DS BL8
RETURNED_KEY_BLOCK_ADDRESS DS A
RETURNED_VUD_BLOCK_ADDRESS DS A
*
RETURNED_VUD_BLOCK DS CL10
         ORG   RETURNED_VUD_BLOCK
RET_VUD_LENGTH DS CL2
RET_ENCRYPTED_PIN_BLOCK DS CL8
         ORG   RETURNED_VUD_BLOCK+10
*
RETURNED_KEY_BLOCK DS CL138
         ORG   RETURNED_KEY_BLOCK
RET_KEY__OVERALL_BLK_LEN DS CL2
RET_KEY1_LENGTH DS CL2
RET_KEY1_FLAG DS CL2
RET_KEY1 DS    CL64
RET_KEY2_LENGTH DS CL2
RET_KEY2_FLAG DS CL2
RET_KEY2 DS    CL64
         ORG   RETURNED_KEY_BLOCK+138
*
PROCESSING_FLAGS DS CL1
SUBFUNCTION_CODE DS CL2
PCICC_SERIAL_NUMBER DS CL8
*
VUD_LIST     DS  6F
         ORG   VUD_LIST
VUD1_LENGTH  DS  F
VUD1_FLAG    DS  XL2
VUD1_NO_FLAG DS  XL2
VUD1_STRING  DS  A
VUD2_LENGTH  DS  F
VUD2_FLAG    DS  XL2
```

```
VUD2_NO_FLAG DS  XL2
VUD2_STRING  DS  A
*
KEY_LIST     DS  6F
         ORG   KEY_LIST
KEY1_LENGTH  DS  F
KEY1_FLAG    DS  XL2
             DS  CL2
KEY1_STRING  DS  A
KEY2_LENGTH  DS  F
KEY2_FLAG    DS  XL2
             DS  CL2
KEY2_STRING  DS  A
*
         ORG   *+1-(*-DATD)/(*-DATD)
ENDDATD DS    0X
DYNSIZE EQU   ((ENDDATD-DATD+7)/8)*8
* THE FOLLOWING INSTRUCTION WILL CAUSE AN ASSEMBLY ERROR IF THE
* SIZE OF THE AUTOMATIC STORAGE AREA IS GREATER THAN 12288 BYTES.
* (12288 bytes is the maximum that can be obtained/released by
*  CSFAGET/CSFAFREE.)
         ORG   DATD+(12288-(*-DATD))
         ORG   ENDDATD
*
*
*********************************************************************
*            Mapping macros                                         *
*********************************************************************
*
         EJECT
         CSFCCVT
         EJECT
         CSFCCVE
         EJECT
         CSFASPB
         EJECT
         CSFGSVT
         EJECT
        END ZUDXSVC
```

Appendix B. UDX Sample Code - Host Piece - Service Stub

This appendix contains a listing of the sample file *zudxstub.bal*. This file is a skeleton for the design of the host piece of a CCA extension.

```
**** START OF SPECIFICATIONS ****************************************  00010015
*                                                                   *  00020015
* MODULE NAME = UDXSTUB                                             *  00030062
* DESCRIPTIVE NAME = UDX Service Stub Sample                        *  00040062
*                                                                   *  00050015
*  FUNCTION =                                                       *  00050115
* THIS IS A SAMPLE SERVICE STUB. IT IS MEANT TO BE LINKEDITED       *  00050215
* WITH THE APPLICATION AND ENTERED VIA A CALL (CALL UDXSTUB). THIS  *  00050362
* STUB CAUSES THE EXECUTION OF THE SERVICE WITH SERVICE NUMBER = 49 *  00050415
*  (DECIMAL).                                                       *  00050515
*                                                                   *  00050615
* MODULE TYPE = ASSEMBLER                                           *  00050715
* PROCESSOR   = ASSEMBLER                                           *  00050815
* MODULE SIZE = ONE BASE REGISTER                                   *  00050915
*                                                                   *  00051015
* TO MODIFY THE STUB FOR USE:                                       *  00051115
* 1.  CHANGE THE VALUE OF 'UDXNUM' FROM 49 TO THE DECIMAL NUMBER OF *  00051215
*     YOUR SERVICE.                                                 *  00051315
* 2.  ISSUE A GLOBAL CHANGE TO CHANGE THE STUB NAME OF 'UDXSTUB' TO *  00051462
*     THE NAME OF YOUR SERVICE STUB.                                *  00051515
*                                                                   *  00051615
**** END OF SPECIFICATIONS ******************************************  00051715
UDXSTUB START 0                                                        00051862
UDXSNUM EQU   49                                                       00051915
UDXSTUB CSECT                                                          00052062
MAINENT DS    0H                                                       00052121
        USING *,R15                                                    00053019
        LAE   R15,0(R15,0)                                             00054020
        L     R15,=A(CICSTEST)                                         00055015
        BAKR  0,R15              PR from CICSTEST will restore GPRs    00056116
        LTR   R15,R15                                                  00057015
        BC    2,NOCICS                                                 00058058
*                                                                      00059015
YESCICS DS    0H                                                       00059139
        SAC   0                                                        00060039
        STM   R14,R12,12(R13)                                          00070033
        LR    R12,R15                                                  00071063
        DROP  R15                                                      00090038
        USING MAINENT,R12                                              00100019
        LR    R3,R0                                                    00110015
        B     NORMAL                                                   00120015
*                                                                      00121027
NOCICS  DS    0H                                                       00121139
        USING MAINENT,R12                                              00122039
        BSM   R14,0                                                    00130039
        BAKR  R14,0                                                    00131015
        LAE   R12,0                                                    00133063
        LR    R12,R15                                                  00134060
        SLR   R13,R13                                                  00135015
*****************************************************************      00136015
* At this point, R0 must contain the service number.                   00137015
*                If we are to call the TRUE, R13 is non-zero           00138015
*                R1 points to the caller's parameter list.             00139015
*****************************************************************      00140015
NORMAL   DS    0H                                                      00140115
         LA    R0,UDXSNUM           R0 gets service number             00140415
         SLR   R10_ZERO,R10_ZERO                                       00140515
         LR    RC,R10_ZERO                                             00140615
         L     R2,CVTPTR                                               00140715
         USING CVT,R2                                                  00140859
         L     R2,CVTABEND                                             00140959
         CLR   R2,R10_ZERO                                             00141058
         BC    8,NOICSF                                                00141158
         USING SCVTSECT,R2                                             00141259
```

```
         L     R2,SCVTCCVT                                              00141359
         CLR   R2,R10_ZERO                                              00141458
         BC    8,NOICSF                                                 00141558
         USING CCVT,R2                                                  00141615
         TM    CCVTSFG1,B'00110000' IS ICSF ACTIVE                      00141852
         BC    1,YESICSF                                                00142058
NOICSF   LA    RC,12               Set return code to 12 decimal        00143415
         L     R7,RETURN_CODE_PTR(,R1)                                  00143515
         ST    RC,RETURN_CODE(,R7)                                      00143615
         SLR   R0,R0                                                    00143715
         L     R7,REASON_CODE_PTR(,R1)                                  00143815
         ST    R0,REASON_CODE(,R7)                                      00143915
         B     FINISHED                                                 00144015
YESICSF  DS    0H                                                       00144115
*********************************************************************** 00144215
* Note that, if we're in CICS, the prolog code pointed R3 at the AFCB   00144315
* and R13 at the caller's savearea--they're still pointing.  Also, R0   00144415
* contains the service number, with the high order bit ON if the TRUE   00144515
* has been tried and found wanting. In this last case, CSFAPRPD will    00144615
* check the high order bit and not attempt to call the TRUE.            00144715
* If R13 is zero, we're using the linkage stack. That means we can      00144815
* call CSFAPRPC.                                                        00144915
* If R13 is not zero, we're using non-stack linkage. That means the     00145015
* caller's savearea will be used. CSFAPRPD uses this kind of linkage.   00145115
* But note that CSFAPRPD won't return here. Instead, it will return     00145215
* directly to the caller--that is, to the owner of the only save        00145315
* area around.                                                          00145415
*********************************************************************** 00145515
         CLR   R13,R10_ZERO                                             00145715
         BC    8,EXECPRPC                                               00145858
         L     R15,CCVTPRPD                                             00145915
         BALR  R14,R15                                                  00146015
         LR    RC,R15                                                   00146115
         B     FINISHED                                                 00146215
EXECPRPC L     R15,CCVTPRPC                                             00146315
         BALR  R14,R15                                                  00146415
         LR    RC,R15                                                   00146515
FINISHED DS    0H                                                       00146615
*                                                                       00146715
*************************************************************           00146815
* This routine uses the linkage stack to save the caller's regs         00146915
* if this is not a CICS environment.  In CICS, it uses the save         00147015
* area pointed to by register 13.  So the epilog code takes one         00147115
* of two forms.  If this is CICS (i.e. if R13 is non-zero),             00147263
* return is via LM and BR 14.  If this is not CICS, return is           00147363
* via PR.                                                               00147463
*                                                                       00147563
* On return, the PR of ESA linkage does not restore registers           00147663
* 0, 1, 14 and 15.  In the LM of normal BR 14 linkage, however,         00147763
* everything but 13 gets restored.  Since this routine has no           00147863
* autodata, there's no way to pass back return and reason codes         00147963
* unless we leave 0 and 15 intact.  The solution is to deviate          00148063
* slightly from normal BR 14 linkage and restore only registers         00148163
* 1 through 12 and 14.                                                  00148263
*************************************************************           00148363
          LTR   R13,R13                                                 00148460
          BC    8,ENDNOCICS                                             00148560
ENDCICS   LR    R15,RC                                                  00148660
          L     R14,SAVE14(,R13)                                        00148860
          LM    R1,R12,24(R13)                                          00148963
          BR    R14                                                     00149060
*                                                                       00149160
ENDNOCICS DS    0H                                                      00149355
          LR    R15,RC                                                  00149455
          PR                                                            00149757
********************************************************************** 00149855
********************************************************************** 00149960
**  CICSTEST:  Decides whether this is a CICS environment               00150060
********************************************************************** 00150160
********************************************************************** 00150260
CICSTEST DS   0H                                                        00150360
        LAE   R12,0              Clear AR 12                            00150560
```

```
         LR    R12,R15             Addressability via R12               00150660
         USING CICSTEST,R12                                             00150760
         L     R15,=A(UDXSTUB)      R15 gets caller's base reg          00151062
         L     R2,CVTPTR            GET CVT POINTER                     00160015
         USING CVT,R2                                                   00170015
         L     R2,CVTABEND          AND SECONDARY CVT POINTER           00180051
         USING SCVTSECT,R2                                              00190051
         L     R2,SCVTCCVT          POINT TO CSF CCVT                   00200051
         LTR   R2,R2                IS CRYPTO INSTALLED?                00210051
         BZ    RETRN                IF NOT, GO HOME                     00220015
         USING CCVT,R2                                                  00230051
         TM    CCVTSFG1,B'00110000' IS ICSF ACTIVE                      00240015
         BNO   RETRN                IF NOT , GO HOME                    00250015
* Check for wait list routine                                           00260015
         TM    CCVTCICS,B'10000000' Q. CCVTPRPA ON?                     00270015
         BZ    RETRN                      no---No CICS capability       00280015
         TM    CCVTCICS,B'01000000' Q. CCVTCKWL ON?                     00290015
         BZ    CKWLHERE                   no---use imbedded routine     00300015
*                                         yes--use installed routine    00310015
         LA    R0,UDXSNUM           R0 gets service number              00320015
         LR    R3,R1                R3 saves R1                         00330015
         LR    R4,R14               R4 saves R14                        00340048
         LR    R5,R15               R5 saves R15                        00350048
         L     R15,CCVTCKWL         R15 gets routine address            00360015
         BALR  R14,R15              Got check for CICS                  00370015
         LR    R0,R15               Save return code in R0              00380063
         LR    R15,R5               Restore R15                         00390048
         LR    R14,R4               Restore R14                         00400048
         LR    R1,R3                Restore R1                          00410015
         LTR   R0,R0                Q. CICS?                            00430015
         BZ    RETRN                  no---return                       00440015
*                                     yes--pass info along              00450015
         O     R15,M_CICS           Enable high bit of R15 to CICS      00460015
         B     RETRN                Return                              00470015
* Cannot use installed routine.  Use imbedded routine                   00480015
CKWLHERE DS    0H                    Imbedded check for TRUE routine     00490015
         SLR   R0,R0                Init R0 to 0                        00500015
         CPYA  R8,R12               Zero AR 8                           00510015
         SLR   R8,R8                Init R8 to 0                        00520063
         USING PSA,R8                                                   00530015
         L     R8,PSATOLD           R8->TCB                             00540015
         USING TCB,R8                                                   00550015
         LTR   R8,R8                Q.  Is there a TCB?                 00560015
         BC    8,RETRN                no---return                       00570061
*                                     yes--check state and key          00580015
         CPYA  R11,R12              Zero AR 11                          00590015
         LA    R11,1                Get PSW state and key in R6         00600015
         ESTA  R6,R11                                                   00610015
         LR    R7,R6                Copy of state & key in R7           00620015
         N     R7,M_KEY             Q.  problem key?                    00630015
         BZ    RETRN                  no---return                       00640015
*                                     yes--check state                  00650015
         N     R6,M_STATE           Q.  problem state?                  00660015
         BZ    RETRN                  no---return                       00670015
*                                     yes--get the CICS eye-catcher     00680015
         LA    R6,2                 Set ARs 6 and 8 to home             00690015
         SAR   R6,R6                                                    00700015
         SAR   R8,R6                                                    00710015
         L     R8,TCBEXT2           R8->TCB extension                   00720015
         USING TCBXTNT2,R8                                              00730015
         ICM   R4,B'1111',TCBCAUF   R4 gets AFCX address                00740015
*                                  Q.  Address there?                  00750015
         BZ    RETRN                      no---return                   00760015
*                                         yes--check eye-catch          00770015
         CLC   0(4,R4),CICS_EYE     Q.  CICS?                           00780015
         BNE   RETRN                  no---return                       00790015
*                                     yes--pass info along              00800015
         LR    R0,R4                R0 gets the AFCX pointer            00810015
         O     R15,M_CICS           Enable high order bit of R15        00820015
RETRN    DS    0H                                                       00830015
         DROP  R12                    Free R12                          00840015
```

```
        PR                      Return from CICSTEST subroutine      00850015
*                                                                    00860015
         LTORG                                                       00870015
         DS    0D                                                    00880015
*                                                                    00890015
UDXDATA  DS    0F                                                    00900015
R10_ZERO EQU   10                                                   00910015
RC       EQU   05                                                   00920015
R0       EQU   0                                                    00930015
R1       EQU   1                                                    00940015
R2       EQU   2                                                    00950015
R3       EQU   3                                                    00960015
R4       EQU   4                                                    00970015
R5       EQU   5                                                    00980015
R6       EQU   6                                                    00990015
R7       EQU   7                                                    01000015
R8       EQU   8                                                    01010015
R9       EQU   9                                                    01020015
R10      EQU   10                                                   01030015
R11      EQU   11                                                   01040015
R12      EQU   12                                                   01050015
R13      EQU   13                                                   01060015
R14      EQU   14                                                   01070015
R15      EQU   15                                                   01080015
*                                                                    01090015
INPUT_PARMS     EQU 0,8,C'C'                                        01100063
RETURN_CODE_PTR EQU INPUT_PARMS,4,C'A'                              01110015
REASON_CODE_PTR EQU INPUT_PARMS+4,4,C'A'                            01120015
RETURN_CODE     EQU 0,4,C'F'                                        01130063
REASON_CODE     EQU 0,4,C'F'                                        01140063
*                                                                    01141063
SAVAREA  EQU   0,72,C'C'                                            01150015
SAVE14   EQU   SAVAREA+12,4,C'A'                                    01160015
SAVE01   EQU   SAVAREA+24,4,C'A'                                    01170015
SCVTSPTR EQU   CVTABEND,4,C'F'                                      01190015
TCBPTR   EQU   PSATOLD,4,C'F'                                       01200015
         DS    0D                                                    01210015
*                                                                    01220015
         DS    0F                 Align                            01230015
M_KEY    DC    X'00800000'        Problem key mask                 01240015
M_STATE  DC    X'00010000'        Problem state mask               01250015
M_NOCICS DC    X'7FFFFFFF'        Not-CICS mask                    01260015
M_CICS   DC    X'80000000'        Yes-CICS mask                    01270015
         DS    0D                                                    01280015
CICS_EYE DC    CL4'AFCX'          CICS eye catcher                 01290015
*                                                                    01300015
         IHAPSA                                                      01310015
         TITLE 'DSECT CVT'                                           01320015
         CVT DSECT=YES                                               01330015
         TITLE 'DSECT SCVT'                                          01340015
         IHASCVT DSECT=YES                                           01350015
         TITLE 'DSECT TCB'                                           01360015
         IKJTCB                                                      01370015
         TITLE 'DSECT CCVT'                                          01380015
         CSFCCVT                                                     01390015
*                                                                    01400015
         END                                                         01410015
```

Appendix C. UDX Sample Code - Host Piece - CSFPCI Post-Processing Exit

This appendix contains a listing of the sample file *zudxexit.bal*. This file is a skeleton for the design of the host piece of a CCA extension.

```
**** START OF SPECIFICATIONS ****************************************  00001020
*                                                                     *  00002020
* MODULE NAME = ZUDXEXIT                                              *  00003020
* DESCRIPTIVE NAME = Sample Post-Processing exit for CSFPCI           *  00004020
*                                                                     *  00005020
*                                                                     *  00006020
**** END OF SPECIFICATIONS ******************************************  00007020
ZUDXEXIT CSECT ,                                                        00008020
ZUDXEXIT AMODE 31                                                       00009020
ZUDXEXIT RMODE ANY                                                      00010020
MAINENT  DS    0H                                                       00020020
         USING *,R15                                                    00030020
         B     PROLOG                                                   00040020
         DC    AL1(42)                                                  00050020
         DC    C'ZUDXEXIT - UDX CSFPCI POST-PROCESSING EXIT'            00060020
         DROP  R15                                                      00070020
PROLOG   BSM   R14,0                                                    00080020
         BAKR  R14,0                                                    00090020
         LAE   R12,0                                                    00100020
         LR    R12,R15                                                  00110020
PSTART   EQU   ZUDXEXIT                                                 00120020
         USING PSTART,R12                                               00130020
         L     R00,DYNDATA_SIZE                                         00140029
         LA    R15,0                                                    00150020
         CPYA  AR01,AR12                                                00160020
         SAC   512                                                      00170020
         CSFAGET  OBTAIN,LENGTH=(0),SP=(15),LINKAGE=SYSTEM              00180020
         LAE   R11,0(,R01)                                              00190020
         USING DATAD,R11                                                00200020
*-------------------------------------------------------------------    00201020
* SAVE THE ADDRESS OF THE XPB                                           00202020
*-------------------------------------------------------------------    00203020
         EREG  R00,R01                                                  00204029
         LR    R03_EXPBPTR,R00                                          00204220
         USING XPB,R03                                                  00204320
         CPYA  R03_EXPBPTR,AR12                                         00204420
*-------------------------------------------------------------------    00204520
* IS THIS INVOKED AS PART OF PRE-PROCESSING OR POST-PROCESSING ?        00204629
*-------------------------------------------------------------------    00204720
         TM    XPBFLAG1,XPBEXIT_TERM_CALL                               00204820
         BNZ   POSTPROC                                                 00204920
PREPROC  DS    0H                          START PRE_PROCESSING         00205020
         LA    R14,1                                                    00206020
         SAR   AR14,R14                                                 00207020
         LR    R14,R01                                                  00208020
         L     R14,REPLY_PARM_DB_LEN_PTR(,R14)                          00209020
         L     R00,REPLY_PARM_DATA_BLOCK_LEN(,R14)  SAVE LENGTH         00210020
         ST    R00,XPBWORD                                              00220020
         B     RCISZERO                                                 00230020
*-------------------------------------------------------------------    00240020
* If the Rule Array indicates to Query Access Control Points, then      00250020
* copy the UDX access control points to the Reply Parameter Data Block  00260020
*-------------------------------------------------------------------    00270020
POSTPROC DS    0H                                                       00280020
         LA    R06,1                                                    00290020
         SAR   AR06,R06                                                 00300020
         LR    R06,R01                     GET PARAMETER LIST ADDR      00310020
         L     R06,RA_PTR(,R06)            GET RULE ARRAY ADDR          00320020
         MVC   LOC_RA(8),RULE_ARRAY(R06)   COPY RULE ARRAY              00330020
         OC    LOC_RA(8),RA_BLANK          FOLD IT TO UPPER CASE        00340020
         CLC   LOC_RA(8),ACPOINTS          IS IT "ACPOINTS" CALL ?      00350020
```

```
         BNE   NOT_ACP                    CALL WAS NOT FOR ACPOINTS    00360020
*                                                                      00370020
GET_ACP  DS    0H                         START OF COPY ACPOINTS CODE  00380020
         LA    R14,1                                                   00390020
         SAR   AR14,R14                                                00400020
         LR    R14,R01                    GET PARAMETER LIST ADDR      00410020
         L     R14,REPLY_PARM_DB_LEN_PTR(,R14)                         00420020
         L     R08,REPLY_PARM_DATA_BLOCK_LEN(,R14)                     00430020
         LR    R09,R08                    SAVE CURRENT DATA BLOCK LEN  00440020
         LA    R00,ACPTLEN                GET ACPT LENGTH              00450020
         ALR   R08,R00                    FIND TOTAL ACP LENGTH        00460021
         CL    R08,XPBWORD                IS THERE ENOUGH ROOM ?       00470021
         BNH   BUFLENOK                   YES,  BUFFER LENGTH IS OK    00480020
*--------------------------------------------------------------------   00490020
* THERE IS NOT ENOUGH ROOM IN THE BUFFER TO HOLD THE ADDITIONAL         00500020
* UDX ACCESS CONTROL POINTS, SO WE WILL FAIL.                           00510020
*--------------------------------------------------------------------   00520020
BUFLENER DS    0H                         NO, BUFFER LENGTH ERROR      00530020
         LA    R04_LOCAL_RC,8             APPLICATION ERROR            00540020
         LA    R05_LOCAL_RS,605           BUFFER LENGTH ERROR          00550020
         OI    XPB_USERCRS,B'00100100'                                 00560020
         B     FINISHED                                                00570020
*--------------------------------------------------------------------   00580020
* THERE IS ENOUGH ROOM IN THE BUFFER TO HOLD THE ADDITIONAL UDX         00590020
* ACCESS CONTROL POINTS, SO COPY THEM TO THE REPLY DATA BLOCK           00600020
*--------------------------------------------------------------------   00610020
BUFLENOK DS    0H                                                      00620020
         LR    R06,R01                                                 00670020
         L     R06,REPLY_PARM_DB_PTR(,R06)                             00671020
         ALR   R06,R09                                                 00680020
         LA    R07,ACPTLEN                GET ACPT LENGTH              00681027
         LA    R04,0                                                   00682025
         SAR   AR04,R04                                                00683025
         LA    R04,UACPTS                                              00690025
         LA    R05,ACPTLEN                GET ACPT LENGTH              00691027
         MVCL  R06,R04                                                 00700025
         ST    R08,REPLY_PARM_DATA_BLOCK_LEN(,R14)                     00710020
NOT_ACP  DS    0H                                                      00720020
RCISZERO DS    0H                                                      00730020
         LA    R04_LOCAL_RC,0                                          00740021
         LA    R05_LOCAL_RS,0                                          00750021
*                                                                      00760020
FINISHED DS    0H                                                      00770020
         L     R00,DYNDATA_SIZE                                        00780029
         LA    R15,0                                                   00790020
         LR    R01,R11                                                 00800020
         CSFAFREE RELEASE,LENGTH=(0),ADDR=(1),SP=(15),LINKAGE=SYSTEM   00810020
         EREG  R01,R01                                                 00820020
         LR    R00,R05_LOCAL_RS                                        00830020
         LR    R15,R04_LOCAL_RC                                        00840020
         PR                                                            00850020
********************************************************************   00860020
* Storage Declares                                                     00870020
********************************************************************   00880020
DYNDATA_SIZE DS    0A                                                  00890020
             DC    A(DYNSIZE)                                          00920020
             DS    0D                                                  00930020
*--------------------------------------------------------------------   00940020
* Register Equates                                                     00950020
*--------------------------------------------------------------------   00960020
R00      EQU   0                                                       00970020
R01      EQU   1                                                       00980020
R02      EQU   2                                                       00990020
R03      EQU   3                                                       01000020
R04      EQU   4                                                       01010020
R05      EQU   5                                                       01020020
R06      EQU   6                                                       01030020
R07      EQU   7                                                       01040020
R08      EQU   8                                                       01050020
R09      EQU   9                                                       01060020
R10      EQU   10                                                      01070020
R11      EQU   11                                                      01080020
```

```
R12      EQU   12                                                      01090020
R13      EQU   13                                                      01100020
R14      EQU   14                                                      01110020
R15      EQU   15                                                      01120020
*-------------------------------------------------------------------   01130020
* Access Register Equates                                             01140020
*-------------------------------------------------------------------   01150020
AR00     EQU   0                                                       01160020
AR01     EQU   1                                                       01170020
AR02     EQU   2                                                       01180020
AR03     EQU   3                                                       01190020
AR04     EQU   4                                                       01200020
AR05     EQU   5                                                       01210020
AR06     EQU   6                                                       01220020
AR07     EQU   7                                                       01230020
AR08     EQU   8                                                       01240020
AR09     EQU   9                                                       01250020
AR10     EQU   10                                                      01260020
AR11     EQU   11                                                      01270020
AR12     EQU   12                                                      01280020
AR13     EQU   13                                                      01290020
AR14     EQU   14                                                      01300020
AR15     EQU   15                                                      01310020
*-------------------------------------------------------------------   01311029
* Parameter block Equates                                             01312029
*-------------------------------------------------------------------   01313029
PARMBLOCK             EQU 0,64,C'C'                                   01314030
RA_COUNT_PTR          EQU PARMBLOCK+16,4,C'A'                         01315030
RA_PTR                EQU PARMBLOCK+20,4,C'A'                         01316030
REPLY_PARM_DB_LEN_PTR EQU PARMBLOCK+56,4,C'A'                         01316130
REPLY_PARM_DB_PTR     EQU PARMBLOCK+60,4,C'A'                         01316230
*                                                                     01317029
RULE_ARRAY_COUNT          EQU 0,4,C'F'                                01318030
RULE_ARRAY                EQU 0,8,C'C'                                01319030
REPLY_PARM_DATA_BLOCK_LEN EQU 0,4,C'F'                                01319429
REPLY_PARM_DATA_BLOCK     EQU 0,,C'C'                                 01319530
END_REPLY_PARM_DATA_BLOCK EQU 0,,C'C'                                 01319629
*-------------------------------------------------------------------   01320020
* Variable Equates                                                    01330020
*-------------------------------------------------------------------   01340020
R03_EXPBPTR  EQU R03                                                  01350020
R04_LOCAL_RC EQU R04                                                  01360020
R05_LOCAL_RS EQU R05                                                  01370020
*-------------------------------------------------------------------   01380020
* Variable Data Areas                                                 01390020
*-------------------------------------------------------------------   01400020
         LTORG                                                        01401028
*-------------------------------------------------------------------   01410020
* Constants                                                           01420020
*-------------------------------------------------------------------   01430020
ACPOINTS DC    CL8'ACPOINTS'                                          01440020
RA_BLANK DC    CL8'        '                                          01450020
*                                                                     01451023
*====================================================================  01452028
*====================================================================  01453028
*                                                                     01454028
* START OF THE UDX ACCESS CONTROL POINT TABLE STRUCTURE (UACPTS)      01455028
*                                                                     01456028
* The UACPTS has Group Information followed by one or more entries    01457028
* of Access Control Point Information.  The Group Information should  01458028
* NOT be changed.  The Access Control Point Information (ACPT) can be 01459028
* copied and most of it modified to describe your UDX Access Control  01460028
* Point.                                                              01470028
*                                                                     01471028
* Descriptions of the 2 types of entries are as follows:              01472028
*                                                                     01473028
* ACP Group Information (This information must not be changed) :      01474028
*    TYPE    - Type of table entry,  01 is grouping information entry 01475028
*    TXT_LEN - Length of TXT field                                    01476028
*    TXT     - ASCII description of the Group                         01477028
*                                                                     01478028
* ACPT Information:                                                   01479028
```

```
*    TYPE      - Type of table entry, 02 is an access control point     01479128
*                entry.  THIS VALUE MUST ALWAYS BE X'02'.               01479228
*    CODE      - The hexidecimal value of the Access Control Point      01479328
*    TXT_LEN   - Length of TXT field                                    01479428
*    TXT       - ASCII description of this Access Control Point         01479528
*    FLAG      - THIS VALUE MUST ALWAYS BE X'00000000'.                 01479629
*    ACPTS_CNT - The number of other ACPTs listed in ACPTS              01479828
*    ACPTS     - The hexidecimal value of the other ACPTs that must be  01479928
*                enabled if this ACPT is enabled.                       01480028
*                                                                       01480128
* To add a UDX ACPT:                                                    01480228
* 1.  Increase the size of the  'UACPTS' to include the new ACPT        01480328
*     Information.   This increase will vary based on the length of     01480428
*     the descriptive text and the number of ACPTs needing to be        01480528
*     enabled for this ACPT.                                            01480628
* 2.  Copy the lines of code between the comments:                      01480728
*     *** COPY STARTING HERE                                            01480828
*     *** ENDING HERE                                                   01480928
*     And put them before the comment:                                  01481028
*     *** PLACE BEFORE THIS LINE                                        01481128
* 3.  Change:                                                           01481228
*     a. The names of the lines, for exapmle A01_... would become       01481328
*        A03_... for the third Access Control Point.                    01481428
*     b. The length of the ACP section, 'A03      DS   CL??'. where     01481528
*        ?? is the actual length of the ACPT information being added.   01481628
*        This value may be different for each ACPT depending on the     01481728
*        length of the text and the number of other ACPTs needing to    01481828
*        be enabled.                                                    01481928
*     c. '_CODE' to the 2 byte hexidecimal ACPT value assigned to the   01482028
*        UDX.                                                           01482128
*     d. '_TXT_LEN' to the length of the descriptive text.              01482228
*     e. '_TXT' to the ASCII repersentation of the descriptive text     01482328
*        for this UDX.                                                  01482428
*     f. '_ACPTS_CNT' to the number of ACPTS to be listed in the        01482629
*        following field.                                               01482728
*     g. '_ACPTS' all the 2 byte hexidecimal value of the ACPTs which   01482829
*        need to be active for this UDX to function.                    01482928
* 4.  Add the OFFSET of this UDX ACPT Information block with the        01483028
*     length of this UDX ACPT Information Block and change the          01483128
*     offset to the next UDX ACPT Information Block.  NOTE: This        01483228
*     new offset should be the same as the length of the 'UACPTS'       01483328
*     from step 1.                                                      01483428
*                                                                       01483528
*=====================================================================  01483628
*=====================================================================  01483728
ACPTSTRT DS    0X                                                      01483828
UACPTS   DS    CL77                WILL CHANGE TO NEW LENGTH           01483928
         ORG   UACPTS                                                  01484028
*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=  01484128
* ACP GROUP INFORMATION - SHOULD NEVER BE CHANGED                      01484228
*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=  01484328
GRP         DS    CL9               LENGTH OF GROUP INFORMATION       01484428
            ORG   GRP                                                   01484528
GRP_TYPE    DC    X'01'             GROUP TYPE OF 01                  01484628
GRP_TXT_LEN DC    X'00000004'       TEXT LENGTH                       01484728
GRP_TXT     DC    X'55445873'       TEXT IS ASCII 'UDXs'              01484828
*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=  01484928
* Access Control Point (ACPT) Information block for the 1st ACPT       01485028
*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=  01485128
         ORG  UACPTS+9             A01 STARTING OFFSET                 01485228
A01      DS   CL45                A01 LENGTH                          01485328
         ORG  A01                  A01 BREAKDOWN                       01485428
A01_TYPE      DC X'02'             A01 TYPE -->MUST REMAIN X'02'       01485528
A01_CODE      DC X'8001'           A01 ACCESS CONTROL POINT            01485628
A01_TXT_LEN   DC X'0000001E'       A01 TEXT LENGTH                     01485728
A01_TXT       DC X'4649525354205544582041434345535320434F4E54524F4C2050*01485828
               4F494E54'           A01 ASCII DESCRIPTION OF ACP        01485928
A01_FLAG      DC X'00000000'       A01 FLAGS                           01486028
A01_ACPTS_CNT DC X'00000000'       A01 ACPS ENABLE COUNT - NONE        01486128
A01_ACPTS     DS 0F                A01 ACPS NEEDING ENABLED - NONE     01486228
*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=  01486328
```

```
* Access Control Point (ACPT) Information block for the 2nd ACPT        01486428
*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-   01486528
         ORG  UACPTS+54             A02 STARTING OFFSET                 01486628
*** COPY STARTING HERE                                                  01486728
A02      DS   CL23                 A02 LENGTH                          01486828
         ORG  A02                   A02 BREAKDOWN                       01486928
A02_TYPE      DC X'02'             A02 TYPE                            01487028
A02_CODE      DC X'8FFF'           A02 ACCESS CONTROL POINT            01487129
A02_TXT_LEN   DC X'00000006'       A02 TEXT LENGTH                     01487228
A02_TXT       DC X'414350542032'   A02 ASCII DESCRIPTION OF ACP        01487328
A02_FLAG      DC X'00000000'       A02 FLAGS                           01487428
A02_ACPTS_CNT DC X'00000001'       A02 ACPS ENABLE COUNT - ONE         01487528
A02_ACPTS     DC X'8001'           A02 ACPS NEEDING ENABLED            01487628
*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-   01487728
* Access Control Point (ACPT) Information block for the NEXT ACPT       01487828
*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-   01487928
         ORG   UACPTS+77                     NEXT UDX ACPT OFFSET      01488028
*** ENDING HERE                                                         01488128
*** PLACE BEFORE THIS LINE                                             01488228
*                                                                       01488328
ACPTEND  DS    0X                                                       01488428
ACPTLEN  EQU   (ACPTEND-ACPTSTRT)                                       01488528
*======================================================================01488628
*======================================================================01488728
*                                                                       01488828
* END OF THE ACP STRUCTURE                                              01488928
*                                                                       01489028
*======================================================================01489128
*======================================================================01489228
*                                                                       01489328
************************************************************************01489420
* Dynamic Data Definitions                                             01490020
************************************************************************01500020
DATAD    DSECT 0F                                                       01510020
         DS    0D                                                       01520020
*----------------------------------------------------------------------01680020
LOC_RA   DS    CL8                                                      01701027
*                                                                       02910020
         ORG   *+1-(*-DATAD)/(*-DATAD)                                  02920020
ENDDATAD DS    0X                                                       02930020
*                                                                       02940020
DYNSIZE  EQU   ((ENDDATAD-DATAD+7)/8)*8                                02950020
*                                                                       02960020
* THE FOLLOWING INSTRUCTION WILL CAUSE AN ASSEMBLY ERROR IF THE         02970020
* SIZE OF THE AUTOMATIC STORAGE AREA IS GREATER THAN 4096 BYTES.        02980020
         ORG   DATAD+(12288-(*-DATAD))                                  02990020
         ORG   ENDDATAD                                                 03000020
*                                                                       03020320
         CSFASPB                                                        03021020
         CSFEXPB                                                        03030020
         CSFCCVT                                                        03040020
         CSFCCVE                                                        03050020
*                                                                       03060020
         END ZUDXEXIT                                                   03070020
```

Appendix D. UDX Sample Code - Coprocessor Piece

This appendix contains a listing of the sample file *zudxsamp.c*.

```
/*****************************************************************************/
/*                                                                           */
/* Module Name: ZUDXSAMP.C                                                  */
/*                                                                           */
/* Descriptive Name: User Defined Extension zudxPIN1                         */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* (C) Copyright IBM Corporation 1999                                        */
/*---------------------------------------------------------------------------*/
/*                                                                           */
/* Version 001, Release 000, Level 000                                       */
/*                                                                           */
/* Author: Kenneth B Kerr                                                    */
/*                                                                           */
/* Function:                                                                 */
/*    This module is the command processor for the user defined              */
/*    extension (verb) zudxPIN1.                                             */
/*                                                                           */
/* Module Type:                                                              */
/*   Attributes:  Serial usable                                              */
/*   Language: IBM Visual Age C++ Version 3.00                               */
/*                                                                           */
/* Entry Points:                                                             */
/*   zudxPIN1                                                                */
/*                                                                           */
/* Input:                                                                    */
/*    pCprbIn               Request CPRB                                     */
/*    RequestId             Identifier number of the request                 */
/*                                                                           */
/* Output:                                                                   */
/*    pCprbOut              Reply CPRB                                       */
/*                                                                           */
/*---------------------------------------------------------------------------*/
/* Change history:                                                           */
/* Date      Programmer  Description                                         */
/* --------  ----------  -----------                                         */
/* 08/26/99  kbk         Created                                             */
/*                                                                           */
/*                                                                           */
/*****************************************************************************/

/*****************************************************************************/
/* Include files.                                                            */
/*****************************************************************************/
#include <stdlib.h>
#include <string.h>

#include "cmncryt2.h"                   /* Cryptographic T2 definitions.    */
#include "cmnerrcd.h"                   /* Common error codes.              */
#include "cam_xtrn.h"                   /* CCA managers                     */

#include "casfunct.h"                   /* Common command processor funct's */
#include "camacm.h"                     /* Needed for access check          */
#include "cxt_cmds.h"                   /* UDX access control codes         */
#include "camdmgr.h"                    /* Domain manager prototypes        */
#include "cassub.h"                     /* Common subroutines               */

/*****************************************************************************/
/* Constants                                                                 */
/*****************************************************************************/
#define RA_COUNT_ZERO 2                /* Rule array length for zero keywords */
#define RA_COUNT_ONE 10                /* Rule array length for one keyword   */
#define RA_COUNT_TWO 18                /* Rule array length for two keywords  */
#define PIN_BLOCK_SIZE 8
#define EXTRA_DATA_SIZE 8
#define EXPECTED_VUD_LENGTH 18
```

```
#define SIZE_OF_DES_KEY 8
#define PINENCI                   0x21 /* PIN-Encrypting IN Key:  IPINENC     */
#define PINENCO                   0x24 /* PIN-Encrypting OUT Key: OPINENC     */


typedef enum { PIN1_KEYWORD_1W, PIN1_KEYWORD_2W, PIN1_KEYWORD_3W } PIN1_RULE1;
typedef enum { PIN1_KEYWORD_1O, PIN1_KEYWORD_2O } PIN1_RULE2;

/*****************************************************************
  ** ENTER
  **       your CCA command extension array entry after this comment.
  **       ==========================================================
  **
  ** Each element of the table is a CCAX_CP_DEF type.  That is, it
  ** contains one 2 character sub-function code, and a pointer to
  ** the corresponding command processor function.
  **
  *********************************/

  CCAX_CP_DEF ccax_cp_listff“ = { { ZUDX_CODE, zudxPIN1 }};


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

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



void zudxPIN1(
         CPRB_structure *pCprbIn,       /* (input) request CPRB               */
         CPRB_structure *pCprbOut,      /* (output) reply CPRB                */
         unsigned long   RequestId,     /* (input) Adapter request identifier */
         role_id_t       roleID )       /* (input) role ID ptr                */
{
  /*****************************************************************************/
  /* Declarations                                                              */
  /*****************************************************************************/
  /* CPRB processing variables */
  ESSS_request_block_structure *pReqBlk;  /* Pointer to request parm block    */
  ESSS_request_block_structure *pRepBlk;  /* Pointer to reply parm block      */
  UCHAR              *pReplyBlockPtr;     /* Pointer for building reply block */
  int                 ReplyBlockLength;   /* Length of the data added to the  */
                                          /* reply parameter block            */

  /* VUD block processing variables */
  verb_unique_data_structure *pVUDBlock;  /* VUD structure                    */
  unsigned char              *pPinBlock;  /* Pointer to input PIN block       */
  unsigned char              *pExtraData; /* Pointer to extra data block      */

  /* Key block processing variables */
  key_data_structure          *pThiskey;  /* Key token from request block     */
  key_data_structure          *pNextkey;  /* Key token from request block     */
  generic_key_block_structure *pToken;    /* Key in parameter block           */
  des_key_token_structure *pInputPinKeyToken;  /* Input PIN key token         */
  des_key_token_structure *pOutputPinKeyToken;  /* Output PIN key token       */
  KEY_FIELD_HEADER       InputKeyHeader;  /* header for key in reply block    */
  KEY_FIELD_HEADER      OutputKeyHeader;  /* header for key in reply block    */

  /* local variables */
  long           ReturnMsg;            /* Return code from function calls     */
  mk_status_var  MstrKeyStatus;        /* Master key status                   */
  mk_selectors   MKSelector;           /* Master key selector                 */
  boolean        Authorized;           /* Truth value that the caller is      */
                                       /* authorized to execute this command  */
  unsigned char  OutputPinBlockff 8 “;  /* Output PIN block                   */
  int            i;                    /* Iteration variable                  */
  DES_TOKEN_CHECK ErrorMessage;        /* error message from DES token check  */
```

```
  UCHAR MKVPfill MKVP_LENGTH ";           /* master key verification pattern     */
  ULONG TVV;                              /* calculated TVV for output key token */

  /* Rule array processing variables */
  int           RuleValuefill 2 ";        /* Output for rule_check               */
  USHORT        RuleMapCount = 5;         /* Number of entries in the rule map   */
  static RULE_MAP RuleMapfill 5 " = { { "KEYW1   ", 1, PIN1_KEYWORD_1W },
                                      { "KEYW2   ", 1, PIN1_KEYWORD_2W },
                                      { "KEYW3   ", 1, PIN1_KEYWORD_3W },
                                      { "KEYO1   ", 2, PIN1_KEYWORD_1O },
                                      { "KEYO2   ", 2, PIN1_KEYWORD_2O } } ;
  PIN1_RULE1  Rule1;                      /* Rule 1 specified                    */
  PIN1_RULE2  Rule2;                      /* Rule 2 specified                    */


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

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

  /*****************************************************************************/
  /* Initialize the CPRB request/reply parameter pointers and then             */
  /* set my local pointers to the request and reply parameter blocks.          */
  /*****************************************************************************/
  InitCprbParmPointers( pCprbIn, pCprbOut );
  pReqBlk = pCprbIn->req_parm_block;
  pRepBlk = pCprbOut->reply_parm_block;

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

  /*****************************************************************************/
  /* Check that the caller is authorized to use this domain.                   */
  /* Set the domain in the master key selector.                                */
  /*****************************************************************************/
  if ( ! dmDomainCheck( pCprbIn ))
  {
    Cas_proc_retc( pCprbOut, DOMAIN_MANAGER_ERROR );
    return;
  }

  /*****************************************************************************/
  /* Initialize the master key selector.                                       */
  /*****************************************************************************/
  MKSelector.mk_set = pCprbIn->Domain;
  MKSelector.type_mks = SYM_MK;

  /*****************************************************************************/
  /* Make sure this service is authorized before we go any further             */
  /*****************************************************************************/

  CHECK_ACCESS_AUTH( pCprbIn,
                     pCprbOut,
                     roleID,
                     UDX_COMMAND_PIN1,
                     &Authorized );
  if ( !Authorized )
  {
    Cas_proc_retc ( pCprbOut, CP_NOT_AUTH ) ;
    return ;
  }
/*************************************************************/
```

```
/*****************************************************************************/
/* Make sure the current master key is valid before we go any further.      */
/*****************************************************************************/
ReturnMsg = mkmGetMasterKeyStatus ( MKSelector, &MstrKeyStatus );
switch ( ReturnMsg)
{
  case MK_NO_ERROR :
    if (( MstrKeyStatus & mks_CMK_VALID ) != mks_CMK_VALID )
    {
      Cas_proc_retc(pCprbOut, MASTER_KEY_ERROR);
      return ;
    }
    break ;

  case MK_SRDI_OPEN_ERROR :
    Cas_proc_retc ( pCprbOut, FT_MK_SRDI_OPENERR ) ;
    return ;
    break ;

  default :
    Cas_proc_retc(pCprbOut, MASTER_KEY_ERROR);
    return ;
}

/*****************************************************************************/
/* Perform consistency check on the request parameter block                  */
/*****************************************************************************/
if ( parm_block_valid( pCprbIn, SEL_REQ_BLK ) == false )
{
  Cas_proc_retc ( pCprbOut, RT_CONSISTENCY_ERROR ) ;
  return ;
}

/*****************************************************************************/
/* Perform consistency check on the rule array - for this verb, the          */
/* rule array may have zero, one or two values.                              */
/*****************************************************************************/
switch( pReqBlk->rule_array_length )
{
  case RA_COUNT_ZERO: /* use default values */
    Rule1 = PIN1_KEYWORD_1W; /* default */
    Rule2 = PIN1_KEYWORD_2O; /* default */
    break;

  case RA_COUNT_ONE :
  case RA_COUNT_TWO :
    RuleValueffl 0 “ = INVALID_RULE;  /* rule_check requires this initialization.*/
    RuleValueffl 1 “ = INVALID_RULE;

    if ( rule_check ( (RULE_BLOCK *) &pReqBlk->rule_array_length,
                                      RuleMapCount,
                                     &RuleMapffl0“,
                           (int *) &RuleValue,
                                     &ReturnMsg ) == false )
    {
      Cas_proc_retc ( pCprbOut, ReturnMsg ) ;
      return ;
    }

    if ( RuleValueffl 0 “ == INVALID_RULE )
      Rule1 = PIN1_KEYWORD_1W; /* default */
    else
      Rule1 = (PIN1_RULE1) RuleValueffl 0 “;

    if ( RuleValueffl 1 “ == INVALID_RULE )
      Rule2 = PIN1_KEYWORD_2O; /* default */
    else
      Rule2 = (PIN1_RULE2) RuleValueffl 1 “;
    break;

  default: /* count not valid */
    Cas_proc_retc( pCprbOut, E_RULE_ARRAY_CNT );
```

```
    return ;

}

/*****************************************************************************/
/* Perform consistency check on the verb unique data. Parse the PIN block   */
/* and extra data block from the VUD.                                        */
/*****************************************************************************/
pVUDBlock = (verb_unique_data_structure *)
                    ((UCHAR *)&pReqBlk->rule_array_length +
                     pReqBlk->rule_array_length );

if ( pVUDBlock->verb_unique_data_length != EXPECTED_VUD_LENGTH )
{
  Cas_proc_retc( pCprbOut, RT_CONSISTENCY_ERROR );
  return;
}

pPinBlock = (unsigned char *) &pVUDBlock->verb_unique_data;
pExtraData = (unsigned char *) pPinBlock + PIN_BLOCK_SIZE;

/*****************************************************************************/
/* Parse the PIN Encrypting Keys from the key block.                         */
/*****************************************************************************/
if ( ! find_first_key_block( pCprbIn, &pThiskey, SEL_REQ_BLK ))
{
  Cas_proc_retc( pCprbOut, RT_CONSISTENCY_ERROR );
  return;
}
pToken = (generic_key_block_structure * ) pThiskey;
pInputPinKeyToken = ( des_key_token_structure * ) ((UCHAR * ) &pToken->label_or_token);

/*****************************************************************************/
/*  Get the output PIN encrypting key token from the request block.          */
/*****************************************************************************/
if ( ! find_next_key_block( pCprbIn, pThiskey, &pNextkey, SEL_REQ_BLK ))
{
  Cas_proc_retc( pCprbOut, RT_CONSISTENCY_ERROR );
  return;
}

pToken = (generic_key_block_structure * ) pNextkey;
pOutputPinKeyToken = ( des_key_token_structure * ) ((UCHAR * ) &pToken->label_or_token);

/*****************************************************************************/
/* Check the input PIN Encrypting Key.                                       */
/*****************************************************************************/
if ( ! cas_des_key_token_check( pInputPinKeyToken, &ErrorMessage ))
  switch( ErrorMessage )
  {
    case DES_TOKEN_CHECK_VERSION :
      Cas_proc_retc( pCprbOut, E_INV_TKNVER );
      return;
      break;

    case DES_TOKEN_CHECK_TOKENFLAG :
      Cas_proc_retc( pCprbOut, E_KEK_ID_FORM );
      return;
      break;

    default :
      Cas_proc_retc( pCprbOut, RT_TKN_UNUSEABLE );
      return;

  } /* select on error message */

if ( cas_key_tokentvv_check( pInputPinKeyToken ) == false )
{
   Cas_proc_retc( pCprbOut, E_INTRN_TOKEN_TVV);
   return;
}
```

```
/*****************************************************************************/
/* Check the output PIN Encrypting Key if it's not a null token.             */
/*****************************************************************************/
if ( pOutputPinKeyToken->tokenFlag != EMPTY_TOKEN_FLAG )
{
  if ( ! cas_des_key_token_check( pOutputPinKeyToken, &ErrorMessage ))
    switch( ErrorMessage )
    {
      case DES_TOKEN_CHECK_VERSION :
        Cas_proc_retc( pCprbOut, E_INV_TKNVER );
        return;
        break;

      case DES_TOKEN_CHECK_TOKENFLAG :
        Cas_proc_retc( pCprbOut, E_KEK_ID_FORM );
        return;
        break;

      default :
        Cas_proc_retc( pCprbOut, RT_TKN_UNUSEABLE );
        return;

    } /* select on error message */

  if ( cas_key_tokentvv_check( pOutputPinKeyToken ) == false )
  {
    Cas_proc_retc( pCprbOut, E_INTRN_TOKEN_TVV);
    return;
  }

}

/*****************************************************************************/
/* Control vector checking.                                                  */
/*****************************************************************************/
/* Perform any necessary checking of the control vector.                     */
/* Add other checks as appropriate.                                          */

/* Check that the input PIN Key is of the input PIN encrypting class */
if ( pInputPinKeyToken->cvBaseff1“ != PINENCI )
{
  Cas_proc_retc( pCprbOut, RT_CV_CONFLICT );
  return;
}


/* If Rule2 is PIN1_KEYWORD_2O, check the output PIN Key */
if ( Rule2 == PIN1_KEYWORD_2O )
{

 if ( pOutputPinKeyToken->cvBaseff1“ != PINENCO )
  {
    Cas_proc_retc( pCprbOut, RT_CV_CONFLICT );
    return;
  }

}

/*****************************************************************************/
/* Determine which master key to use to decipher the input PIN key.          */
/*****************************************************************************/
switch( cas_master_key_check( pInputPinKeyToken ))
{
  case OLD :
    /*************************************************************************/
    /* The key token's MKVP matches the old master key's MKVP. Generate a   */
    /* warning reason code that a key is encrypted under the old master key.*/
    /*************************************************************************/
    MKSelector.mk_register = old_mk;
    Cas_proc_retc( pCprbOut, RT_OMK_TOKEN_USED );
    break;
```

```
  case CURRENT :
    /*************************************************************************/
    /* The key token's MKVP matches the current master key's MKVP.           */
    /*************************************************************************/
    MKSelector.mk_register = current_mk;
    break;

  case OUT_OF_DATE :
  default :
    /*************************************************************************/
    /* The key token's MKVP doesn't match current or old master key's        */
    /* MKVP.  We don't know what to do with the key token.                   */
    /*************************************************************************/
    Cas_proc_retc( pCprbOut, RT_KEY_INV_MKVN );
    return;
    break;
}

/*****************************************************************************/
/* Decipher the input PIN key under the corresponding master key.            */
/*****************************************************************************/
ReturnMsg = triple_decrypt_under_master_key_with_CV( &MKSelector,
                                        &(pInputPinKeyToken->cvBase),
                                        &(pInputPinKeyToken->keyLeftfff0“),
                                        &(pInputPinKeyToken->keyLeftfff0“) );

if ( ReturnMsg != mk_NO_ERROR )
{
  Cas_proc_retc( pCprbOut, ReturnMsg );
  return;
}

ReturnMsg = triple_decrypt_under_master_key_with_CV( &MKSelector,
                                        &(pInputPinKeyToken->cvExten),
                                        &(pInputPinKeyToken->keyRightfff0“),
                                        &(pInputPinKeyToken->keyRightfff0“) );

if ( ReturnMsg != mk_NO_ERROR )
{
  Cas_proc_retc( pCprbOut, ReturnMsg );
  return;
}

/*****************************************************************************/
/* Generate warning return code if the PIN key does not have odd parity.     */
/*****************************************************************************/
for ( i = 0; i < SIZE_OF_DES_KEY; i++ )
{
  if ( (cas_parity_odd( pInputPinKeyToken->keyLeftfffi“ ) == FALSE)  ||
       (cas_parity_odd( pInputPinKeyToken->keyRightfffi“) == FALSE) )
  {
    Cas_proc_retc( pCprbOut, CP_KDATA_NOTODD );
    break;
  }
}

/*****************************************************************************/
/* If the output PIN key is used, determine which master key to use to       */
/* decipher the output PIN key.                                              */
/*****************************************************************************/
if ( Rule2 == PIN1_KEYWORD_2O )
{

  switch( cas_master_key_check( pOutputPinKeyToken ) )
  {
    case OLD :
      /*********************************************************************/
      /* The key token's MKVP matches the old master key's MKVP. Generate a */
      /* warning reason code that a key is encrypted under the old master  */
      /* key.                                                              */
      /*********************************************************************/
      MKSelector.mk_register = old_mk;
```

```
      Cas_proc_retc( pCprbOut, RT_OMK_TOKEN_USED );
      break;

    case CURRENT :
      /*************************************************************************/
      /* The key token's MKVP matches the current master key's MKVP.           */
      /*************************************************************************/
      MKSelector.mk_register = current_mk;
      break;

    case OUT_OF_DATE :
    default :
      /*************************************************************************/
      /* The key token's MKVP doesn't match current or old master key's        */
      /* MKVP.  We don't know what to do with the key token.                   */
      /*************************************************************************/
      Cas_proc_retc( pCprbOut, RT_KEY_INV_MKVN );
      return;
      break;
  }

  /***************************************************************************/
  /* Decipher the input PIN key under the corresponding master key.          */
  /***************************************************************************/
  ReturnMsg = triple_decrypt_under_master_key_with_CV( &MKSelector,
                                       &(pOutputPinKeyToken->cvBase),
                                       &(pOutputPinKeyToken->keyLeftffl0“),
                                       &(pOutputPinKeyToken->keyLeftffl0“) );

  if ( ReturnMsg != mk_NO_ERROR )
  {
    Cas_proc_retc( pCprbOut, ReturnMsg );
    return;
  }

  ReturnMsg = triple_decrypt_under_master_key_with_CV( &MKSelector,
                                       &(pOutputPinKeyToken->cvExten),
                                       &(pOutputPinKeyToken->keyRightffl0“),
                                       &(pOutputPinKeyToken->keyRightffl0“) );

  if ( ReturnMsg != mk_NO_ERROR )
  {
    Cas_proc_retc( pCprbOut, ReturnMsg );
    return;
  }

  /***************************************************************************/
  /* Generate warning return code if the PIN key does not have odd parity.  */
  /***************************************************************************/
  for ( i = 0; i < SIZE_OF_DES_KEY; i++ )
  {
    if ( (cas_parity_odd( pOutputPinKeyToken->keyLeftffli“ ) == FALSE)  ||
         (cas_parity_odd( pOutputPinKeyToken->keyRightffi“) == FALSE) )
    {
      Cas_proc_retc( pCprbOut, CP_KDATA_NOTODD );
      break;
    }
  }

} /* Rule2 is keyword 2o */

/*****************************************************************************/
/* Processing required for this verb based on inputs.                        */
/*****************************************************************************/
/* ... */
for ( i = 0; i < 8; i++ )
{
  if( Rule1 == PIN1_KEYWORD_1W )
    OutputPinBlockffl i “ = pExtraDataffl i “ ¬ pPinBlockffl i “;
  else
    OutputPinBlockffl i “ = pExtraDataffl i “ & pPinBlockffl i “;
}
```

```
/*****************************************************************************/
/* Reencipher the key token if the token was enciphered under the old       */
/* master key.                                                               */
/*****************************************************************************/
if ( pCprbOut->secy_return_code == RT_OMK_TOKEN_USED )
{
  if ( cas_master_key_check( pInputPinKeyToken ) == OLD )
  {
    MKSelector.mk_register = current_mk;
    /*************************************************************************/
    /* Encipher the input PIN key under the new master key.                  */
    /*************************************************************************/
    ReturnMsg = triple_encrypt_under_master_key_with_CV( &MKSelector,
                                        &(pInputPinKeyToken->cvBase),
                                        &(pInputPinKeyToken->keyLeftfff0"),
                                        &(pInputPinKeyToken->keyLeftfff0") );

    if ( ReturnMsg != mk_NO_ERROR )
    {
      Cas_proc_retc( pCprbOut, ReturnMsg );
      return;
    }

    ReturnMsg = triple_encrypt_under_master_key_with_CV( &MKSelector,
                                        &(pInputPinKeyToken->cvExten),
                                        &(pInputPinKeyToken->keyRightfff0"),
                                        &(pInputPinKeyToken->keyRightfff0") );

    if ( ReturnMsg != mk_NO_ERROR )
    {
      Cas_proc_retc( pCprbOut, ReturnMsg );
      return;
    }

    /*************************************************************************/
    /* Complete the target key token.                                        */
    /*************************************************************************/
    /* Get the MKVP of the current master key and put it in the token.      */
    /*************************************************************************/
    CasCurrentMkvp( &MKSelector, (UCHAR *) &MKVP );
    memcpy( pInputPinKeyToken->mkvp,
            &MKVP,
            sizeof( pInputPinKeyToken->mkvp ));

    /*************************************************************************/
    /* Calculate the TVV and copy it to the token.                           */
    /*************************************************************************/
    pka96_tvvgen( DES_TOKEN_LENGTH, (UCHAR *) pInputPinKeyToken, &TVV );
    memrev( (UCHAR *) &(pInputPinKeyToken->tvv), (UCHAR *) &TVV, TVV_LENGTH );

  } /* input PIN key enciphered under the old master key */

  if ( Rule2 == PIN1_KEYWORD_2O )
  {

    if ( cas_master_key_check( pOutputPinKeyToken ) == OLD )
    {
      MKSelector.mk_register = current_mk;
      /***********************************************************************/
      /* Encipher the input PIN key under the new master key.                */
      /***********************************************************************/
      ReturnMsg = triple_encrypt_under_master_key_with_CV( &MKSelector,
                                        &(pOutputPinKeyToken->cvBase),
                                        &(pOutputPinKeyToken->keyLeftfff0"),
                                        &(pOutputPinKeyToken->keyLeftfff0") );

      if ( ReturnMsg != mk_NO_ERROR )
      {
        Cas_proc_retc( pCprbOut, ReturnMsg );
        return;
```

```
      }

      ReturnMsg = triple_encrypt_under_master_key_with_CV( &MKSelector,
                                          &(pOutputPinKeyToken->cvExten),
                                          &(pOutputPinKeyToken->keyRightff0“),
                                          &(pOutputPinKeyToken->keyRightff0“));

      if ( ReturnMsg != mk_NO_ERROR )
      {
        Cas_proc_retc( pCprbOut, ReturnMsg );
        return;
      }

      /*************************************************************************/
      /* Complete the target key token.                                        */
      /*************************************************************************/
      /* Get the MKVP of the current master key and put it in the token.    */
      /*************************************************************************/
      CasCurrentMkvp( &MKSelector, (UCHAR *) &MKVP );
      memcpy( pOutputPinKeyToken->mkvp,
              &MKVP,
              sizeof( pOutputPinKeyToken->mkvp ));

      /*************************************************************************/
      /* Calculate the TVV and copy it to the token.                           */
      /*************************************************************************/
      pka96_tvvgen( DES_TOKEN_LENGTH, (UCHAR *) pOutputPinKeyToken, &TVV );
      memrev( (UCHAR *) &(pOutputPinKeyToken->tvv),
              (UCHAR *) &TVV,
              TVV_LENGTH );

    } /* output PIN key enciphered under the old master key */
  }
}

/*****************************************************************************/
/* Build the reply CPRB.                                                     */
/* ------------------------------------------------------------------        */
/* |Sub-    |Rule  |Rule     |Verb  |Verb     |Key   |Key Fields|           */
/* |Function|Array |Array    |Data  |Unique   |Block |          |           */
/* |Code    |Block |Elements |Block |Data     |Length|          |           */
/* |        |Length|         |Length|         |      |          |           */
/* |        ---------------------------------------------------------        */
/* |        | ( Length = X ) | ( Length = Y ) | ( Length = Z )  |           */
/* |        ---------------------------------------------------------        */
/* |0       |2     |4        |4+X   |6+X      |6+X+Y |8+X+Y     |8+X+Y+Z    */
/* ------------------------------------------------------------------        */
/* For UDX zudxPIN1, the output PIN block and the key tokens are returned   */
/* to the caller.                                                            */
/*****************************************************************************/
pCprbOut->reply_parm_block = pRepBlk;
pReplyBlockPtr = (UCHAR *) pRepBlk;
ReplyBlockLength = 4;

/*****************************************************************************/
/* Add the rule array which is empty.                                        */
/*****************************************************************************/
pRepBlk->rule_array_length = NO_RULEARRAY;

/*****************************************************************************/
/* Add the output PIN block to the VUD.                                      */
/*****************************************************************************/
ReplyBlockLength += BuildParmBlock( pReplyBlockPtr + ReplyBlockLength, 1,
                                    (USHORT) PIN_BLOCK_SIZE, &OutputPinBlock );

/*****************************************************************************/
/* Add the key block.                                                        */
/*****************************************************************************/
InputKeyHeader.Length = KEY_HDR_LEN + DES_TOKEN_LENGTH;
InputKeyHeader.Flags = DES96_TYPE | ACTION_NOOP;
OutputKeyHeader.Length = KEY_HDR_LEN + DES_TOKEN_LENGTH;
OutputKeyHeader.Flags = DES96_TYPE | ACTION_NOOP;
```

```
  ReplyBlockLength += BuildParmBlock(
                                pReplyBlockPtr + ReplyBlockLength, 4,
                                KEY_HDR_LEN, &InputKeyHeader,
                                (USHORT) DES_TOKEN_LENGTH, pInputPinKeyToken,
                                KEY_HDR_LEN, &OutputKeyHeader,
                                (USHORT) DES_TOKEN_LENGTH, pOutputPinKeyToken );

  /*****************************************************************************/
  /* Enough room in the CRPB?                                                  */
  /*****************************************************************************/
  pCprbOut->replied_parm_block_length = ReplyBlockLength;

  if ( pCprbOut->reply_parm_block_length < pCprbOut->replied_parm_block_length )
  {
    Cas_proc_retc( pCprbOut, REPLY_TOO_LONG );
    return;
  }

  /*****************************************************************************/
  /* Return to the caller.                                                     */
  /*****************************************************************************/
  Cas_proc_retc ( pCprbOut, S_OK ) ;
  return;

} /* end-of zudxPIN1( ) */
```

Appendix E. UDX Sample Code - Workstation Host - Test Code

This appendix contains a listing of the sample file *sxt_samp.c*. This file is a skeleton for the design of the workstation host piece of a CCA extension for use in the initial testing of the coprocessor piece of the UDX.

```
/*----------------------------------------------------------------------------*/
/* Module Name: SXT_SAMP.C                                                    */
/*                                                                            */
/*  Sample callable service for UDX - PIN Block Processing Service            */
/*                                                                            */
/* (C) Copyright IBM Corporation, 2001                                        */
/*----------------------------------------------------------------------------*/
/*  Function: This file contains the sample SAPI CCA API extension verb       */
/*            zPIN1. It illustrates interfacing with the application and   */
/*            shows how to send the request to the cryptographic adapter      */
/*            card. This program will process an encrypted PIN block (assume  */
/*            a proprietary block form) and return the block encrypted under  */
/*            the original or a second key.                                   */
/*                                                                            */
/* Module Type:                                                               */
/*   Attributes:  Serial usable                                               */
/*   Language: IBM Visual Age C++ Version 3.00                                */
/*                                                                            */
/* Entry Points:                                                              */
/*   zPIN1                                                                    */
/*                                                                            */
/*  Inputs:   Rule Array Count  Number of keywords passed in rule array       */
/*                                                                            */
/*            Rule Array        Keywords (0, 1 or 2 keywords may be passed.   */
/*                                                                            */
/*            Input Pin Key Id  Input PIN encrypting key identifier. The      */
/*                              input PIN block is enciphered under this key. */
/*                              The identifier is a token.                    */
/*                                                                            */
/*            Input Pin Block   Enciphered PIN block to be processed.         */
/*                                                                            */
/*            Output Pin Key Id Output PIN encrypting key identifier or a     */
/*                              null token. The identifier is a token. If     */
/*                              the key is not used, a null token is supplied */
/*                                                                            */
/*            Extra Data        Extra data to be used in processing the PIN   */
/*                              block (always 8 bytes).                       */
/*                                                                            */
/* Processing: 1. Build the request parameter block and request CRPB.         */
/*             2. Submit the request.                                         */
/*             3. Process the reply CPRB and parameter block.                 */
/*             4. Return to caller.                                           */
/*                                                                            */
/*  Outputs:  Return Code       Return code from processing                   */
/*                                                                            */
/*            Reason Code       Reason code from processing                   */
/*                                                                            */
/*            Output Pin Block  Processed enciphered PIN block                */
/*                                                                            */
/*            Input Pin Key Id  Reenciphered token if the key was             */
/*                              enciphered under the old master key.          */
/*                                                                            */
/*            Output Pin Key Id Reenciphered token if the key was             */
/*                              enciphered under the old master key.          */
/*                                                                            */
/*  Change History                                                           */
/*                                                                            */
/*  Date      Programmer  Description                                         */
/*  --------  ----------  -----------                                         */
/*  03/14/01  kbk         Created                                             */
```

```
/*                                                                           */
/*---------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------*/
/* Define compiler variable so entry points are not redefined.               */
/*---------------------------------------------------------------------------*/
#define CSUC_32BIT_SOURCE
#include "csunincl.h"                  /* Callable services prototypes       */

/*---------------------------------------------------------------------------*/
/* Includes and local defines.                                               */
/*---------------------------------------------------------------------------*/
#include <string.h>
#include <stdlib.h>
#include "cmncryt2.h"
#include "cmnerrcd.h"
#include "safhead1.h"
#include "safcextn.h"
#define EXTRA_DATA_LENGTH 8
#define PIN_BLOCK_LENGTH 8
#define ZUDX_CODE 0x5258
/*---------------------------------------------------------------------------*/
/*                    FUNCTION : zPIN1                                       */
/*---------------------------------------------------------------------------*/
void SECURITYAPI zPIN1(
                         long  *pReturnCode,
                         long  *pReasonCode,
                         long  *pRuleArrayCount,
                         UCHAR *pRuleArray,
                         UCHAR *pInputPINKeyId,
                         UCHAR *pInputPINBlock,
                         UCHAR *pExtraData,
                         UCHAR *pOutputPINKeyId,
                         UCHAR *pOutputPINBlock )
{

   CPRB_ptr pCprb;                     /* CPRB pointer                       */
   REQUEST_REPLY_BUF *pRequestReplyBuffer; /* buffer area pointer for request */
                                       /* and reply CPRB/parameter areas.    */
   UCHAR * pRequestParmBlock;          /* request parm blk pointer           */
   USHORT rqpb_len;                    /* request parm buffer length         */

   KEY_FIELD_HEADER key_hdr_input;     /* header for key parm                */
   KEY_FIELD_HEADER key_hdr_output;    /* header for key parm                */

   long msg;                           /* message of SAPI routines           */
   UCHAR * pReturnVUD;                 /* pointer to returned PIN block      */

   /*------------------------------------------------------------------------*/
   /* Check if return code or reason code is NULL.                           */
   /*------------------------------------------------------------------------*/
   if (pReturnCode == NULL ]] pReasonCode == NULL)
     return;                                  /* return right away         */

   /*------------------------------------------------------------------------*/
   /* Check if pointers are NULL.                                            */
   /*------------------------------------------------------------------------*/
   if ( pInputPINBlock == NULL ]] pInputPINKeyId  == NULL ]]
        pExtraData     == NULL ]] pOutputPINKeyId == NULL )
     {
     CSUC_PROCRETC(pReturnCode, pReasonCode, E_NULL_PTR);
     return;                                  /* return error if any of    */
     }                                        /* the conditions are met    */

   /*------------------------------------------------------------------------*/
   /* Set return code and reason code to zero.                               */
   /*------------------------------------------------------------------------*/
   *pReturnCode = 0;
   *pReasonCode = 0;

   /*------------------------------------------------------------------------*/
   /* Allocate space for a working area.                                     */
```

```
/*-------------------------------------------------------------------------*/
pRequestReplyBuffer = malloc( sizeof( REQUEST_REPLY_BUF ));

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

pCprb = (CPRB_ptr) &(pRequestReplyBuffer->request_buf[ 0 ]);
pRequestParmBlock = &(pRequestReplyBuffer->request_buf[ 0 ])
                                              + sizeof( CPRB_structure );
/*-------------------------------------------------------------------------*/
/* Request parameter block                                                 */
/*                                                                         */
/* +---------+------+-------//-+------+------//-+------+-----//--+         */
/* ]Sub-     ]Rule  ]Rule      ]Verb Unique     ]Key Block       ]         */
/* ]Function ]Array ]Array     ]Data Fields     ]Fields          ]         */
/* ]Code     ]Block ]Elements  ]      ]         ]      ]         ]         */
/* ]         ]Length]          ]Length] Data    ]Length] Fields  ]         */
/* ]         ]------+-------//-+------+------//-+------+-----//--+         */
/* ]         ] ( Length = X )  ] ( Length = Y ) ] ( Length = Z ) ]         */
/* ]         ]------+-------//-+------+------//-+------+-----//--+         */
/* ]0        ]2     ]4         ]2+X   ]4+X      ]2+X+Y ]4+X+Y    ]         */
/* +---------+------+-------//-+------+------//-+------+-----//--+         */
/*-------------------------------------------------------------------------*/
/* Part 1 of 4. 2-byte subfunction code                                    */
/*-------------------------------------------------------------------------*/
*((USHORT *)pRequestParmBlock) = htoas(ZUDX_CODE);
rqpb_len = 2;

/*-------------------------------------------------------------------------*/
/* Part 2 of 4. Rule array block                                           */
/*-------------------------------------------------------------------------*/
if (( *pRuleArrayCount < RAC_MIN ) ]]
    ( *pRuleArrayCount > RAC_MAX ))
  {
  free( pRequestReplyBuffer );
  CSUC_PROCRETC( pReturnCode, pReasonCode, E_RULE_ARRAY_CNT );
  return;
  }
else
  {
  rqpb_len += BuildParmBlock( pRequestParmBlock + rqpb_len, 1,
                              (USHORT) ( 8 * *pRuleArrayCount ), pRuleArray );
  }

/*-------------------------------------------------------------------------*/
/* Part 3 of 4. Verb-unique data block: Input PIN block and extra data     */
/*-------------------------------------------------------------------------*/
rqpb_len += BuildParmBlock( pRequestParmBlock + rqpb_len, 2,
                            PIN_BLOCK_LENGTH, pInputPINBlock,
                            EXTRA_DATA_LENGTH, pExtraData );

/*-------------------------------------------------------------------------*/
/* Part 4 of 4. Key block                                                  */
/*              The keys are in this order:                                */
/*                  Input PIN Key Id  (token)                              */
/*                  Output PIN Key Id (token)                              */
/*-------------------------------------------------------------------------*/
/* Input PIN Key Id                                                        */
key_hdr_input.Length = htoas(KEY_HDR_LEN + DES_TOKEN_LENGTH);
key_hdr_input.Flags = htoas(DES96_TYPE ] ACTION_NOOP);

/* Output PIN Key Id */
key_hdr_output.Length = htoas(KEY_HDR_LEN + DES_TOKEN_LENGTH);
key_hdr_output.Flags = htoas(DES96_TYPE ] ACTION_NOOP);

/*-------------------------------------------------------------------------*/
/* Build the key block.                                                    */
/*-------------------------------------------------------------------------*/
rqpb_len += BuildParmBlock(pRequestParmBlock + rqpb_len, 4,
```

```
                    KEY_HDR_LEN, &key_hdr_input,
                    (USHORT)DES_TOKEN_LENGTH, pInputPINKeyId,
                    KEY_HDR_LEN, &key_hdr_output,
                    (USHORT)DES_TOKEN_LENGTH, pOutputPINKeyId);

  /*-------------------------------------------------------------------------*/
  /* build CPRB                                                              */
  /*-------------------------------------------------------------------------*/
  CSUC_BULDCPRB( pCprb,
                 (UCHAR *) ESSS_FUNCTION_ID_S,
                 rqpb_len,
                 pRequestParmBlock,           /* Request parm. buffer       */
                 0, (UCHAR *) NULL,             /* Request data buffer        */
                 sizeof( pRequestReplyBuffer->reply_buf ),
                 pRequestReplyBuffer->reply_buf, /* Reply parameter buffer    */
                 0, (UCHAR *) NULL);            /* Reply data buffer          */

  /*-------------------------------------------------------------------------*/
  /* Call Security Server using function CSNC_SP_SCSRFBSS.                   */
  /*-------------------------------------------------------------------------*/
   CSNC_SP_SCSRFBSS((CPRB_ptr) pCprb, (long *) &msg);


  /* Note:  CSUC_PROCRETC returns ERROR if the error code in msg is higher    */
  /* than the error code already in *pReturn_code and *pReason_code.          */
  /* msg is the return code and reason code, concatenated in a single long    */
  /* integer - for example, msg=00080012 is equivalent to return code 8,      */
  /* reason code 12.                                                          */

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

  /*-------------------------------------------------------------------------*/
  /* Process the returned data, which is in the Reply Parameter Block.       */
  /*-------------------------------------------------------------------------*/
  /* Examine the Reply Parameter Block to make sure it is OK.  If not,       */
  /* something is wrong in the adapter - it should return valid data.        */
  if ( ! parm_block_valid( (CPRB_structure *) pCprb, SEL_REPLY_BLK ))
    CSUC_PROCRETC(pReturnCode, pReasonCode, CP_DEV_HWERR);

  else
    {
    /* The output PIN block is returned in the VUD block. Since the rule      */
    /* array block is empty, the VUD block is two bytes after the             */
    /* rule array block. This corresponds to the first rule array element.    */
    pReturnVUD = &pCprb->reply_parm_block->first_rule_array_element;
    memcpy( pOutputPINBlock, pReturnVUD, PIN_BLOCK_LENGTH );
    }

   free( pRequestReplyBuffer );
   return;

} /* end of zPIN1 */
```

Appendix F. Moving a UDX from the Model 1 Card to the Model 2 Card

Because of the changes in the functionality of CCA, transferring code from a model 1 environment to a model 2 environment may require changes to the UDX code, particularly in the following segments:

Master Key Manager Changes

For the model 2 PKA token structures, the verification pattern stored in the operational key is the 16-byte MDC4 hash of the master key. Thus, there are new master key structures which contain this 16-byte field as well as the 20-byte SHA-1 hash of the older PKA tokens. This has no effect on your UDX code unless you are building your own master keys. (The functions provided (combine_mk_parts, load_mk_from_shares, and so on) will take care of this automatically.)

Because of the separate verification pattern, separate master keys are stored for the PKA keys and the DES keys. The mk_selectors structure has a new field, type_mks, which may be one of SYM_MK (for symmetric keys), ASYM_MK (for PKA keys), BOTH_MK (if you are using only one master key set). This structure must be updated/set if you are using any of the functions in Chapter 6, CCA Master Key Manager Functions, especially:

clear_master_keys
get_mk_verification_pattern
ede3_triple_decrypt_under_master_key_with_CV
ede3_triple_encrypt_under_master_key_with_CV
ede3_triple_decrypt_under_master_key_with_CV
ede3_triple_encrypt_under_master_key_with_CV
triple_decrypt_under_master_key
triple_encrypt_under_master_key
triple_decrypt_under_master_key_with_CV
triple_encrypt_under_master_key_with_CV

These functions already used the mk_selectors parameter, so the structure will be filled improperly unless the code is changed.

All of other master key manager functions except init_master_keys() and reinit_master_keys() now take the mk_selectors parameter. The functions used in the old toolkit are still available, however they map to the SYM_MK keyset automatically. Any calls to these functions should be checked and if necessary, changed to use the appropriate set of keys. For example, if you are using PKA keys, you will want to replace any calls to generate_random_master_key() with:

```
mk_selector.mk_set=MK_SET_DEFAULT;
mk_selector.mk_register = MK_NEW;
mk_selector.mk_type = ASYM_MK;
mkmGenerateRandomMK(mk_selector);
```

The KEY_STORE_MKVP_TOKEN structure, the first structure in the key storage file, has changed (names only, not the actual byte information.) If you have written a function which sets up a key storage file which uses the CCA key storage structures, this will need to be changed.

The CPRB structure has changed (again, only in the names of the fields). Some fields which were included in reserved fields have been added to allow closer linkage with zSeries functions (domain, and so on). There will be no issues here as long as you have initialized the CPRB structure to 0x00 before filling, or use the CSUC_BULDCPRB() function to fill the CPRB structure.

Makefile Changes

The UDX toolkit for the Model 2 card uses a different set of basic library functions. When building the card-side portion of the UDX, it is important to link clib.lib from the ...\UDXTK\002\lib\nt\msvc or ...\UDXTK\002\lib\zSeries\msvc, or ...\UDXTK\002\lib\nt\vac or ...\UDXTK\002\lib\zSeries\vac directory, rather than scctk\002\lib\scc\nt\msvcmasm or scctk\002\lib\scc\nt\vacppmsm. Adjust your makefiles accordingly. The other scctk libraries may be included from the scctk library directories, as they were for the Model 1.

Appendix G. Reserved Values

Certain values have been reserved for the use of UDX developers. IBM will not use these values in future upgrades to the CCA, so there will be no overlap between a UDX using these values and an upgrade of the toolkit. The specific values are as follows:

For completion codes: The values between 0x5000 and 0x5FFF have been reserved for UDX writers.

For access control points: For zSeries, the values between 0x8000 and 0xEFFF may be used. (Values between 0xF000 and 0xFFFF will be used on zSeries by IBM UDX writers.)

For subfunction codes: The values between “XA” (0x5841) to “XZ” (0x585A), “YA” (0x5941) to “YZ” (0x595A), “X0” (0x5830) to “X9” (0x5839), and “Y0” (0x5930), to “Y9” (0x5939) are reserved for UDX writers. The values between “WA” (0x5741) to “WZ” (0x575A) and “W0” (0x5730) to “W9” (0x5739) are also reserved for UDX writers. On the zSeries server, these values (beginning with “W”) will be used by IBM UDX writers. On other platforms, they are available for all UDX writers.

For DES control vectors: Bits 4, 5, and 61 will not affect or be affected by the import or export of a key. Bits 4 and 5 will be ignored by the CCA at all times. Bit 61 will prevent a key from use in any standard CCA verb, thus reserving a key for use only in a UDX function.

Appendix H. Data Structures

This appendix identifies useful data structures from the toolkit header files.

Structures Used in Communications Between NT Host and Coprocessor

These structures may be used on the coprocessor or on the NT host machine. If you are writing code for the zSeries 4758, the host side instructions in this section will be useful only for building the NT test DLL.

A `REQUEST_REPLY_BUF` structure should be declared in the host function to allocate the data storage for the CPRB Structures and the request and reply buffers. This structure has two fields, both of 5120 bytes (BLK_LEN_MAX).

REQUEST_REPLY_BUF

Field name	Size of field	Purpose
request_buf	5120 bytes	Holds the CPRB structure and the request block.
reply_buf	5120 bytes	Holds the (return) CPRB structure and the reply block.

The `REQUEST_REPLY_BUF` structure is filled with the following structures (hence they are declared as pointers into the `REQUEST_REPLY_BUF` structure).

First, a CPRB structure:

Note that the (request) CPRB structure is filled (as completely as it needs to be) by calling `CSUC_BULDCPRB()` with the appropriate lengths and pointers from within the host function. Fields not filled by this function will be filled by the Security Server when the coprocessor is called. Within the coprocessor code, the output CPRB fields are filled by copying the values from the input CPRB. Changing the values of these fields is not recommended, except for the `replied_parm_block_length` and `replied_data_block_length` fields in the coprocessor code.

Changing the values in a “FILLED AUTOMATICALLY” field will have one of two effects:

1. SECY will overwrite the changed value with the correct value.
2. The call will fail because of an invalid value.

Since the `CPRB_structure` is used exclusively as a pointer into the `REQUEST_REPLY_BUF` structure, the type `CPRB_ptr` has been typedefed as a pointer to the `CPRB_structure`.

CPRB_structure, or *CPRB_ptr

Field name	Size/Type	Purpose
CPRB_length	USHORT	This field should contain D'112' (little endian).

Field name	Size/Type	Purpose
cprb_version_id	1 byte	Flag indicating the version of this structure.
MAC_content_flags	1 byte	Flags for the message authentication function. FILLED AUTOMATICALLY
SRPI_return_code	unsigned long	Return code from SECY.
SRPI_verb_type	1 byte	This field should have the value X'1' . FILLED AUTOMATICALLY
reserved_1	1 byte	This field should contain X'0'.
function_id	2 bytes	This field should contain 'T2'
S390Checkpoint	1 byte	FILLED AUTOMATICALLY.
reserved_2	1 byte	This field should contain X'0'.
req_parm_block_length	unsigned short	Request parameter block length (little-endian).
req_parm_block	pointer (4 bytes)	Address of the request parameter block.
req_data_block_length	unsigned long	Request data block length (little-endian)
req_data_block_addr	pointer (4 bytes)	Address of request data block.
reply_parm_block_length	unsigned short	Reply parameter block length (little-endian)
pad_001	unsigned short	Number of bytes to pad to ensure proper alignment. FILLED AUTOMATICALLY. (little-endian)
reply_parm_block	pointer (4 bytes)	Address of reply parameter block.
reply_data_block_length	unsigned long	Reply data block length (little-endian)
reply_data_block	pointer (4 bytes)	Address of reply data block
secy_return_code	unsigned long	This is the returnCode]]reasonCode combination.
replied_parm_block_length	unsigned short	The length of the reply data returned from the coprocessor in the reply parameter block.
MAC_data_length	unsigned short	The length of the data to be authenticated. FILLED AUTOMATICALLY. (little-endian)
replied_data_block_length	unsigned long	The length of the reply data returned from the coprocessor in the reply DATA block. (little-endian)
requestor_id	unsigned short	ID of requestor FILLED BY ROUTER.
resource_origin	8 bytes	FILLED AUTOMATICALLY.

Field name	Size/Type	Purpose
MAC_value	4 bytes	FILLED AUTOMATICALLY.
logon_identifier	8 bytes	FILLED AUTOMATICALLY.
Domain	unsigned short	Usage/control domain. FILLED AUTOMATICALLY.
UsageDomainMask	4 bytes	Usage domain mask. FILLED AUTOMATICALLY.
ControlDomainMask	4 bytes	Control domain mask. FILLED AUTOMATICALLY.
S390EnforcementMask	4 bytes	S390 Enforcement mask. FILLED AUTOMATICALLY.
reserved_for_requestors	6 bytes	Reserved for requestors. FILLED AUTOMATICALLY.
secy_name_length	unsigned short	Length of the security server name (8 bytes) FILLED AUTOMATICALLY.
server_name	8 bytes	Security server name (“SECY ”)FILLED AUTOMATICALLY.

On the host side, you will only need one `CPRB_ptr`, since the request CPRB you build will be replaced by the reply CPRB from the coprocessor during the call to `CSNC_SP_SCSRFBSS( )`. On the coprocessor, two of the parameters for a command function are `pCprbIn`, and `pCprbOut`. Therefore, you do not need to declare either a `REQUEST_REPLY_BUF` or a `CPRB_ptr`.

Following the `CPRB_structure` in the buffer is a request block:

The `ESSS_request_block_structure` defines the structure for the request or reply block. Since request and reply blocks are variable length, this structure is used purely as a pointer into the `request_buf` or `reply_buf` field of the `REQUEST_REPLY_BUF` structure. `RBFPTR` is typedefed as a pointer to an `ESSS_request_block_structure`, and thus is more commonly used.

On the host side, you may want to declare an `RBFPTR` for the request buffer. On the coprocessor code, you may want to declare an `RBFPTR` for both the request buffer and the reply buffer.

ESSS_request_block_structure, or *RBFPTR

Field name	Size of field	Purpose
subfunction_code	unsigned short	Holds the two-byte subfunction code in little-endian format.
rule_array_length	unsigned short	Total length of rule array and this field, in little-endian format.
first_rule_array_element	1 byte	First character of first rule array element, if rule_array_length is greater than 2. Otherwise, this will be the first byte of the verb unique data length field.

Filling the rule array is easy using the BuildParmBlock() function:

```
BuildParmBlock (ptr1,
                1,
                SIZE_OF_RULE * (*pRuleCount), pRuleArray);
```

To parse a rule array with the rule_check( ) function, two more structures are used. A pointer to a RULE_BLOCK is passed to the function to be parsed. Note that the rule array format within the ESSS_request_block structure is, in fact, a RULE_BLOCK:

RULE_BLOCK

Field name	Size/Type	Purpose
length	unsigned short	Total length of rule block. (little-endian)
data	80 bytes	Up to 10 (8-byte) rules.

The other structure required is a RULE_MAP structure. This maps 8-byte strings into a value array, assigning a unique value to each string, and 1 or more strings to each position in the array, depending on mutual exclusion issues.

RULE_MAP

Field name	Size/Type	Purpose
keyword	9 bytes (8 chars plus null terminator)	String to be matched in rule array.
order_no	1 byte	Group number: all rules which are mutually exclusive to each other will have the same group number.
map_value	int (4 bytes)	The numeric value associated with this rule.

To check the values in the rule array, use the rule_check() function:

```
rule_check((RULE_BLOCK *)&pReq->rule_array_length,
            sizeof(aRuleMap)/sizeof(RULE_MAP),
            aRuleMap,
            aRuleValue,
            &returnMessage);
```

Immediately following the rule array in the REQUEST_REPLY_BUF is the verb unique data. Two types of structures are supplied for working with verb unique data, the VUD_DATA_RECORD, which is a length/tag/data structure (the data preceded by a DATA_RECORD_HEADER structure), and the verb_unique_data_structure, which is a length/data structure.

DATA_RECORD_HEADER

Field name	Size	Purpose
Length	unsigned short	Length of this verb data.
Flag	unsigned short	User defined: usually type of data.

```
#define DATA_HEADER_LENGTH  sizeof( DATA_RECORD_HEADER )
```

If you want to use the length/tag/data format for your verb unique data, declare a DATA_RECORD_HEADER structure to place before the data, and use the BuildParmBlock() function to place it before the data.

```
BuildParmBlock(ptr,
               2,
               DATA_HEADER_LENGTH, &DataHeader,
               dataLength, &Data[0]);
```

The FindFirstDataBlock() function returns a pointer to a VUD_DATA_RECORD, so that you can access your data in this format easily:

VUD_DATA_RECORD

Field name	Size/Type	Purpose
Length	unsigned short	Length of this verb data
Flag	unsigned short	See above DATA_RECORD_HEADER
Data	1 byte	The first byte of the data.

```
FindFirstDataBlock(pCPRB, SEL_REPLY_BLK, &pVerbDataRecord);
 if(pVerbDataRecord->Flag == EncryptedKey)
 {
 memcpy(pKeyParameter, &pVerbDataRecord->Data,
 pVerbDataRecord->Length - DATA_HEADER_LENGTH);
 }
```

On the other hand, if you have no need to access the Flags field, you can use the verb_unique_data_structure type instead:

verb_unique_data_structure

Field name	Size/Type	Purpose
verb_unique_data_length	unsigned short	Length of this verb data
verb_unique_data	1 byte	The first byte of the data

```
BuildParmBlock(ptr,
               2,
               sizeof(short), &vudLength,
               dataLength, &Data);
```

To retrieve the above data, you must first cast the verb_unique_data_structure as a VUD_DATA_RECORD:

```
FindFirstDataBlock ( pCPRB, SEL_REPLY_BLK, (VUD_DATA_RECORD **)&pVerbUniqueDataStructure);
                   *pLengthParm = atohs(pVerbUniqueDataStructure->
                    verb_unique_data_length) - LENGTH_FIELD_SIZE;
                    memcpy(pReturnedData,
                    &pVerbUniqueDataStructure->verb_unique_data, *pLengthParm);
```

If the only piece of data which is being passed has a fixed length (for example, if it is a structure), you need not use either of the verb structures shown:

```
BuildParmBlock(ptr,
               1,
               sizeof(Structure), &Structure);
```

Then to access the data:

```
FindFirstDataBlock(pCPRB, SEL_REPLY_BLOCK, (VUD_DATA_RECORD **)&pData);
                  memcpy(&Structure, pData, sizeof(Structure));
```

If you use this method, you must not pass more than one piece of verb unique data, as the `FindNextDataBlock()` function uses the length field to determine where to look for the next piece of data.

Following the verb unique data, the key data is organized into key fields and key data structures. Each key is preceded by a `KEY_FIELD_HEADER` structure:

KEY_FIELD_HEADER

Field name	Size/Type	Purpose
Length	unsigned short	Total length of this key block. (little-endian)
Flags	unsigned short	Flags indication action required by the Security Server and type of key.

On the host side, you will need to declare a `KEY_FIELD_HEADER` structure for each key you will be passing to the coprocessor. On the coprocessor, you will need to declare a `KEY_FIELD_HEADER` structure for each key you will be passing to the host. If you are passing a token to be written to the key storage file, you must declare *two* `KEY_FIELD_HEADER` structures, and pass first the label of the key to write to, then the key token to write into the key storage file.

```
BuildParmBlock(ptr,
               4,   // 2 for each key you will be passing
               sizeof(KEY_FIELD_HEADER), &keyFieldHeader1,
               KEY_LABEL_LENGTH, keyLabel,
               sizeof(KEY_FIELD_HEADER), &keyFieldHeader2,
               keyTokenLength, keyToken);
```

The `find_first_key_block()` function returns a pointer to a `key_data_structure`:

key_data_structure

Field name	Size/Type	Purpose
key_field_data_length	unsigned short	Total length of this key data.
key_data	1 byte	First byte of keyFieldHeader.Flags

Since there is no reason to access the first byte of the keyFieldHeader.Flags field, you will usually declare a `generic_key_block_structure` pointer, and cast it as a `key_data_structure` in the function call.

generic_key_block_structure

Field name	Size/Type	Purpose
length	unsigned short	Total length of this key data. (little-endian)
flags	unsigned short	Flag bytes (little-endian) (ignore)
label_or_token	1 byte	First byte of key token or label.

```
find_first_key_block(pCprb, (key_data_structure **)&pGenericKeyBlockStructure,SEL_REQ_BLK);
                     keyLength = atohs(pGenericKeyBlockStructure->length) -
                      sizeof(KEY_FIELD_HEADER);
                     pKeyToken = &pGenericKeyBlockStructure->label_or_token;
```

Notice that the value of the byte in the `label_or_token` field can be used in the macro `TOKEN_LABEL_CHECK` to determine whether the token is a key token with key data or the label of a key in key storage.

If the key which has been passed is an RSA key, some of the functions which manipulate and check it take parameters of type `RsaKeyTokenHeader`:

RsaKeyTokenHeader

Field name	Size/Type	Purpose
tokenId	1 byte	Indicates Internal PKA, External PKA, Label, or “not RSA”
version	1 byte	Version of RSA token
tokenLength	unsigned short	Total length of token (big-endian)
reserved	4 bytes	valued to 0
nextSection	1 byte	First byte of next token section - indicates public or privateModexponent, private Chinese remainder, and so on.

In most cases, you should simply cast the pointer to the token as an `RsaKeyTokenHeader` pointer.

Data Structures for Caching Functions

Only one new data structure is required for the use of the cache functions, the `short_tag_t`:

short_tag_t

Field name	Size/Type	Purpose
tag_1	1 byte	First byte of 2 byte short tag, index into linked list of second bytes.
tag_2	1 byte	Second byte of 2 byte short tag, index into linked list of entries.

You may choose to cast a 2-byte value as a `short_tag_t` for the function call.

Other Useful Data Structures

The `mk_selectors` data structure is used to indicate which of several master keys to use in a given master key function.

mk_selectors

Field name	Size/Type	Purpose
mk_set	unsigned short	Domain of master key set: This should contain the same value as the domain field in the input CPRB structure.
mk_register	enumeration	old_mk, current_mk, new_mk to determine which of the three registers to access.
type_mks	enumeration	SYM_MK, ASYM_MK, Both_MK, to determine which type of master key to use or change.

The `RsaRecoverClearKeyTokenUnderXport()` function requires a type of `double_length_key`.

double_length_key

Field name	Size/Type	Purpose
left	8 bytes	First 8 bytes of key.
right	8 bytes	Second 8 bytes of key.

The functions `load_first_mk_part()` and `combine_mk_parts()` require a `TRIPLE_LENGTH_KEY`:

TRIPLE_LENGTH_KEY

Field name	Size/Type	Purpose
first	8 bytes	First 8 bytes of key.
middle	8 bytes	Second 8 bytes of key.
last	8 bytes	Third 8 bytes of key.

The `dbl_ulong` (double, unsigned, long) data type is used to pass the number of bits of data for SHA1 hashing:

dbl_ulong

Field name	Size/Type	Purpose
upper	unsigned long	The high-order 8 bytes of the value.
lower	unsigned long	The low-order 8 bytes of the value.

Appendix I. 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
e (logo)
Multiprise
OS/390
RACF
S/390 Parallel Enterprise Server

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 zSeries 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.

APF. Authorized Program Facility. A facility that permits identification of programs authorized to use restricted functions.

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 zSeries 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.

CKDS. Cryptographic Key Data Set. In OS/390 ICSF, a VSAM data set that contains DES cryptographic keys used by an installation. Besides the encrypted key value, an entry in the cryptographic key data set contains information about the key.

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 zSeries PCI Cryptographic Coprocessor). 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.

DSS. Digital Signature Standard. A public key algorithm used only for digital signature.

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 zSeries PCI Cryptographic Coprocessor, the zSeries server 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.

L

LIC. Licensed Internal Code

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.

PKDS. Public Key Data Set (PKA Cryptographic Key Data Set)

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 zSeries 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

RACF. Resource Access Control Facility. An IBM-licensed program that provides for access control by identifying and verifying the users to the system, authorizing access to protected resources, logging the detected unauthorized attempts to enter the system, and logging the detected accesses to protected resources.

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 zSeries 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 zSeries 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 zSeries PCI Cryptographic Coprocessor. (2) Synonym for *agent*.

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

T

TKE. Trusted Key Entry

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.

VSAM. Virtual Storage Access Method. An access method for indexed or sequential processing of fixed and variable-length records on direct-access devices. The records in a VSAM data set or file can be organized in logical sequence by means of a key field (key sequence), in the physical sequence in which they are written on the data set or file (entry-sequence), or by means of relative-record number.

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

16-NOV-01, 14:51

Printed in U.S.A.