IBM 4767 PCIe Cryptographic Coprocessor
CCA User Defined Extensions
Reference and Guide
Note: Before using this information and the products it supports, be sure to read the general information under "Notices" on page 331.

Fifth Edition (April, 2019)
This and other publications related to the IBM 4767 PCIe Cryptographic Coprocessor can be obtained in PDF format from the product website. Click on the HSM PCIeCC2 link at www.ibm.com/security/cryptocards, and then click on the Library link.

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

© Copyright International Business Machines Corporation 2016, 2019.
Note to U. S. Government Users—Documentation related to restricted rights Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.
Contents

About this manual ........................................................................................................................................ x
Prerequisite knowledge ................................................................................................................................. x
Organization of this document ...................................................................................................................... xi
Typographic conventions ............................................................................................................................... xi
Syntax diagrams ............................................................................................................................................ xi
Related publications ...................................................................................................................................... xii
Summary of changes .................................................................................................................................... xii
1 Understanding the UDX environment ....................................................................................................... 1
  1.1 Structure of a UDX on a workstation ..................................................................................................... 1
  1.2 Structure of a UDX on the coprocessor .................................................................................................. 4
  1.3 CCA communication structures ........................................................................................................... 7
    1.3.1 Request and reply parameter block format .................................................................................... 7
    1.3.2 Request and reply data blocks ....................................................................................................... 8
    1.3.3 Key blocks .................................................................................................................................... 10
    1.3.4 Operation tracking structure ........................................................................................................ 10
  2 Building a CCA user-defined extension ................................................................................................... 14
    2.1 Files you use in building a UDX ...................................................................................................... 14
    2.2 Workstation piece of a UDX ............................................................................................................ 16
    2.3 Coprocessor piece of a UDX ............................................................................................................ 17
  3 4767 functions ......................................................................................................................................... 20
    3.1 Host-side 4767 API functions ............................................................................................................ 20
    3.2 Coprocessor-side 4767 API functions ................................................................................................ 20
    3.3 Coprocessor-side Fiber-management macros ................................................................................... 21
    3.4 Header files for Fiber macros ........................................................................................................... 21
    3.5 Summary of Macros .......................................................................................................................... 21
      3.5.1 CALLXC_FIBER – Generic fiber calling Macro ........................................................................... 23
      3.5.2 CALLXCDES – Macros to fill request block and call xcDES ........................................................................... 24
      3.5.3 CALLXCTDES – Macros to fill request block and call xcTDES ....................................................... 26
      3.5.4 CALLXCDES3KEY – Macro to fill request block and call xcDES3Key ............................................. 28
      3.5.5 CALLXCDES3KEY_ECB Macro to fill request block and call xcDES3Key in a loop ............... 29
      3.5.6 CALLXCSHA2 – Macro to fill request block and call xcSha2 ......................................................... 31
      3.5.7 CALLXCSHA1 – Macro to fill request block and call xcSha1 ......................................................... 32
      3.5.8 Communication functions ........................................................................................................ 33
    3.6 Header files for communications functions ...................................................................................... 33
    3.7 Summary of functions ......................................................................................................................... 33
      3.7.1 BuildParmBlock - build a parameter block ............................................................................... 35
      3.7.2 Cas_proc_retc - prioritize return code ....................................................................................... 38
      3.7.3 CSNC_SP_SCSRFBS - send a request to the coprocessor ................................................................. 39
      3.7.4 CSUC_BUCDPRB - build CGBR ............................................................................................... 41
      3.7.5 CSUC_PROCRETC - prioritize return code ............................................................................ 43
      3.7.6 FindFirstDataBlock - search for address of first data block ......................................................... 44
      3.7.7 FindNextDataBlock - search for address of next data block ..................................................... 45
      3.7.8 find_first_key_block - search for first key data block ................................................................. 46
      3.7.9 find_next_key_block - find address of next key data block ....................................................... 47
      3.7.10 jFindFirstDataBlock - search for address of first data block ................................................... 48
      3.7.11 jFindNextDataBlock - search for address of next data block ................................................ 49
      3.7.12 jfind_first_key_block - search for first key data block ............................................................. 50
      3.7.13 jfind_next_key_block - find address of next key data block .................................................... 51
      3.7.14 jparm_block_valid - examine and verify a parameter block .................................................... 52
      3.7.15 keyword_in_rule_array - search for rule array keyword ....................................................... 53
      3.7.16 okt_build_tokens - build and encrypt one or more key tokens .............................................. 54
      3.7.17 parm_block_valid - examine and verify a parameter block .................................................... 57
      3.7.18 parse_keys - decrypt and parse the keys from a key block ..................................................... 58
3.7.19 rule_check - verify rule array.................................................................62
3.7.20 saf_process_key_label - process key label............................................65
3.7.21 TOK_ACCESS_AND_INIT_BUILD – set up operation tracking token build structure.66
3.7.22 TOK_ACCESS_AND_INIT_PARSE – set up operation tracking token structure for parse_keys......67
4 FCV management functions........................................................................68
4.1 Header files for function control vector management functions................68
4.2 Summary of functions..............................................................................68
4.2.1 getMaximumCurveOrder - get ECC key length........................................69
4.2.2 getSymmetricMaxModulusLength - get RSA key length..........................70
4.2.3 isFunctionEnabled - check whether a function is enabled.......................71
5 Master key manager functions....................................................................73
5.1 Header files for master key manager functions.........................................73
5.2 Overview of the CCA master keys.............................................................73
5.2.1 Location of the master keys....................................................................74
5.2.2 Initialization of the master key SRDI........................................................74
5.2.3 CCA master key manager interface functions.........................................74
5.2.4 Required variables..................................................................................75
5.3 Summary of functions................................................................................75
5.4 Functions to set and manage the master keys............................................78
5.4.1 Summary of functions............................................................................78
5.4.2 clear_master_keys - clear master key.....................................................79
5.4.3 mkmCombineMKParts - combine master key parts.................................80
5.4.4 mkmGenerateMKShares - generate master key shares..............................82
5.4.5 mkmGenerateRandomMK - generate random master key.........................84
5.4.6 init_master_keys - create and initialize master keys...............................86
5.4.7 mkmLoadFirstMKPart - load first master key part....................................87
5.4.8 mkmLoadMKFromShares - load master key shares..................................88
5.4.9 mkmSetMasterKey - set master key......................................................90
5.4.10 reinit_master_keys - reinitialize master keys.........................................92
5.5 Functions to check master key values and status.......................................93
5.5.1 Summary of functions............................................................................93
5.5.2 compute_mk_verification_pattern - compute a verification pattern..........94
5.5.3 get_mk_verification_pattern - read a verification pattern......................96
5.5.4 mkmGetAsymmetricVerificationPattern - read the asymmetric keys verification pattern...97
5.5.5 mkmGetMasterKeyStatus - get master key status..................................98
5.6 Functions to encrypt and decrypt using the master key..............................100
5.6.1 Summary of functions............................................................................100
5.6.2 aes_engine_with_MK - encrypt or decrypt data using the AES or APKA master key.101
5.6.3 aes_unwrap_with_hash_and_MK - decrypt using the AESKW mechanism....103
5.6.4 aes_unwrap_with_MK - decrypt using the AESKW mechanism...............105
5.6.5 aes_wrap_with_hash_and_MK - encrypt using the AESKW mechanism.......107
5.6.6 aes_wrap_with_MK - encrypt using the AESKW mechanism...................109
5.6.7 ede3_triple_decrypt_under_master_key.................................................111
5.6.8 ede3_triple_encrypt_under_master_key................................................113
5.6.9 mkmTDESDecryptUnderMasterKey......................................................115
5.6.10 mkmTDESEncryptUnderMasterKey......................................................117
5.6.11 triple_decrypt_under_master_key........................................................119
5.6.12 triple_decrypt_under_master_key_with_CV..........................................120
5.6.13 triple_encrypt_under_master_key.......................................................122
5.6.14 triple_encrypt_under_master_key_with_CV..........................................123
6 Hashing functions.......................................................................................125
6.1 Header files for SHA functions................................................................125
6.2 Summary of functions............................................................................125
6.3 Chained operations..................................................................................126
6.3.1 computeHMAC_SHA1 - compute HMAC using SHA-1 algorithm.............128
8.5.6 PkhHashQueryWithMK - return master key version
8.5.7 PkhMkvQueryWithMK - return master key version
8.5.8 TokenMkvpMatchMasterKey - test encryption of RSA key
8.5.9 ValidatePkaTokenWithMinModulus - validate RSA key token
8.5.10 VerifyRSAKeyTokenConsistency - verify RSA key token consistency

8.6 Functions to manipulate RSA key tokens

8.6.1 blde_cca_rsa_from_xcrsa_tok - create operational key token
8.6.2 blde_xcrsa_from_cca_rsa_tok - create an RSA token in xc format
8.6.3 CreateInternalKeyTokenWithMK - create operational key token
8.6.4 CreateRsIntegralSectionWithMK - create CCA RSA internal section
8.6.5 GenerateCcaRsaToken - create CCA RSA key token (deprecated)
8.6.6 GenerateRsIntegralToken - create SCC RSA key token (deprecated)
8.6.7 GetModulus - extract and copy RSA modulus
8.6.8 GetPublicExponent - extract and copy public exponent
8.6.9 GetRSAKeyNumSection - get the section containing the name of the key
8.6.10 GetRsaPrivateKeySection - return private key
8.6.11 GetRsaPublicKeySection - return public key
8.6.12 GetRsaPuKeyCertSection - return public key certificate
8.6.13 pka96_tvgenc - calculate token validation value
8.6.14 RecoverPkaClearKeyTokenAESKw
8.6.15 RecoverPkaClearKeyTokenUnderXport
8.6.16 RecoverPkaClearKeyTokenUnderMKWithMKWithMinModulus
8.6.17 ReEncipherPkaKeyTokenWithMK - re-encipher PKA key token

8.7 Functions to use the RSA keys

8.7.1 generate_dSig - generate digital signature
8.7.2 RequestRSACrypto - perform an RSA operation
8.7.3 verify_dSig - verify digital signature

8.8 Functions to handle the on-board key storage

8.8.1 delete_KeyToken - delete a key from on-board storage
8.8.2 getKeyToken - get a PKA token from on-board storage
8.8.3 store_KeyToken - store registered or retained key

9 CCA SRDI manager functions

9.1 Header files for SRDI manager functions
9.2 OverView
9.3 CCA SRDI manager operation
9.3.1 An example: opening a SRDI
9.4 Using the BBRAM without the SRDI manager
9.5 Using the flash storage
9.6 Summary of functions

9.6.1 close_cca_srdi - close CCA SRDI
9.6.2 create_cca_srdi - create CCA SRDI
9.6.3 delete_cca_srdi - delete CCA SRDI
9.6.4 get_cca_srdi_length - get CCA SRDI length
9.6.5 open_cca_srdi - open CCA SRDI
9.6.6 resize_cca_srdi - resize CCA SRDI
9.6.7 save_cca_srdi - save CCA SRDI
9.6.8 Example SRDI code

10 Access control manager functions
10.1 Header files for access control manager functions
10.2 Summary of functions
10.2.1 SRDI files for the access control manager
10.3 Functions for general use with the access control manager
10.3.1 ac_get_list_sizes - get sizes of role and profile lists
10.3.2 ac_init - initialize the access control manager
10.3.3 ac_reinit - reinitialize the access control manager
10.3.4 acm_xlt_errorcode - translate an ACM error code to a SAPI error code
10.4 Functions to manage roles.................................................................249
  10.4.1 ac_del_role - delete role.........................................................250
  10.4.2 ac_get_role - get role............................................................251
  10.4.3 ac_list_roles - list roles.......................................................252
  10.4.4 ac_load_roles - load roles....................................................253
  10.4.5 ac_query_role - return the length of a role............................255
10.5 Functions to manage profiles.......................................................256
  10.5.1 ac_chg_prof_auth_data - change profile authentication data........257
  10.5.2 ac_chg_prof_exp_date - change profile expiration date..............259
  10.5.3 ac_del_profile - delete user profile.......................................260
  10.5.4 ac_get_profile - get profile..................................................261
  10.5.5 ac_list_profiles - list user profiles.......................................262
  10.5.6 ac_load_profiles - load user profiles.....................................263
  10.5.7 ac_query_profile - return the length of a user profile...............265
  10.5.8 ac_reset_logon_fail_cnt - reset logon failure count..................266
  10.5.9 update_logon_failure_count - adjust count of logon failures.......267
10.6 Functions to manage logged on users.............................................268
  10.6.1 ac_check_authorization - check authorization to execute function....269
  10.6.2 ac_lu_add_user - add a user to the list of logged on users............270
  10.6.3 ac_lu_drop_user - remove a user from the logged on list..............271
  10.6.4 ac_lu_get_ks - get a copy of a session key..............................272
  10.6.5 ac_lu_get_num_users - get the number of logged on users............273
  10.6.6 ac_lu_get_role - get role from the logon list.........................274
  10.6.7 ac_lu_ks_dec - decrypt data with session key..........................275
  10.6.8 ac_lu_ks_hmacgen - Compute an HMAC Using Session Key................276
  10.6.9 ac_lu_ks_hmacver - verify an HMAC using session key................277
  10.6.10 ac_lu_list_users - list the IDs of the logged on users...............278
  10.6.11 ac_lu_query_user - check if a user is logged on......................279
11 Cache management functions.........................................................280
  11.1 Header files for caching functions..........................................280
  11.2 Overview of cache management functions....................................280
  11.3 Summary of functions................................................................280
     11.3.1 cache_clear - remove all data from cache..............................282
     11.3.2 cache_delete - delete cache..............................................283
     11.3.3 cache_delete_item - delete one item from cache......................284
     11.3.4 cache_get_item - get an item from cache..............................285
     11.3.5 cache_get_item_b - get an item in a new buffer......................286
     11.3.6 cache_init - initialize a cache..........................................287
     11.3.7 cache_status - determine cache status................................288
     11.3.8 cache_write_item - add an item to cache..............................289
12 Miscellaneous functions .............................................................290
  12.1 Header files for miscellaneous functions.....................................290
  12.2 Summary of functions................................................................290
     12.2.1 CasGetFileDescriptor - Deprecated....................................292
     12.2.2 check_access_auth_fcn - verify user authority.......................293
     12.2.3 GetAnyKeyLength - get length of key token..........................294
     12.2.4 GetRandomNumber - generate random number.........................295
     12.2.5 intel_long_reverse - convert 4-byte values...........................296
     12.2.6 intel_word_reverse - convert 2-Byte values...........................297
     12.2.7 LIBR_AllocStorage - allocate memory....................................298
     12.2.8 LIBR_FreeStorage - free memory........................................299
     12.2.9 LIBR_ReleaseSemaphore.....................................................300
     12.2.10 LIBR_RequestSemaphore...................................................301
     12.2.11 SWAP64 - byte-reverse a 64-byte integer..............................302
     12.2.12 TOKEN_IS_A_LABEL - identifies the token as a label..............303
     12.2.13 TOKEN_LABEL_CHECK - determine if key identifier is a label.....304
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.2.14 xcSetClock – Set the coprocessor clock</td>
<td>305</td>
</tr>
<tr>
<td>13 UDX sample code</td>
<td>306</td>
</tr>
<tr>
<td>14 Reserved values</td>
<td>308</td>
</tr>
<tr>
<td>15 Data structures</td>
<td>309</td>
</tr>
<tr>
<td>15.1 Structures used in communications between the workstation host and the coprocessor</td>
<td>309</td>
</tr>
<tr>
<td>15.2 Structures used to manage keys and key tokens</td>
<td>319</td>
</tr>
<tr>
<td>15.3 Structures used for handling the master keys</td>
<td>323</td>
</tr>
<tr>
<td>15.4 Structures used in cache handling</td>
<td>324</td>
</tr>
<tr>
<td>15.5 Constants used in key parsing and building</td>
<td>325</td>
</tr>
<tr>
<td>16 Modifying a 4765 UDX for the 4767</td>
<td>330</td>
</tr>
<tr>
<td>16.1 Makefile changes</td>
<td>330</td>
</tr>
<tr>
<td>16.2 Changes to function calls</td>
<td>330</td>
</tr>
<tr>
<td>17 Notices</td>
<td>331</td>
</tr>
<tr>
<td>17.1 Copying and distributing softcopy files</td>
<td>331</td>
</tr>
<tr>
<td>17.2 Trademarks</td>
<td>331</td>
</tr>
<tr>
<td>18 List of abbreviations and acronyms</td>
<td>333</td>
</tr>
<tr>
<td>19 Glossary</td>
<td>335</td>
</tr>
<tr>
<td>20 Index</td>
<td>339</td>
</tr>
</tbody>
</table>
Figures

Figure 1 View of workstation CCA with user-defined extensions ......................................................... 2
Figure 2 View of CCA with user-defined extensions on coprocessor .................................................. 4
Figure 3 Request and reply parameter block formats ........................................................................... 7
Figure 4 Example UDX access control points ....................................................................................... 18
Figure 5 Master Key Status Bits ........................................................................................................... 98
Figure 6 Master SRDI manager overview ............................................................................................. 228
Figure 7 Master SRDI read illustration part 1 ....................................................................................... 229
Figure 8 Master SRDI read illustration part 2 ....................................................................................... 229
Figure 9 Master SRDI read illustration part 3 - controlling concurrent access to a SRDI............... 230
About this manual
The IBM 4767 PCIe Cryptographic Coprocessor CCA User Defined Extensions Reference and Guide describes the Common Cryptographic Architecture (CCA) application programming interface (API) function calls that are available to user-defined extensions to CCA. A user-defined extension (UDX) allows a developer to add customized operations to IBM's CCA Support Program. UDXs are written and invoked in the same manner as base CCA functions and have access to the same internal functions and services as the CCA Support Program.

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

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

Most of the functions are in the first class.

This manual is intended for developers who need to write a UDX. This manual should be used in conjunction with the manuals listed under “Related publications” below.

Prerequisite knowledge
The reader of this manual should understand how to perform basic tasks (including editing, system configuration, file system navigation, and creating application programs) on the host machine and in the Linux environment, and should understand the use of IBM's CCA Support Program (as described in the IBM 4767 PCIe Cryptographic Coprocessor CCA Support Program Installation Manual and the CCA Basic Services Reference and Guide). Familiarity with the IBM 4767 application development process (as described in the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Developer's Toolkit Guide) is also required. Familiarity with the functions available in the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference will also be helpful.

Organization of this document
This document is organized as follows:

“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.

“Building a CCA user-defined extension” discusses how to build each portion of a UDX.

“4767 functions” summarizes the IBM 4767 cryptographic coprocessor API on top of which IBM's CCA coprocessor application modules are built. A UDX may use this API if so desired.

“Communication 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.

“FCV management functions” describes the functions that allow a UDX to determine which cryptographic operations have been authorized by the CCA function control vector (FCV) and how long certain cryptographic keys may be.

“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) or Advanced Encryption Standard (AES) algorithms.

“Hashing 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, SHA-224, SHA-256, SHA-384, or SHA-512).
“Symmetric Key Utility Functions” describes the functions that a UDX can use to manipulate and obtain information about key tokens and other cryptographic structures.

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

“CCA SRDI Manager Functions” describes the functions that an application can use to store and retrieve data in the coprocessor's nonvolatile memory areas (BBRAM).

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

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

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

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

“UDX Sample Code - Host Piece” contains a sample of the workstation code piece.

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

“Data Structures” contains explanations of useful data structures from the Toolkit header files.

“Notices” includes product and publication notices.

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

**Typographic conventions**

This publication uses the following typographic conventions:

- Commands that you enter verbatim onto the command line are presented in monospace type.
- Variable information and parameters, such as file names, are presented in italic type.
- Constants are presented in bold type.
- The names of items that are displayed in graphical user interface (GUI) applications--such as pull-down menus, check boxes, radio buttons, and fields--are presented in bold type.
- Items displayed within pull-down menus are presented in bold italic type.
- Function names are presented in italic type.
- System responses in a shell-based environment are presented in monospace type.
- Web addresses and directory paths are presented in italic type.

For readability in information that applies to both Linux® and Windows®, this publication uses Linux naming conventions instead of showing both Linux and Windows conventions. For example, `cctk/<version>/samples` would be `cctk\<version>\samples` on Windows.

**Syntax diagrams**

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

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

A value for `firstarg` must be specified. `secondarg` may be omitted. Either `a` or `b` must be specified.
Related publications
Publications about IBM’s family of cryptographic coprocessors are available at: www.ibm.com/security/cryptocards.

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

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

Summary of changes
This edition of the IBM 4767 PCIe Cryptographic Coprocessor CCA User Defined Extensions Reference and Guide contains product information that is current with the IBM 4767 PCIe Cryptographic Coprocessor announcements.
1 Understanding the UDX environment

The IBM 4767 Developer’s Toolkit provides scaffold code, object modules, header files, and macros that you can use to extend the IBM-developed Common Cryptographic Architecture (CCA) application program which employs the IBM 4767 PCIe Cryptographic Coprocessor. You can use as much or as little of the CCA application function as required to meet your processing requirements.

You can write a user-defined extension (UDX) to be installed on an IBM AIX®, Linux®, or Windows® workstation.

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

This manual also assumes that you are familiar with the techniques for creating and testing coprocessor application programs as described in the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide. This manual describes the IBM 4767 API on which the CCA application is built.

You may also benefit from understanding the services that you can obtain from the Linux operating system on the IBM 4767.

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

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

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

The CCA verbs (callable services) that a host application can request are generally serviced, on a one-for-one basis, by a command processor portion of coprocessor application code. A common infrastructure is employed to format a verb request, transport the request to the coprocessor, dispatch the command processor, and return the reply to the host. Command processors and the top layer of CCA host code (security application program interface (SAPI)) make extensive use of a set of common subroutines described in this manual.

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

1.1 Structure of a UDX on a workstation

Figure 1 depicts the major elements of code that form the CCA workstation implementation for the UDX test environment. The boxes with dotted lines designate the UDX components. Each block represents a

---

1 A few CCA verbs are implemented as subroutines in the top layer of CCA host code and do not send a request to the coprocessor.
section of the runtime code. Blocks 1 through 5 are host-system shared libraries with block 5 actually split between a shared object and the physical device driver. An overview of these code blocks follows.

![Diagram of workstation CCA with user-defined extensions](image)

**Figure 1** View of workstation CCA with user-defined extensions

The Security API (SAPI), Security Server (SECY), and Key Store code are all contained in the shared object *libcsulcca.so* (*libcsufcca.a* on AIX).

1. **Security API (SAPI)**

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

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

   The fixed-length CPRB structure carries a primary function code, return and reason code values, and pointers to, and lengths of, the request and reply parameter blocks and data to be read to/from the

---

2. **Typical CCA host code file names begin with csuX where the “X” is “f” for AIX or “l” for Linux.**

2. **IBM 4767 CCA UDX Reference**
coprocessor. The variable-length request and reply parameter blocks carry this information. See Figure 3 on page 7 for details.

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

The subroutines used to construct and to parse these control blocks are used by all of the verb routines in SAPI. These same subroutines are entry points that can be called by the UDX-SAPI code. See “Communication functions” on page 33 for more information.

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

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

**UDX-SAPI**

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

Once the CPRB and request parameter block are constructed, you use the `CSNC_SP_SCSRFBSS()` subroutine to pass control to the SECY layer. Upon regaining control, your code should update the caller’s variables with the information that is parsed from the CPRB and reply parameter block. See the UDX sample in the Toolkit for an example. This sample is described in “UDX sample code” on page 306.

**SECY**

SECY, which is also in the `libcsulcca.so` shared object (`libcsufcca.a` on AIX), receives control from SAPI with a pointer to the CPRB. CCA examines the key-block fields of the request and reply parameter blocks to determine if the key storage server should be called to allocate, delete, or list labels in key storage, or to fetch or store key records under key labels already existing in key storage. The security server also passes the name of the key storage files to the directory server. On input, except for a few key-storage services which do not require use of the coprocessor, the security server calls the adapter interface after completing any required key storage actions. Likewise, on output the adapter interface returns control to the security server which completes any required key storage requests and then returns control to SAPI. Information in a key block header (see “Key blocks” on page 10) triggers the security server to process a key block.

**Key storage server**

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

Understanding the UDX environment
Device driver and access layer
The device driver code is split between a shared object and a physical device driver. The API and function of the device driver is explained in the *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference*. The device driver and Linux operating system work together to ensure that the reply to a request is routed back to the source process and thread that initiated the associated request.

1.2 Structure of a UDX on the coprocessor
Figure 2 depicts the major elements of code that form the CCA with a UDX implementation on the 4767 cryptographic coprocessor.

Linux operating system services
The *init.sh* shell file is run automatically in Segment 3 following a reset sequence (which also runs the *init.sh* file in Segment 2). The *csulcca* application’s entry point is called by the *startcdud* function and CCA registers itself with the IBM 4767 Communications Manager.

When the IBM 4767 Communications Manager receives a request from the host it checks for a registered application identifier; the identifier is a constant prearranged between the adapter interface layer and the CCA application. CCA host requests include the CPRB and request parameter block. The application interface layer presents sufficient information that the CCA Dispatcher can request the Communications Manager to obtain the CPRB and request parameter block.

Other low-level services for date and time, storage of data in BBRAM, and communication with external functions (particularly through the serial port) are available through Linux (C) functions to the UDX code.

4 IBM 4767 CCA UDX Reference
CCA dispatcher
When the Communications Manager responds to the CCA dispatcher’s request for input because of the receipt of a host request, the dispatcher obtains the CPRB and request parameter block. The dispatcher also locates the role that governs the processing of the CCA request, either the default role or the role associated with a logged-on profile.

Each thread of each workstation host process can logon to a role through an associated profile. However, a single profile can be associated with only one host thread at a time; a correct logon to a profile from another thread will be honored and a new session key generated without any indication of this action reported to the “older” logged-on thread (until and if the older thread makes a new request).

The dispatcher uses the sub-function code in the first two bytes of the request parameter block in a table lookup operation to locate a UDX command processor entry point. If a match is not found, the dispatcher checks the CCA entry point table for a match. (Of course, if again no match is found, the dispatcher constructs a reply CRPB and fills it with a return and reason code indicating that no such function exists.)

The dispatcher then creates an Operation Tracking Structure (OPTX) for the request. The OPTX contains the role ID and Domain, pointers to the CPRB and request parameter block in read-only memory, and working space (in read-write memory) for keys and output data for the reply data buffer. The dispatcher calls the command processor and passes the OPTX structure.

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

The CCA implementation is single-threaded. The hardware (chips) on the coprocessor operate in asymmetric fashion – a call to the hardware will return immediately, though the request has not yet been processed. Each individual OPTX structure represents a single request, referred to as a “fiber”. The CCA thread processes the fiber to the point of a hardware call, then places the OPTX structure into a queue to await completion of the hardware function. The CCA thread then moves on to another OPTX structure which is ready for more processing, and so on. This design permits overlapped operation by the CPU, the modular-exponentiation hardware, the DES/TDES/SHA-1 hardware, the random number (bit) generator, and the external communication hardware.

The main() function in the cammgr.c module receives control when the Linux OS loads the CCA application. The main() function calls various initialization functions in the other managers (for example, to obtain the master keys and role security relevant data items (SRDI), and so on), then calls any defined UDX initialization functions. UDX initialization functions are defined by placing them in the variable ccax_init_list and defining the ccax_init_list_size variable to indicate the number of initialization functions to call.

After the initialization, twenty-four OPTX structures are created and placed on the qWOnHost queue.

When a request is received, the working thread removes the request from the queue, determines which profile and role applies, fills an empty OPTX structure, examines the sub-function code, and calls an appropriate CCA or UDX command processor. If the call is one that requires isolated processing, it is then placed on the qWonQuiesce queue to await the completion of the existing requests (a UDX may be defined to run in isolated style by a definition in the array of UDX command processors). When the command processor requires hardware service, the thread calls the libxccmn.so library and places the OPTX structure on the qWonAsync queue. After placing an OPTX on a given queue, the thread calls xcAsyncWorkCheck to determine whether there is an OPTX on the qWonAsync queue to be processed. If there is none, the qWonQuiesce queue is checked, and if it is empty, the xcGetRequest function is called to fill another OPTX structure and restart the process. When a command processor completes, the code passes the reply to the Communication Manager and continues.

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

- Declare Fiber Mutex variables as static or global variables, they must not be declared on the...
- Use RUN-ISOLATED UDXes for command processes that modify global data (for example, the master key or ACP SRDIs).

### 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:

<table>
<thead>
<tr>
<th>Manager</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SRDI Manager</strong></td>
<td>The CCA coprocessor application code generally uses the SRDI Manager to access information that is held in persistent memory. The manager is responsible for serializing the use of the SRDIs to accommodate the multitasking environment. See “CCA SRDI manager functions” on page 227 for details.</td>
</tr>
<tr>
<td><strong>Access Control Manager</strong></td>
<td>All operations on roles and profiles are carried out by the Access Control Manager. Command processors call the manager to determine if individual control points are authorized. When a command processor is designed, one or more control points are assigned, as required for security purposes, to authorize function within the command processor. See “Access control manager functions” on page 241. Also, “Reserved values” on page 308 documents a range of control points (and also reason codes and subfunction codes) reserved for UDX developers.</td>
</tr>
<tr>
<td><strong>Master Key Manager</strong></td>
<td>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 “Master key manager functions” on page 73 for details.</td>
</tr>
</tbody>
</table>

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

### CCA command processors

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

Command processor code can call any of the other CCA subroutines and manager functions as well as functions available on the Linux operating system. In general, a command processor will perform the following steps. See “UDX sample code” on page 306 for more information.

- Set the return code and reason code to 0, 0 using `Cas_proc_retc()`.
- Call the Access Control Manager to determine if the appropriate control point is authorized using `CHECK_ACCESS_AUTH()`.
- Set up the operation tracking token structures by calling `TOK_ACCESS_AND_INITPARSE()`.
- Check that the request parameter block is formed in a valid manner by calling `jparm_block_valid()`.
- Check the length of the rule array data area by examining the rule array area length bytes. For CCA, this value is $8x+2$ where $x=0, 1, ..., n$; however, you could make this portion of the request parameter block contain data of almost any length. You can check the rule array elements using...
Check the length of any VUD, data formatted to the needs of the command processor.

- Validate and decrypt the keys using the `parse_keys()` function.
- Perform the desired command function. Use the CALLXC macros to call any required hardware function. This will handle the enqueueing required.
- Determine that the reply will not exceed the permissible reply size.
- Fill in the reply block with the rule array length and any elements, fill in the VUD length and any data, and fill in the key-block area length and any key blocks.
- Return to the dispatcher.

**UDX command processors**

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

### 1.3 CCA communication structures

Four 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
- Request and reply data blocks
- Operation Tracking Structure

CCA key tokens and access control structures are described in Appendix B of the *IBM CCA Basic Services Reference and Guide*.

#### 1.3.1 Request and reply parameter block format

The request and reply parameter blocks immediately follow a data structure of type `CPRB_structure`. Here is the request and reply parameter block format.

**Note:** The host-side code processes the lengths in big-endian format.

<table>
<thead>
<tr>
<th>Field</th>
<th>Sub-FunctionCode</th>
<th>Rule Array Count</th>
<th>Rule Array</th>
<th>Verb Unique Data Length</th>
<th>Verb Unique Data</th>
<th>Key Block Length</th>
<th>Key Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>2</td>
<td>2</td>
<td>x</td>
<td>2</td>
<td>y</td>
<td>2</td>
<td>z</td>
</tr>
<tr>
<td>Offset</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>4+x</td>
<td>6+x</td>
<td>6+x+y</td>
<td>8+x+y</td>
</tr>
</tbody>
</table>

Figure 3 Request and reply parameter block formats
### 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 CCA-provided utility routines. BuildParmBlock is the routine available on the coprocessor as well as the workstation host. The developer calls BuildParmBlock three times to build a request/reply parameter block: once for rule information, once for the verb unique data, and once for the key data. The order is important: rules first, then verb unique data, followed by key data. This routine simplifies request/reply parameter block creation by accepting an arbitrary number of argument pairs (length + data pointer pairs) and constructs the sub-blocks in the previous table.

Similarly, while it is possible to extract data from the request/reply parameter blocks “by hand” using pointer arithmetic, it is recommended that the UDX developer instead use the CCA-provided utility routines FindFirstDataBlock, FindNextDataBlock, find_first_key_block, and find_next_key_block or the corresponding jFindFirstDataBlock, jFindNextDataBlock, jfind_first_key_block and jfind_next_key_block.

**Note:** An example of the use of these functions (BuildParmBlock and CSUC_BULDCPRB) is in the UDX sample in the Toolkit. This sample is described in “UDX sample code” on page 306.

#### 1.3.2 Request and reply data blocks

If more data must be passed, it is possible to pass the host address to the coprocessor for reading or writing with the CSUC_BULDCPRB command on the workstation. The buffer so addressed for sending to the coprocessor is referred to as a request data block. The length and pointer for the reply data block can be used for reading data from the coprocessor. The data buffers must not overlap and must be a multiple of eight bytes long.
// First, set the CPRB structures properly, with Rule Array, // Verb Unique Data, and Key Blocks. // To set the Request Data Block: 
LocalRequestTextLength = *pTextLength; 
LocalReplyTextLength = *pTextLength; 
CSUC_BULDCPRB( pCprb, 
(UCHAR *) ESSS_FUNCTION_ID_S, 
RequestBlockLength, // Req.Parm block len + adr 
pRequestParmBlock, 
LocalRequestTextLength, // Req.Data block len + adr 
(UCHAR *) pInpText, 
sizeof( pRequestReplyBuffer->reply_buf ), 
pRequestReplyBuffer->reply_buf, 
LocalReplyTextLength, // Reply parm 
(UCHAR *) pOutText);

On the coprocessor:

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

BulkBlockLength = pRequestCprb->req_data_block_length; 

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

// Use the space defined in the Operation Tracking // Structure for your output text. It has been // created for you with the maximum size 

OutTxt = pOptx->outData; 

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

// Write the Length of OutTxt in the CPRB 

pReplyCprb->replied_data_block_length = BulkBlockLength;

// Write the address of OutTxt in the CPRB 

pReplyCprb->reply_data_block_addr = OutTxt;

// Then return to the host function. 

return;

On the host workstation:

Understanding the UDX environment 9
1.3.3 Key blocks
The key blocks portion of the request and reply parameter blocks is used to transport zero or more key identifiers: key labels and/or key tokens. A key block is a data structure consisting of a header and appended key label and/or key token data.

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

<table>
<thead>
<tr>
<th>Flags indicating the type of key (one required)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKA96_TYPE</td>
</tr>
<tr>
<td>DES96_TYPE</td>
</tr>
<tr>
<td>AES_TYPE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flags indicating the action to be taken (one required)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION_READ</td>
</tr>
<tr>
<td>ACTION_WRITE</td>
</tr>
<tr>
<td>ACTION_NOOP</td>
</tr>
</tbody>
</table>

1.3.4 Operation tracking structure
The operation tracking structure is the single parameter provided to each command processor by the CCA dispatcher. The dispatcher creates this structure from the input CPRB structure, request parameter block, and request data block provided by the host application. Most of this structure and the data to which it points are read-only (owned by the coprocessor’s device driver). Exceptions to this rule are noted below. The following fields are set prior to input to the command processor (CCA or UDX):

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.t2.in_CPRB</td>
<td>A pointer to the input CPRB structure provided by the host application.</td>
</tr>
<tr>
<td>c.t2.outCPRB</td>
<td>A pointer to the output CPRB structure which will be returned to the host application when the command processor completes. This area is writeable by the application (for instance, to set the error codes or the length of returned data).</td>
</tr>
<tr>
<td>subfnid</td>
<td>The subfunction code from the request parameter block.</td>
</tr>
</tbody>
</table>
The operation tracking structure also contains the following fields which may be used by the application.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>optok_cnt</td>
<td>The number of keys the parser should attempt to find (from 1 through 11, inclusive).</td>
</tr>
<tr>
<td>optok[]</td>
<td>An array of clear keys in optx_tok_t format. The application sets types expected and the parsing required for each key in the key block, in order. The parse_keys() function will locate the encrypted keys and process them as requested, returning the clear keys, control vectors, kuf fields in this optok[] array.</td>
</tr>
<tr>
<td>optokbld[]</td>
<td>An array of up to 3 keys in optx_tokbld_t format. The application sets algorithm, type, clear keys and control vectors or kuf fields and encryption required. The otk_build_tokens() function will then format the data into an encrypted token in the requested format.</td>
</tr>
</tbody>
</table>

The operation tracking optok[] array of optx_tok_t structures contains the following input key fields. All of these fields are writeable.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tok_type_exp</td>
<td>An XOR of the key types (for example, NULL token, Label, External key, Internal key, and so forth), which may be expected in this position. Constants will begin with TOKTE_.</td>
</tr>
<tr>
<td>tok_alg_exp</td>
<td>An XOR of valid algorithms (for example, DES, AES, RSA) for the key in this position. Constants will begin with TOKAE_.</td>
</tr>
<tr>
<td>in_flags</td>
<td>An XOR of the flags to set for a key in this position, which informs the parser to decrypt with a KEK, or that this is a KEK, for example. Constants will begin with TOKI_.</td>
</tr>
<tr>
<td>algflags</td>
<td>An XOR of the flags to set that are algorithm-specific. Constants will begin with ALGI_.</td>
</tr>
<tr>
<td>algflags2</td>
<td>An XOR of other flags to set. Constants will begin with ALGI2_.</td>
</tr>
</tbody>
</table>

Understanding the UDX environment
usage | An XOR of possible usages the parser should check for the key in this position. Constants will begin with TOKU_.

The operation tracking optok array of optx_tok_t structures contains the following output key fields. These fields are not writeable by the user except in the circumstances listed in the table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tok_alg</td>
<td>The actual key type found by the parser.</td>
</tr>
<tr>
<td>tok_type</td>
<td>The actual algorithm for the key found by the parser.</td>
</tr>
<tr>
<td>outflags</td>
<td>Indicators of items found by the parser, for example, key was encrypted with Old Master Key, key was encrypted with a KEK, an RSA key had no private key section, and others.</td>
</tr>
<tr>
<td>error</td>
<td>The return code / reason code pair that was returned for this particular key. This value is also set in the output CPRB (c.t2.outCprb-&gt;secy_return_code).</td>
</tr>
<tr>
<td>tok_len</td>
<td>Length of the key token (unparsed).</td>
</tr>
<tr>
<td>tok</td>
<td>A pointer to the (unencrypted) key token in the key block.</td>
</tr>
<tr>
<td>ckey_bit_len</td>
<td>For variable length tokens, the bit length of the clear key.</td>
</tr>
<tr>
<td>ckey_len</td>
<td>The byte length of the butter pointed to by ckey.</td>
</tr>
<tr>
<td>ckey</td>
<td>A pointer to the “main” key. This is the decrypted key for DES and AES, or the RsakeyTokenHeader with the appropriate clear sections for RSA. The key may be an encrypted key if decryption flags were not set, for example.</td>
</tr>
<tr>
<td>cv_len</td>
<td>The byte length of the buffer pointed to by cv.</td>
</tr>
<tr>
<td>cv</td>
<td>A pointer to the control vector(s). If there are multiple control vectors such as left and right, they are concatenated.</td>
</tr>
<tr>
<td>cv2_len</td>
<td>The byte length of the buffer pointed to by cv2.</td>
</tr>
<tr>
<td>cv2</td>
<td>A pointer to the KMF fields, if any.</td>
</tr>
<tr>
<td>tok_alg_sec_len</td>
<td>The byte length of the buffer pointed to by tok_alg_sec.</td>
</tr>
<tr>
<td>tok_alg_sec</td>
<td>A pointer to the tokenmarks field in the input DES token, or the clear private key section of an RSA token, or the key label in the associated data section of a vartoken (AES or HMAC).</td>
</tr>
<tr>
<td>tok_alg_sec2_len</td>
<td>The byte length of the buffer pointed to by tok_alg_sec2.</td>
</tr>
<tr>
<td>tok_alg_sec2</td>
<td>A pointer to the public key section of an RSA token or the IBM extended associated data section of a vartoken (AES or HMAC).</td>
</tr>
<tr>
<td>tok_alg_val</td>
<td>The bit length of the modulus of an RSA key.</td>
</tr>
<tr>
<td>ckey_opt_len</td>
<td>The byte length of the buffer pointed to by ckey_opt.</td>
</tr>
<tr>
<td>ckey_opt</td>
<td>A pointer to the beginning of the payload in the vartoken in the key block.</td>
</tr>
</tbody>
</table>
The operation tracking structure contains the following `optx_tokbld_t` fields. All of these fields are writeable.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pSRCtok</td>
<td>A pointer to the <code>optx_tok_t</code> structure that has the clear key. This is optional, and may be NULL if you are not basing the output key on the input key token. If this field is NULL, you MUST set <code>tok_type_out</code> and <code>tok_alg_out</code>, <code>kt.des</code> or <code>kt.aes</code>, or <code>kt.var</code>, <code>cv_len</code> and <code>cv</code>, the <code>tok_alg_sec</code> fields and the length which apply to this type of key.</td>
</tr>
<tr>
<td>pKEKtok</td>
<td>A pointer to the <code>optx_tok_t</code> structure which has the Key Encrypting Key to encrypt this key. This is required if the output key is to be an external key; otherwise it must be set to NULL.</td>
</tr>
<tr>
<td>bldiflags</td>
<td>Input flags for the builder. Largely for a UDX, this would be used to set the XLAT CNTRL bit on or to build a public key only RSA token. Otherwise, these flags are used to match legacy CCA processing and are not needed for UDXes.</td>
</tr>
<tr>
<td>wt.ccawrap</td>
<td>Set this flag to indicate the type of wrapping.</td>
</tr>
<tr>
<td>ckey_bit_len</td>
<td>If present, this will override the value in <code>pSRCtok-&gt;ckey_bit_len</code>.</td>
</tr>
<tr>
<td>ckey_len</td>
<td>If present, this will override the value in <code>pSRCtok-&gt;ckey_len</code>.</td>
</tr>
<tr>
<td>ckey</td>
<td>If present, this will override the value in <code>pSRCtok-&gt;ckey</code>.</td>
</tr>
<tr>
<td>cv_len</td>
<td>If present, this will override the value in <code>pSRCtok-&gt;cv_len</code>.</td>
</tr>
<tr>
<td>cv</td>
<td>If present, this will override the value in <code>pSRCtok-&gt;cv</code>.</td>
</tr>
<tr>
<td>cv2_len</td>
<td>If present, this will override the value in <code>pSRCtok-&gt;cv2_len</code>.</td>
</tr>
<tr>
<td>cv2</td>
<td>If present, this will override the value in <code>pSRCtok-&gt;cv2</code>.</td>
</tr>
<tr>
<td>tok_type_out</td>
<td>If <code>pSRCtok</code> is non-NULL, this value must match the value in <code>pSRCtok-&gt;tok_type_out</code>.</td>
</tr>
<tr>
<td>tok_alg_out</td>
<td>If <code>pSRCtok</code> is non-NULL, this value must match the value in <code>pSRCtok-&gt;tok_alg_out</code>.</td>
</tr>
<tr>
<td>tok_out_len</td>
<td>Length in bytes of the buffer supplied in which to build the token. If <code>tok_out</code> is NULL on input, this value is ignored (on input) and the allocated length will be returned in this variable.</td>
</tr>
<tr>
<td>tok_out</td>
<td>If this value is NULL, space will be allocated in which to build the token, an address to that space will be passed back in this field, and the <code>tok_out_len</code> will be set appropriately.</td>
</tr>
</tbody>
</table>
2 Building a CCA user-defined extension

This chapter describes the process you can follow in creating a User-Defined Extension (UDX) for the CCA application that performs within and accesses the coprocessor. The chapter begins with an explanation of the files that you will use and then continues with the steps that you can follow in developing the host and the coprocessor pieces of code. It is assumed that you are familiar with developing and testing applications for the coprocessor, and that you have knowledge of the 4767 API as explained in the other Toolkit publications (see "Related publications" on page 12).

2.1 Files you use in building a UDX

A developer will need the following files, which are not provided with the Toolkit, to produce a UDX:

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

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

- A header file (for example, `csulextn.h`) that defines the interface the UDX exports to a user's application. This header file is included by the user's application and by the workstation host piece of the UDX and should contain a function prototype for each service the UDX provides. Such services are implemented in the same manner as CCA verbs; an example appears in Workstation piece of a UDX on page 16.
- A header file (for example, `cxt_cmds.h`) that defines the interface between the workstation host piece of the UDX and the coprocessor piece of the UDX. This header file is included by the workstation host piece and the coprocessor piece and should define UDX subfunction codes and the access control points and completion codes used by the UDX. The sample provided with the Toolkit includes comments that indicate the range of acceptable values for each of these elements.
- One or more C source files (for example, `tst_udx.c`) that implement the workstation host piece of the UDX. One sample file is provided with the Toolkit:

```c
...samples/udx/wks/host/tst_udx.c exports functions to the user's application. Each function checks its input parameters, constructs a request block, sends the request to the coprocessor and receives the reply, extracts the result, and returns it to the user's application.
```
- One or more C source files (for example, `card.c`) that implement the coprocessor piece of the UDX. One sample file is provided with the Toolkit:

```c
.../wks/card/card.c implements command processors. Each command processor receives a request from the host, validates the request and extracts the arguments it contains, performs a simple operation, constructs a reply block, and returns the reply to the host piece of the UDX. Each command processor must be added to the `ccax_cp_list` array, which must be defined in this or another program file which is compiled with the coprocessor piece of the UDX.
```

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

- A makefile that builds the shared object or dynamic load library that implements the workstation host piece of the UDX. The Toolkit provides `host.mak` (which creates `libcsufextn.a` for use on AIX, `libcsulextn.so` for use on Linux, or `tst_udx.dll` for use on Windows).

14 IBM 4767 CCA UDX Reference
- A makefile (for example, card.mak) that builds the executable that implements the coprocessor piece of the UDX. This library includes the object(s) for the UDX. This makefile uses the gcc cross-compiler to build code which will run on the PowerPC coprocessor. The makefile provided with the Toolkit, in conjunction with the toolkit_samples.inc include file, assumes that the environment variable CCTK_FS_ROOT points to the appropriate base of the Toolkit directory structure (for example, /opt/cctk/<version>). It also assumes that the environment variable CROSS points to the root of the tree where the cross-compiler for ppc is stored on your system.

- A shell script (for example, cctk.wks.debug.init.sh) that will run on the coprocessor to start the required processes. This file must be renamed init.sh and copied into the /flashS3 directory of the filesystem which is ported to the coprocessor. The Toolkit provides a debug shell to set up a daemon on the coprocessor to run the ICAT debugger, and a production shell to set up a production version of the UDX, without the debugger.

- A makefile (for example, cctk.set3.image.mak) that incorporates the required binaries and shell files into a file structure and formats them for incorporation onto the read-only disk using the mkfs.jffs2 call. The provided makefile requires that the user have sudo chown permissions, and that the environment variable CCTK_JFFS2_DIR be set to the directory where the mkfs.jffs2 executable is found.

The following binary files are used to produce a UDX:

- A program named csulccaW, which contains the coprocessor CCA object modules for workstations. The toolkit provides this file in the bin/card directory.

- A library named libudx.so that contains the coprocessor UDX object modules. The csulcca program dynamically loads this object on the coprocessor. As a result, the csulcca program is a coprocessor application executable that contains all of the standard CCA functions and those functions provided by the UDX. The Toolkit provides a makefile (card.mak) for use when building a UDX to run on a workstation.

- A program named startcdud, which changes the effective user ID and group ID of the process running on the coprocessor, and copies the executable files and libraries into the appropriate directory on the coprocessor. This assures that the process does not run as root, but can still access required files (such as SRDs). The Toolkit provides this file in the bin/card directory.

- A library named libxcoa.so, which runs the Outbound Authentication Manager on the coprocessor. The Toolkit provides this file in the lib/card directory.

- A library named libxccmn.so which contains the xcrypto library interface to the coprocessor crypto chips and other common code. The Toolkit provides this file in the lib/card directory.

- Optionally, the zdaemon program, which is the coprocessor side of the ICATPZX debugger. The Toolkit provides this file in the $(ICAT_ROOT)/card directory.

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

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

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

Building a CCA user-defined extension 15
2.2 Workstation piece of a UDX

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

This section lists the steps a developer must complete in order to create the host piece of a UDX which will be run on a workstation.

1. Define the UDX API

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

Prototypes for the standard CCA API functions may serve as examples and are located in csucincl.h, which is in the Toolkit inc directory. Sample CCA command extensions are in cxt_cmds.h in that directory.

UDX prototypes may have any number of parameters, although for consistency reasons it is recommended that all UDX extension functions follow the parameter instructions in the sample CCA extension functions. Every parameter must either be a pointer to a 64-bit integer, a pointer to an array of bytes, or a pointer to an array of integers. In the case of arrays, the number of elements in the array is by convention passed in a separate parameter.

Refer to the IBM CCA Basic Services Reference and Guide for more information about parameter types.

2. Define the subfunction codes for the UDX

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

A list of the subfunction codes for the standard CCA API functions appears in cmncriyt2.h, which is part of the Toolkit.

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 the IBM CCA Basic Services Reference and Guide. If no standard code is applicable to a particular situation, new completion codes must be defined in a header file (for example, cxt_cmds.h) that is #included in both pieces of the UDX. See cxt_cmds.h for an example of such a definition.
4. **Design and code the logic of the host piece of the UDX**

The host piece of a UDX is typically straightforward - it essentially constructs a request block, sends the block to the coprocessor, and parses the result. See the UDX sample in the Toolkit for an example. This sample is described in "UDX sample code" on page 306.

In general, the host piece of a UDX should be as small as possible. Most of the work should be performed by the coprocessor piece. This approach makes it much easier to port the test code to/from different platforms, as well as making the resulting code more secure.

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

The Toolkit includes sample makefiles for gcc on Linux (host.mak). Statements should be added to compile the source files that contain the host piece of the UDX and create the UDX shared object or library file. The user's application links with the library file to resolve references to the functions the UDX exports. For further information about the build environment, including required compiler options, refer to Chapter 3 of the *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Developer's Toolkit Guide*.

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

When running a UDX application on AIX, the shared object libcsufcca.a must be available in your LD_LIBRARY_PATH. When running a UDX application on a Linux workstation, the shared object libcsulcca.so must be available in your LD_LIBRARY_PATH. This file is provided with CCA, and is typically installed in the /usr/lib64 directory. When running a UDX application on a Windows workstation, the dynamic link library csuncca.dll must be available in your PATH. This file is provided with CCA and is typically installed in the C:\Windows\System32 directory.

2.3 **Coprocessor piece of a UDX**

The coprocessor piece of a UDX is a collection of one or more command processors that is linked into a shared object (libudx.so) that IBM's CCA coprocessor application executable can load. The coprocessor piece of a UDX may invoke any of the CCA services and can also invoke Linux functions.

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

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

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

On entry to a command processor, pOptx contains the address of an operation tracking structure. The operation tracking structure contains a pointer to a copy of the CPRB which the host piece of the UDX passed to the call that caused the command processor to gain control, as well as pointers to the request parameter block and space for the data to be returned to the host, and the role identifier for the process on the host which made the call to the coprocessor.

2. **Define access control points for the UDX**

Associated with each profile on the host is a role, or set of coprocessor operations the profile is allowed to invoke. If access to the functions exported by the coprocessor piece of the UDX needs to be restricted in
any way, new access control point (ACP) values must be defined in a header file (for example, `cxt_cmds.h`) that is included by the coprocessor piece of the UDX. See `cxt_cmds.h` for an example of such a definition.

A command processor can use access control points in conjunction with the role identifier supplied as an argument to the command processor to determine whether or not a particular operation is authorized. See “check_access_auth_fcn - verify user authority” on page 293 for details.

If you are testing the access control points on a workstation, you must add these values to the `csuap.def` file in the `cnm` subdirectory of the CCA (for example, `/opt/ibm/4767/cnm`). The cnm utility uses this file to enable editing of roles. Refer to the IBM 4767 PCIe Cryptographic Coprocessor CCA Support Program Installation Manual for more information.

```c
/#************ 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 4 Example UDX access control points

3. **Add the UDX command processors 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 sub-function code, run type, and the name of the corresponding command processor. A run type of RUN_NORMAL will cause the function to run interleaved with other functions, while a run type of RUN_ISOLATED will cause the program to wait for all running fibers to finish. This is the equivalent of running as a single-threaded operation, and is recommended for routines that modify major systems on the coprocessor.

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

The file `cxt_cmds.c` contains an example of the requisite definitions.

4. **If desired, add the UDX initialization and cleanup routines to the UDX initial array**

IBM's CCA coprocessor application modules use an array to determine which UDX initialization and cleanup routines to invoke. An entry for each set of routines (initialization is paired with cleanup) must be added to the `ccax_init_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, run type, and the name of the corresponding command processor. A run type of RUN_NORMAL will cause the function to run interleaved with other functions, while a run type of RUN_ISOLATED will cause the program to wait for all running fibers to finish. This is the equivalent of running as a single-threaded operation, and is recommended for routines that modify major systems on the coprocessor.

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

See `cxt_cmds.c` for an example of the requisite definitions.
5. **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 (`card.c`) appears in “UDX sample code” on page 306.

6. **Build the UDX coprocessor shared object**

The Toolkit includes a sample makefile (`card.mak`) that works with the cross-compiler on SLES 11. Statements should be added to compile the source files that contain the coprocessor piece of the UDX. The executable file must be named `libudx.so` to work with the `csulcca` program and the `init.sh` file on the coprocessor.

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

When running a UDX application on an AIX workstation, the following shared object must be available (in your LIBPATH): `libcsufcca.a`. This file is provided by the installation of CCA.

When running a UDX application on a Linux workstation, the following shared object must be available (in your LD_LIBRARY_PATH): `libcsulcca.so`. This file is provided by the installation of CCA.

When running a UDX application on a Windows workstation, the following dynamic link library must be available (in your PATH): `csuncca.dll`. This file is provided with CCA.

8. **Modify the initialization shell script**

When the coprocessor is reset, the boot sequence includes the running of a segment 2 shell file which calls `/flashS3/init.sh` as its final command. This file must be set up to start the `startcdud` process on the coprocessor, as well as (optionally), the daemons to run the debugger (`zdaemon`) and the logger (`cryptologkd`). The Toolkit includes 2 shell files: `cctk.wks.debug.init.sh` and `cctk.wks.prod.sh`. The debug shell file includes instructions to start the debugger, `zdaemon`, while the prod shell file does not.

9. **Build the UDX image**

The Toolkit includes a sample makefile (`cctkseg3.image.mak`) in the build_seg3_image directory which will place the executables and shared objects required into the required format within the flashS3 directory. You may change compiler variables within this makefile or use command-line options to obtain a debug image or a production image. The makefile builds `cctk.wks.debug.<datetimestamp>.bin` or `cctk.wks.prod.<datetimestamp>.bin`.

10. **Load the image onto the coprocessor**

The Toolkit includes DRUID, which is used to install the image onto the coprocessor. Refer to the [IBM 4767 PCIe Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide](https://www.ibm.com/support/knowledgecenter/SSTYXW/CEC36/ref-guide) for an explanation of the DRUID utility.

To securely load your application into a coprocessor requires that the application be signed with keys certified by IBM. Refer to the [IBM 4767 PCIe Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide](https://www.ibm.com/support/knowledgecenter/SSTYXW/CEC36/ref-guide) for an explanation of the process to obtain certified keys and to sign your application.

Also, the coprocessor must have the Toolkit loaded (segment 2 owner ID 3 and segment 3 owner ID 6) before you use DRUID to install the image. Chapter 5 of the [IBM 4767 PCIe Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide](https://www.ibm.com/support/knowledgecenter/SSTYXW/CEC36/ref-guide) describes how to use CLU to get the coprocessor into the proper state.
3 4767 functions

The CCA API is built on top of the secure cryptographic coprocessor API, a lower-level API that allows the host piece of IBM's CCA Support Program to interact with the coprocessor piece of the CCA Support Program and allows the coprocessor piece of the CCA Support Program to perform various cryptographic operations and to manipulate persistent storage on the coprocessor. 4767 API functions can also be invoked by a UDX. The 4767 API includes a set of functions an application running on the host may invoke (the host-side API) and a set of functions an application running on the coprocessor may invoke (the coprocessor-side API). However, because of the fiber and queue model CCA uses, it is strongly suggested that the coprocessor-side API be accessed only through the macros described in this chapter (section 3.5.1 on page 23).

This section briefly describes the 4767 API. A more detailed description may be found in the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference.

3.1 Host-side 4767 API functions

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

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

3.2 Coprocessor-side 4767 API functions

The coprocessor API includes functions in the following categories:

<table>
<thead>
<tr>
<th>Function category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communications</td>
<td>Allows a coprocessor application to interact with a host application and obtain permission to request services from the coprocessor device managers.</td>
</tr>
<tr>
<td>Coprocessor configuration</td>
<td>Configures certain processor features or return information about the coprocessor.</td>
</tr>
<tr>
<td>DES and AES</td>
<td>Allows a coprocessor application to request services from the Secure Key Cryptographic Hash (SKCH) Manager, which uses the coprocessor's SKCH chip to support DES operations with key lengths of 56, 112, or 168 bits, or AES operations with 128, 192, or 256 bits.</td>
</tr>
<tr>
<td>Fiber</td>
<td>Allows a coprocessor application to suspend processing while waiting for a hardware operation to complete. CCA uses the fiber-management system to interleave requests, so only the SUSPEND function should be used in a UDX operation.</td>
</tr>
<tr>
<td>Fiber Mutex</td>
<td>Allows a coprocessor application to enforce mutual exclusion across fibers.</td>
</tr>
<tr>
<td>Hash</td>
<td>Allows a coprocessor application to compute a condensed representation of a block of data using various standard hash algorithms.</td>
</tr>
<tr>
<td>Large Integer Modular Math</td>
<td>Allows a coprocessor application to direct the PKA Manager to perform specific operations on large integers.</td>
</tr>
<tr>
<td>Operation management</td>
<td>Allows a coprocessor to determine whether a library call is completed or still in progress.</td>
</tr>
<tr>
<td>Public Key Algorithm</td>
<td>Allows a coprocessor application to request services from the Public Key...</td>
</tr>
<tr>
<td>Function category</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Algorithm (PKA) Manager</td>
<td>Uses the coprocessor's large-integer modular math hardware to support public key cryptographic algorithms.</td>
</tr>
<tr>
<td>Random Number Generator</td>
<td>Allows a coprocessor application to request services from the Deterministic Random Bit Generator (DRBG) Manager, which uses a hardware noise source and software to deliver random bits that meet the standards described in FIPS Publication 140-1, section 4.11.</td>
</tr>
</tbody>
</table>

### 3.3 Coprocessor-side Fiber-management macros

In a coprocessor running CCA, there is one thread which handles multiple operation tracking structures over the asynchronous hardware requests. Using the macros described in this section to call any xcrypto library function will allow your code to run safely within the fiber context of the overall CCA implementation.

The macros in this section are available only on the coprocessor.

### 3.4 Header files for Fiber macros

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

```c
#include "cmnfiber.h"
#include "xc_types.h"
#include "xc_api.h"
```

### 3.5 Summary of Macros

The “CALLXC_FIBER” macro is the basic one of this set. All of the other macros call it as their final step, and you can also use it to call any other xcrypto library routine.

<table>
<thead>
<tr>
<th>xcrypto API used</th>
<th>description</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY</td>
<td>CALLXC_FIBER</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>This macro will set the cmn field of the request block with the data required to return, then call the xcrypto library function requested, then call xcfbr_SUSPEND. Execution will resume at the point after this macro when the control function returns to this fiber after the xcrypto library function completes.</td>
<td></td>
</tr>
<tr>
<td>xcDES</td>
<td>CALLXCDES</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Fills a request block for any single DES encryption or decryption.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CALLXCDES_ZEROVEC</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Sets up zero-value init vector and term vector, then calls CALLXCDES.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CALLXCDES_8BENC</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Calls CALLXCDES_ZEROVEC for 8 bytes of data with option ENCIPHER.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CALLXCDES_8BDEC</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Calls CALLXCDES_ZEROVEC for 8 bytes of data with option DECRYPTER.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Function</td>
<td>Length</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>xcTDES</td>
<td>CALLXCTDES</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>CALLXCTDES_ZEROVEC</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>CALLXCTDES_8B_ECB_ENC</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>CALLXCTDES_8B_ECB_DEC</td>
<td>26</td>
</tr>
<tr>
<td>xcDES3Key</td>
<td>CALLXCDES3KEY</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>CALLXCDES3KEY_ECB</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>Calls CALLXCDES3KEY in a loop.</td>
<td></td>
</tr>
<tr>
<td>xcSha1</td>
<td>CALLXCSHA1</td>
<td>32</td>
</tr>
<tr>
<td>xcSha2</td>
<td>CALLXCSHA2</td>
<td>31</td>
</tr>
</tbody>
</table>
3.5.1 CALLXC_FIBER – Generic fiber calling Macro

CALLXC_FIBER sets the common parameters (the cmn structure in the request block) properly for a call to the xcrypto library, calls the requested function, and tests the return value for an immediate error. If no immediate error was found, xCFbr_SUSPEND is called, suspending this request until the hardware call completes.

Note: The remaining macros in this chapter call CALLXC_FIBER as part of their processing.

Macro format

```c
#define CALLXC_FIBER ( optx_t *pOptx,
                     int     reserved,
                     (FunctionName),
                     (RequestBlock),
                     uint32_t ReturnMsg )
```

Input

When calling this macro:

- `pOptx` is a pointer to the operation tracking structure input to this command processor.
- `reserved` is ignored.
- `FunctionName` is the name of the xcrypto library function to be called (for example, xCECC or cmackeyblockgeneratet31).
- `RequestBlock` is the request block for the xcrypto library function `FunctionName`. This request block must be initialized as described in the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference.

Output

When this macro completes:

- `ReturnMsg` has been set.

When the CCA command process determines that the xcrypto library call has completed, the fields in the request block will have been set appropriately, and execution will commence immediately following this macro. Further information is in the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference.

Return codes

Refer to `xc_err.h` for a comprehensive list of return codes. The macro does not have any specific internal codes but passes through the code from the underlying `FunctionName`.
3.5.2 CALLXCDES – Macros to fill request block and call xcDES

CALLXCDES sets the required parameters in the request block structure for a call to the xcDES, then calls CALLXC_FIBER.

CALLXCDES_ZeroVEC creates a DES vector which is set to all zeros, and uses this for both the IV and the TV for a call to CALLXCDES.

CALLXCDES_8BENC creates a DES vector which is set to all zeros, and uses this for both the IV and TV for a call to CALLXCDES using a length of 8 for both source_length and destination_length, and options of Encrypt and CBC mode.

CALLXCDES_8BDEC creates a DES vector which is set to all zeros, and uses this for both the IV and TV for a call to CALLXCDES using a length of 8 for both source_length and destination_length, and options of Decrypt and CBC mode.

**Macro format**

The following macros take these arguments:

```c
CALLXCDES ( optx_t *pOptx,
            xcDES_key_t *pKey,
            int source_length,
            unsigned char *pSource,
            int destination_length,
            unsigned char *pDestination,
            xcDES_vector_t *pIV,
            xcDES_vector_t *pTv,
            uint32_t options,
            uint32_t ReturnMsg )

CALLXCDES_ZeroVEC ( optx_t *pOptx,
                    xcDES_key_t *pKey,
                    int data_length,
                    unsigned char *pSource,
                    unsigned char *pDestination,
                    uint32_t options,
                    uint32_t ReturnMsg )

CALLXCDES_8BENC ( optx_t *pOptx,
                  xcDES_key_t *pKey,
                  unsigned char *pSource,
                  unsigned char *pDestination,
                  uint32_t ReturnMsg )

CALLXCDES_8BDEC ( optx_t *pOptx,
                 xcDES_key_t *pKey,
                 unsigned char *pSource,
                 unsigned char *pDestination,
                 uint32_t ReturnMsg )
```

**Input**

When calling one of these macros:

*pOptx* is a pointer to the operation tracking structure input to this command processor.

*pKey* is the 8-byte DES key to use for the function.

If present, *data_length* is the length of the input AND the output data. This must be a multiple of 8.
If present, `source_length` is the length of the data to encrypt or decrypt. This must be a multiple of 8.

`pSource` is the address of the data to be encrypted or decrypted.

If present, `destination_length` is the length of the buffer to hold the output. This must be at least as large as `source_length`.

`pDestination` is the address of a writeable buffer.

If present, `pIV` is a pointer to an 8 byte initialization vector.

If present, `pTv` is a pointer to an 8 byte buffer where the termination vector can be stored.

If present, `options` is the exclusive or of the options available to the xcDES function (see the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference).

**Output**

When one of these macros completes,

`ReturnMsg` has been set. The operation has been completed (either failing with an appropriate error code, or the fiber has been suspended and is now at the top of the run queue and has been restarted).

If the function was successful, when the `xcAsyncWorkCheck` returns this `pOptx` structure as completed, `Destination` contains the result of the operation.

If present, `destination_length` contains the length of the data at `Destination`.

If present, `pTV` contains the termination vector of the operation.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMGood</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>DMNoBuffer</td>
<td>The source length or destination length was too large.</td>
</tr>
<tr>
<td>DMBadParm</td>
<td>One of the parameters was invalid.</td>
</tr>
</tbody>
</table>

Refer to `xc_err.h` for a comprehensive list of return codes.
3.5.3 CALLXCTDES – Macros to fill request block and call xcTDES

CALLXCTDES sets the required parameters in the request block structure for a call to the xcTDES hardware routine with no pre- or post-padding data, then calls CALLXC_FIBER.

CALLXCTDES_ZEROVEC creates a zero-valued initialization vector and uses this for the initialization and termination vector for a call to CALLXCTDES.

CALLXCTDES_8B_ECB_ENC calls CALLXCTDESZEROVEC, using a data_length of 8 and options of encrypt and ECB mode.

CALLXCTDES_8B_ECB_DEC calls CALLXCTDESZEROVEC, using a data_length of 8 and options of decrypt and ECB mode.

Macro format

```
#define CALLXCTDES ( optx_t *pOptx,
                 xcDES_key_t *pKey1,
                 xcDES_key_t *pKey2,
                 xcDES_key_t *pKey1,
                 int source_length,
                 unsigned char *pSource,
                 int destination_length,
                 unsigned char *pDestination,
                 xcDES_vector_t *pIV,
                 xcDES_vector_t *pTv,
                 uint32_t options,
                 uint32_t ReturnMsg )
```

```
#define CALLXCTDES_ZEROVEC ( optx_t *pOptx,
                            xcDES_key_t *pKey1,
                            xcDES_key_t *pKey2,
                            xcDES_key_t *pKey3,
                            int data_length,
                            unsigned char *pSource,
                            unsigned char *pDestination,
                            uint32_t options,
                            uint32_t ReturnMsg )
```

```
#define CALLXCTDES_8B_ECB_ENC ( optx_t *pOptx,
                            unsigned char *pSource,
                            unsigned char *pDestination,
                            xcDES_key_t *pKey1,
                            xcDES_key_t *pKey2,
                            xcDES_key_t *pKey3,
                            uint32_t ReturnMsg )
```

```
#define CALLXCTDES_8B_ECB_DEC ( optx_t *pOptx,
                            unsigned char *pSource,
                            unsigned char *pDestination,
                            xcDES_key_t *pKey1,
                            xcDES_key_t *pKey2,
                            xcDES_key_t *pKey3,
                            uint32_t ReturnMsg )
```

Input

When calling one of these macros:

pOptx is a pointer to the operation tracking structure input to this command processor.

26 IBM 4767 CCA UDX Reference
pKey1 is the 8-byte DES key to use as the first key for the function.
pKey2 is the 8-byte DES key to use as the second key for the function.
pKey3 is the 8-byte DES key to use as the third key for the function.
If present, source_length is the length of the data to encrypt or decrypt. This must be a multiple of 8.
If present, data_length is the length of the data to encrypt or decrypt, and the output buffer. This must be a multiple of 8.
pSource is the address of the data to be encrypted or decrypted.
destination_length is the length of the buffer to hold the output. This must be at least as large as source_length.
pDestination is the address of a writeable buffer.
If present, pIV is a pointer to an 8 byte initialization vector.
If present, pTv is a pointer to an 8 byte buffer where the termination vector can be stored.
If present, options is the exclusive or of the options available to the xcTDES function (see the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference).

Output
When this macro completes,
ReturnMsg has been set.
If the call was successful,
Destination contains the result of the operation
If present, destination_length contains the length of the data at Destination.
If present, pTV contains the termination vector of the operation.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMGood</td>
<td>The function completed successfully</td>
</tr>
<tr>
<td>DMBadParm</td>
<td>One of the parameters was invalid</td>
</tr>
</tbody>
</table>

Refer to xc_err.h for a comprehensive list of return codes.
### 3.5.4 CALLXCDES3KEY – Macro to fill request block and call xcDES3Key

CALLXCDES3KEY sets the required parameters in the request block structure for a call to the xcDES3Key library call, then calls CALLXC_FIBER.

#### Macro format

```c
#define CALLXCDES3KEY ( optx_t *pOptx,
                        unsigned char *pSource,
                        xcDES_key_t *pKey1,
                        xcDES_key_t *pKey2,
                        xcDES_key_t *pKey3,
                        unsigned char *pDestination,
                        uint32_t options,
                        uint32_t ReturnMsg )
```

#### Input

When calling this macro:

- `pOptx` is a pointer to the operation tracking structure input to this command processor.
- `pKey1` is the 8-byte DES key to use as the first key for the function.
- `pKey2` is the 8-byte DES key to use as the second key for the function.
- `pKey3` is the 8-byte DES key to use as the third key for the function.
- `pSource` is the address of the data to be encrypted or decrypted.
- `pDestination` is the address of a writeable buffer.
- `options` is the exclusive or of the options available to the xcDES3Key function (see the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference).

#### Output

When this macro completes,

- `ReturnMsg` has been set.

If the function was successful, when the xcAsyncWorkCheck returns this `pOptx` structure as completed, `pDestination` contains the result of the operation.

#### Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMGood</td>
<td>The function completed successfully</td>
</tr>
<tr>
<td>DMBadParm</td>
<td>One of the parameters was invalid</td>
</tr>
</tbody>
</table>

Refer to `xc_err.h` for a comprehensive list of return codes.
3.5.5 CALLXDES3KEY_ECB Macro to fill request block and call xcDES3Key in a loop

CALLXDES3KEY_ECB sets the required parameters in the request block structure for a call to the xcDES3Key hardware function, then loops calls to CALLXC_FIBER to encrypt or decrypt a multiple of 8 bytes of data.

Note: if the supplied data is not a multiple of 8 bytes, the final partial block of bytes will not be encrypted or copied and no error will be returned.

Macro format

```
#define CALLXDES3KEY_ECB ( optx_t *pOptx,
                         unsigned char *pSource,
                         xcDES_key_t *pKey1,
                         xcDES_key_t *pKey2,
                         xcDES_key_t *pKey3,
                         unsigned char *pDestination,
                         int data_length,
                         uint32_t options,
                         uint32_t ReturnMsg )
```

Input

When calling this macro:

- `pOptx` is a pointer to the operation tracking structure input to this command processor.
- `pKey1` is the 8-byte DES key to use as the first key for the function.
- `pKey2` is the 8-byte DES key to use as the second key for the function.
- `pKey3` is the 8-byte DES key to use as the third key for the function.
- `pSource` is the address of the data to be encrypted or decrypted.
- `pDestination` is the address of a writeable buffer.
- `pIV` is a pointer to an 8 byte initialization vector.
- `data_length` is the length of the data to be encrypted, this must be a multiple of 8.
- `options` is the exclusive or of the options available to the xcDES3Key function (see the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference).

Output

When this macro completes,

- `ReturnMsg` has been set
- `pDestination` contains the result of the operation

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMGood</td>
<td>The function completed successfully</td>
</tr>
<tr>
<td>DMBadParm</td>
<td>One of the parameters was invalid</td>
</tr>
</tbody>
</table>
Refer to `xc_err.h` for a comprehensive list of return codes.
3.5.6 CALLXCSHA2 – Macro to fill request block and call xcSha2

CALLXCSHA2 sets the required parameters in the request block structure for a call to the xcSha2 library call, then calls CALLXC_FIBER.

Macro format

#define CALLXCSHA2 (   optx_t *pOptx,
                     xcSha2_RB_t *pSha2RB,
                     unsigned char *pSource,
                     unsigned char *pHash,
                     int source_length,
                     int hash_length,
                     uint32_t options,
                     uint32_t ReturnMsg )

Input

When calling this macro:

*pOptx* is a pointer to the operation tracking structure input to this command processor.

*pSha2RB* is a pointer to a SHA2 request block to be used for the call. If *options* indicates that this is a MIDDLE or LAST call, the *kH*, *kL*, and *magic* fields must be the same as what was returned from the previous call to xcSha2. The remainder of the fields will be set to the value from the parameter of the same name:

*pSource* is the address of the data to be hashed.

*pHash* is the address of a writeable buffer. If the options indicate this is a middle or last part of the hash, pHash must contain the chaining vector for the previous step.

source_length is the length of data to be hashed.

hash_length is the length of the buffer at pHash.

options is the exclusive or of the options available to the xcSha2 function (see the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference).

Output

When this macro completes,

ReturnMsg has been set.

*pHash* contains the result of the operation – If *options* specified an ONLY operation or a LAST operation, then this is the hash of the data.

If *options* included FIRST or MIDDLE, pSha2RB contains the required information for a chained call - the *kH* and *kL* fields and the *magic* field must be passed to the next call to CALLXCSHA2.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMGood</td>
<td>The function completed successfully</td>
</tr>
<tr>
<td>DMBadParm</td>
<td>One of the parameters was invalid</td>
</tr>
</tbody>
</table>

Refer to xc_err.h for a comprehensive list of return codes.
3.5.7 CALLXCSHA1 – Macro to fill request block and call xcSha1

CALLXCSHA1 sets the required parameters in the request block structure for a call to the xcSha1 library call, then calls CALLXC_FIBER.

Macro format

```c
#define CALLXCSHA1( optx_t *pOptx,
    xcSha1_RB_t *pSha1RB,
    unsigned char *pSource,
    unsigned char *pHash,
    int source_length,
    int hash_length,
    uint32_t options,
    uint32_t ReturnMsg)
```

Input

When calling this macro:

- `pOptx` is a pointer to the operation tracking structure input to this command processor.
- `pSha1RB` is a pointer to an xcSha1 request block to be used for the call. If this call is for a MIDDLE or LAST operation, the `kH` and `kL` and `magic` fields must contain the value returned from the previous call. Each of the other fields in the block will be set (by this macro) to the parameter of the same name:
  - `pSource` is the address of the data to be encrypted or decrypted.
  - `pHash` is the address of a writeable buffer.
  - `source_length` is the length of data to be hashed.
  - `hash_length` is the length of the buffer pointed to by `pHash`.
  - `options` is the exclusive or of the options available to the xcSha1 function (see the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference).

Output

When this macro completes,

- `ReturnMsg` has been set.
- `pHash` contains the result of the operation

If `options` included FIRST or MIDDLE, `pSha1RB` contains information required for the next call in the chain, specifically `running_length`.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMGood</td>
<td>The function completed successfully</td>
</tr>
<tr>
<td>DMBadParm</td>
<td>One of the parameters was invalid</td>
</tr>
</tbody>
</table>

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

3.6 Header files for communications functions

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

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

3.7 Summary of functions

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

<table>
<thead>
<tr>
<th>Function name</th>
<th>Definition</th>
<th>Coprocessor / workstation</th>
<th>Page reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>BuildParmBlock</td>
<td>Build a parameter block.</td>
<td>C, W</td>
<td>See page 35</td>
</tr>
<tr>
<td>Cas_proc_retc</td>
<td>Prioritizes a return code in the reply CPRB.</td>
<td>C</td>
<td>See page 38</td>
</tr>
<tr>
<td>CSNC_SP_SCSRFBSS</td>
<td>Send a request to the coprocessor.</td>
<td>W</td>
<td>See page 41</td>
</tr>
<tr>
<td>CSUC_BULDCPRB</td>
<td>Construct a well-formed CPRB block.</td>
<td>W</td>
<td>See page 41</td>
</tr>
<tr>
<td>CSUC_PROCRETC</td>
<td>Prioritize a return code.</td>
<td>W</td>
<td>See page 43</td>
</tr>
<tr>
<td>FindFirstDataBlock</td>
<td>Search for the first data block.</td>
<td>W, C</td>
<td>See page 44</td>
</tr>
<tr>
<td>FindNextDataBlock</td>
<td>Search for the next data block.</td>
<td>W, C</td>
<td>See page 45</td>
</tr>
<tr>
<td>find_first_key_block</td>
<td>Search for the first key block.</td>
<td>W, C</td>
<td>See page 46</td>
</tr>
<tr>
<td>find_next_key_block</td>
<td>Search for the next key block.</td>
<td>W, C</td>
<td>See page 47</td>
</tr>
<tr>
<td>jFindFirstDataBlock</td>
<td>Search for the first data block.</td>
<td>C, W</td>
<td>See page 48</td>
</tr>
<tr>
<td>jFindNextDataBlock</td>
<td>Search for the next data block.</td>
<td>C, W</td>
<td>See page 49</td>
</tr>
<tr>
<td>jfind_first_key_block</td>
<td>Search for the first key block.</td>
<td>C, W</td>
<td>See page 50</td>
</tr>
<tr>
<td>jfind_next_key_block</td>
<td>Search for the next key block.</td>
<td>C, W</td>
<td>See page 51</td>
</tr>
<tr>
<td>jparm_block_valid</td>
<td>Examine and verify a parameter block</td>
<td>C, W</td>
<td>See page 52</td>
</tr>
<tr>
<td>keyword_in_rule_array</td>
<td>Search for a keyword in the rule array.</td>
<td>C, W</td>
<td>See page 53</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Access</td>
<td>See page</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>otk_build_tokens</strong></td>
<td>Builds one or two key tokens and encrypts as required</td>
<td>C</td>
<td>54</td>
</tr>
<tr>
<td><strong>parm_block_valid</strong></td>
<td>Examine and verify a parameter block.</td>
<td>W, C</td>
<td>57</td>
</tr>
<tr>
<td><strong>parse_keys</strong></td>
<td>Parses one or more key tokens from the key block structure, validates the keys and decrypts them, returning the constituent parts for use in a command processor.</td>
<td>C</td>
<td>58</td>
</tr>
<tr>
<td><strong>rule_check</strong></td>
<td>Verify a rule array.</td>
<td>C, W</td>
<td>62</td>
</tr>
<tr>
<td><strong>saf_process_key_label</strong></td>
<td>Process a key label.</td>
<td>W</td>
<td>65</td>
</tr>
<tr>
<td><strong>TOK_ACCESS_AND_INIT_BUILD</strong></td>
<td>Set up an operation tracking token build structure to be used in the otk_build_tokens function.</td>
<td>C</td>
<td>66</td>
</tr>
<tr>
<td><strong>TOK_ACCESS_AND_INIT_PARSE</strong></td>
<td>Set up an operation tracking token structure for use in the parse_keys function.</td>
<td>C</td>
<td>67</td>
</tr>
</tbody>
</table>
3.7.1 BuildParmBlock - build a parameter block

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

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

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

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

Function prototype

```c
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 ith item, in bytes.
- `Data i` is a pointer to the ith data item to be added.

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 or jFindNextDataBlock to parse the result.

For example, first define a buffer that is long enough to hold both the request parameter block, the reply parameter block, and the CPRB structure.

```c
BlockLength = 0;
pCprb = (CPRB *) & (Buffer);
pRequestBlock = &(Buffer) + sizeof(CPRB_structure);
```

**Step 1: Add the subfunction code**

```c
BlockLength +=2
*(((USHORT *) pReqBlk) = htonl( CCAFNC1_ID );
```

<table>
<thead>
<tr>
<th>CPRB Structure (Empty)</th>
<th>(Empty) reply parameter block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subfunction code</td>
<td></td>
</tr>
</tbody>
</table>
Step 2: Add the rule array

\[
\text{BlockLength } += \text{BuildParmBlock(pRequestBlock+BlockLength,} \\
\text{1, /* adding 1 rule array */} \\
\text{(*pRuleArrayCount) *8, /* length of rule array */} \\
\text{pRuleArray );}
\]

Step 3: Add the verb unique data

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

36 IBM 4767 CCA UDX Reference
Step 4: Add the key blocks

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

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

Note: for the next step, see the CSUC_BULDCPRB function.

Output

On successful exit from this function:

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

Return codes

This function has no return codes.

Notes:

Building the Parameter Blocks

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

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

For more information about key block structures, see “Key blocks” on page 10.

See “UDX sample code” on page 306 for sample code which includes parameter block building.

Byte Alignment of Structures

It is important that all structures which are passed from the host to the coprocessor or the coprocessor to the host be aligned consistently. If you are passing a user-defined structure to the coprocessor, either as verb unique data or as key data, you must ensure that your compilers align the host and coprocessor structures on the same boundaries.
3.7.2 Cas_proc_retc - prioritize return code
This function is available only 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. In addition, if the return code of this error is 8 or greater, the error will be logged by the logging daemon, including the line number and function from which this function was called.

Function prototype

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

Input
On entry to this routine:

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

Output
On successful exit from this routine:

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

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>ERROR</td>
<td>The return code in msg was greater than Warning level (level 4).</td>
</tr>
</tbody>
</table>
3.7.3 CSNC_SP_SCSRFBSS - send a request to the coprocessor

This function is available only on the workstation host.

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

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

Function prototype

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

Input

On entry to this routine:

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

pMsg is a pointer to a variable for the return code of the function.

Output

On successful exit from this routine:

pCprb contains the reply CPRB, with the correct Reply Parameter Block address.

pCprb->replied_parm_block_length has the updated value of the amount of data returned in the reply parameter block.

pCprb->replied_data_block_length has the updated value of the amount of data returned from the coprocessor in the data block.

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

Note: the data in pCprb->reply_data_block has been overwritten for pCprb->reply_data_block_length bytes. This is true whether or not pCprb->replied_data_block_length is less than pCprb->reply_data_block_length or even if an error was returned and pCprb->replied_data_block_length is zero. If pCprb->replied_data_block_length is 0, the contents of pCprb_reply_data_block are indeterminate.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>ERROR</td>
<td>The return code in msg was greater than warning level (level 4).</td>
</tr>
<tr>
<td>E_ALLOCATE_MEM</td>
<td>Unable to allocate memory for checking data.</td>
</tr>
<tr>
<td>RT_SWERR</td>
<td>An error was encountered in the CPRB.</td>
</tr>
<tr>
<td>E_INVALID_MAC.VAL</td>
<td>The data returned from the coprocessor could not be validated.</td>
</tr>
</tbody>
</table>
Other error codes may be returned, depending on the functions called in the coprocessor section of the code.
3.7.4 CSUC_BULDCPRB - build CPRB

This function is available only on the workstation host.

CSUC_BULDCPRB builds a new CPRB from a request or reply block built with BuildParmBlock and an optional data field. Length fields in the CPRB are filled with big-endian integers to conform with the standard CPRB format.

Function prototype

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

Input

On entry to this routine:

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

Output

On successful exit from this routine:

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

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

• `reply_data_block_length` is the length of the reply data block.

• `reply_data_block` is a pointer to the reply data block in the host memory.

The request data block is copied into the message which is sent to the coprocessor. The reply data block is copied by the device driver into the buffer specified when the request returns from the coprocessor. The length of the data copied into the reply data block may be found in the CPRB structure, in the field `replied_data_block_length`. This length field is in big-endian format, as are all length fields in the CPRB structure. The communication method requires that they be a multiple of eight bytes long.

**Return codes**

This function has no return codes.
3.7.5 CSUC_PROCRETC - prioritize return code

This function is available only on the workstation host.

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

Function prototype

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

Input

On entry to this routine:

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

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

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

Output

On successful exit from this routine:

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

reason_code_ptr contains the reason code matching the priority of return_code_ptr.

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

Return codes

This function has no return codes.
3.7.6 FindFirstDataBlock - search for address of first data block

This function is available only on the workstation host. On the coprocessor, use “jFindFirstDataBlock - search for address of first data block.”

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

Function prototype

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

Input

On entry to this routine:

- `pCprb` is a pointer to the CPRB, which has the parameter block attached.
- `ParmBlockChoice` is a value of either SEL_REQ_BLK or SEL_REPLY_BLK, indicating whether the structure you have passed is a Request Parameter Block or a Reply Parameter Block.

Output

On successful exit from this routine:

- `pFirstDataBlock` contains the address of the first data block in the Verb Unique Data.
- FindFirstDataBlock returns a boolean value of TRUE if a data block was found, FALSE otherwise.

Return codes

This function has no return codes.
3.7.7 FindNextDataBlock - search for address of next data block

This function is available only on the workstation host. On the coprocessor, use "jFindNextDataBlock - search for address of next data block."

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

Function prototype

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

Input

On entry to this routine:

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

Output

On successful exit from this routine:

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

Return codes

This function has no return codes.
3.7.8  find_first_key_block - search for first key data block

This function is available only on the workstation host. On the coprocessor, use "jfind_first_key_block - search for first key data block."

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 subsequent key blocks in the parameter block.

Function prototype

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

Input

On entry to this routine:

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

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

Output

On successful exit from this routine:

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

find_first_key_block returns a boolean value of TRUE if key data was found, FALSE otherwise.

Return codes

This function has no return codes.
3.7.9 find_next_key_block - find address of next key data block

This function is available only on the workstation host. On the coprocessor, use "jfind_next_key_block - find address of next key data block."

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

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

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

Function prototype

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

Input

On entry to this routine:

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

this_keyblock is a pointer to a key block within the parameter block. The function attempts to locate the key block following this one.

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

Output

On successful exit from this routine:

next_keyblock contains the address of the key block following the one specified by this_keyblock or NULL if none was found.

find_next_key_block returns a boolean value indicating whether new key data was found.

Return codes

This function has no return codes.
3.7.10  jFindFirstDataBlock - search for address of first data block

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

\textit{jFindFirstDataBlock} locates the address of the first data block in the Verb Unique Data (VUD). If the parameter block contains Verb Unique Data, the address of the first data block is returned and the function result is set to \textbf{TRUE}. If there is no Verb Unique Data, the function result is set to \textbf{FALSE}.

\textbf{Function prototype}

\begin{verbatim}
boolean jFindFirstDataBlock( VUDPTR pVUD,
                           VUD_DATA_RECORD **ppFirstDataBlock )
\end{verbatim}

\textbf{Input}

On entry to this routine:

\textit{pVud} is a pointer to the verb unique data section of the Request Parameter Block. The Operation Tracking Structure holds this address in the \textit{c.t2.in_vud} field.

\textbf{Output}

On successful exit from this routine:

\textit{ppFirstDataBlock} contains the address of the first data block in the Verb Unique Data.

\textit{jFindFirstDataBlock} returns a boolean value of \textbf{TRUE} if a data block was found, \textbf{FALSE} otherwise.

\textbf{Return codes}

This function has no return codes.
### jFindNextDataBlock - search for address of next data block

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

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

**Function prototype**

```c
boolean jFindNextDataBlock( VUDPTR *pVUD,
                           VUD_DATA_RECORD *pThisDataBlock,
                           VUD_DATA_RECORD **ppNextDataBlock )
```

**Input**

On entry to this routine:

- `pVud` is a pointer to the verb unique data section of the Request Parameter Block. The Operation Tracking Structure stores this value in the `c.t2.in_vud` field.

- `pThisDataBlock` is a pointer to the current data block. The function attempts to find the data block following the one pointed to by this parameter.

**Output**

On successful exit from this routine:

- `ppNextDataBlock` contains the address of the data block following `pThisDataBlock` or `NULL` if none was found.

- `jFindNextDataBlock` returns a boolean value indicating whether a block was found.

**Return codes**

This function has no return codes.
3.7.12 jfind_first_key_block - search for first key data block

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

jfind_first_key_block finds the address of the first key data block attached to the Request 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 jfind_next_key_block, which is used to locate subsequent key blocks in the parameter block.

Function prototype

```c
boolean jfind_first_key_block(VUDPTR pVUD,
                               key_data_structure **first_keyblock)
```

**Input**
On entry to this routine:

- `pVUD` is a pointer to the start of the verb unique data in the Request Parameter Block. The Operation Tracking Structure field `c.t2.in_vud` points to this area.

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

- `first_keyblock` contains the address of the first key block contained in the request parameter block in the operation tracking structure `pOptx`.

- `jfind_first_key_block` returns a boolean value of TRUE if key data was found, FALSE otherwise.

**Return codes**
This function has no return codes.
3.7.13  jfind_next_key_block - find address of next key data block
This function is available on both the workstation host and the coprocessor.

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

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

Function prototype

    boolean jfind_next_key_block( VUDPTR pVUD,
                                key_data_structure *this_keyblock,
                                key_data_structure **next_keyblock)

Input
On entry to this routine:

pVUD is a pointer to the verb unique data section of the Request Parameter block. The Operation Tracking structure holds the required pointer in the c.t2.in_vud field.

this_keyblock is a pointer to a key block within the parameter block. The function attempts to locate the key block following this one.

Output
On successful exit from this routine:

next_keyblock contains the address of the key block following the one specified by this_keyblock or NULL if none was found.

jfind_next_key_block returns a boolean value indicating whether new key data was found.

Return codes
This function has no return codes.
3.7.14  jparm_block_valid - examine and verify a parameter block
This function is available on both the workstation host and coprocessor.

*jparm_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 jparm_block_valid ( RBFPTR parm_block,
                                uint32_t parm_block_length )

**Input**

On entry to this routine:

*parm_block* is a pointer to the parameter block to be tested. The Operation Tracking structure stores a pointer to the Request Parameter Block in the *c.t2.in_parm_block* field.

*parm_block_length* is the length of the parameter block.

**Output**

On successful exit from this routine:

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

**Return codes**

This function has no return codes.
3.7.15  **keyword_in_rule_array - search for rule array keyword**

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

*keyword_in_rule_array* determines whether a specified rule array keyword is present in the rule array passed within the given request block.

Input parameters are a pointer to the parameter block 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.

Before using this function, the caller should have verified the integrity of the CPRB using function *parm_block_valid* or *jparm_block_valid*. See “parm_block_valid - examine and verify a parameter block” on page 57 for information about *parm_block_valid*.

**Function prototype**

```c
boolean keyword_in_rule_array ( RBFPTR parm_block,
                                 rule_array_element keyword )
```

**Input**

On entry to this routine:

*parm_block* is a pointer to the parameter block structure.

*keyword* is the keyword you are looking for in the rule array.

**Output**

On successful exit from this routine:

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

**Return codes**

This function has no return codes.
3.7.16  otk_build_tokens - build and encrypt one or more key tokens

This function is available only on the coprocessor.

`otk_build_tokens` takes an operation tracking structure with certain fields containing clear key data and information about the expected token, and returns encrypted keys with the specified attributes.

**Function prototype**

```c
uint32_t otk_build_tokens ( optx_t *pOptx )
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure, with the fields set as follows.
- Within the `pOptx` structure is at least one `optokbld[]` member with the following fields set:
  - `bldiflags` may contain one or more of the following constants:
    - `BLDI_S390_USE_NEW_MK`
    - `BLDI_USE_OLD_MK`
    - `BLDIDES_BLDI_DES_CHK_CV_EMK`
    - `BLDI_DES_EXT_CV_NULL`
    - `BLDIIGN_KEY_TYPE`
    - `BLDI_DESIGN_PARITY`
    - `BLDI_DES_NO_SET_CV`
    - `BLDI_AESIGN_BITLEN`
    - `BLDI_DES_SET_TM_CVZ`
    - `BLDI_S390_SET_NOCV`
    - `BLDI_KEYCLEAR`
    - `BLDI_DESXLAT_CNTL`
    - `BLDI_CKEYIS_OPK`
    - `BLDI_CHK_TOKO_SZ`
  - `ckey_len` must indicate the length of the data in the `ckey` field.
  - `ckey` must point to the clear key to be placed in the token. Further descriptions of the data in this field are below.
  - `ckey2_len` must indicate the length of the data in the `ckey2` field.
  - `ckey2` may be a pointer to the clear public key to be placed in the token (if this key is a RSA key).
  - `kt` must be the type of key to be built (unless `pSRCtok` contains a valid token). These token types are defined in `cmntyped.h`, and will be defined as `kt.des`, `kt.aes`, or `kt.hmac`, depending on the type of token to be built.
  - `tok_out` is a pointer to a buffer where the new token will be created.
• **tok_out_len** is the length of the output buffer.

• **tok_type_out** is one of **TOKT_INT_KEY** (to build an internal token), **TOKT_EXT_KEY**, or **TOK_VAR_INT** or **TOK_VAR_EXT**.
  - If **tok_type_out** is **TOKT_EXT_KEY**, or **TOK_VAR_EXT**, **pKEKtok** must be a pointer to a key encrypting key in **optx_tok_t** format, which will be used to encrypt the output token.
  - If **tok_type_out** is one of **TOK_VAR_EXT** or **TOK_VAR_INT**, and the **pSRCtok** field is **NULL**, 
    - **cv** must point to the key usage fields for the new token.
    - **cv2** must point to the key management fields for the new token.
    - **tok_alg_val** must be set to the key type of the new token.
    - **tok_alg_sec** may point to the key name section for the new token.
    - **tok_alg_sec2** must point to the IBM extended associated data.
    - **tok_alg_sec3** may point to the user associated data.

• **wt** must be one of **WRAP_TK_LEGACY**, **WRAP_TK_ENH_CBC**, **WRAP_TK_AESKW**, **WRAP_TK_PKOAEP2**, or **WRAP_TKB_DEFAULT**.

• **pSRCKtok** is a pointer to a key token on which this key is to be based (in which case **tok_type_out**, **tok_alg_out**, and **kt** are optional), OR, if **pSRCKtok** is **NULL**, then **tok_alg_out**, **tok_type_out**, and **kt** are all **REQUIRED**.

• **tok_alg_out** is a pointer to the algorithm type of the expected key. Acceptable types are:
  - **TOKA_DES** – a DES or TDES key token. In this case, the following fields must also be set:
    - **ckey_len** must be either 8 or 16.
    - **ckey** must point to the clear DES key, 8 or 16 bytes long.
    - **cv_len** must be the length of data in the **cv** field.
    - **cv** may be a DES control vector (or the control vector will be built depending on the **kt** field).
  - **TOKA_AES** – an AES key token. In this case, the following fields must also be set:
    - **ckey_bit_len** must be 128, 192, or 256.
    - **ckey_len** must be 16, 24, or 32.
    - **ckey** must point to the clear AES key for the token, 16, 24, or 32 bytes of data.
  - **TOKA_RSA** – an RSA key token. In this case, the following fields must also be set:
    - **ckey_len** must be the length of the data in the **ckey** field.
    - **ckey** must point to an RsaKeyTokenHeader with the clear RSA key.
  - **TOKA_DSA** – a DSA key token.
  - **TOKA_HMAC** – an HMAC key token.
    - **ckey_bit_len** must be set to the bit length of the clear key (80-2048).
    - **ckey_len** must be the length of the provided ckey.
    - **ckey** must point to the clear HMAC key.
  - **TOKA_NO_ALG** – valid only if the **pSRCKtok** is pointing to a valid key token.
Output
On successful exit from this routine:

The `tok_out` field in the `optokbld` field of the `pOptx` structure is pointing to a correctly structured and encrypted key token of the type specified.

The `bldoflags` field in the `optokbld` field of the `pOptx` structure may contain one or more of the following:

- **BLDO_TOKOUT_ALLOC**: The newly created token is stored in memory that was allocated for the purpose. The CCA dispatcher will automatically free this memory when the command processor returns. This flag must not be reset or zeroed out, to avoid a memory leak in the coprocessor.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>E_INV_CKEY_BITLEN</td>
<td>The bit length of the clear key was invalid. (Typically HMAC keys)</td>
</tr>
<tr>
<td>RT CV_CONFLICT</td>
<td>There was an issue with the control vector (Typically DES keys)</td>
</tr>
<tr>
<td>RT SWERR</td>
<td>An error was encountered in the token. This was probably a setup error where the request was badly formed. Check your code.</td>
</tr>
<tr>
<td>RT TKN_UNUSEABLE</td>
<td>There was an error in the supplied token.</td>
</tr>
</tbody>
</table>
3.7.17  parm_block_valid - examine and verify a parameter block

This function is available only on the workstation host. On the coprocessor use "jparm_block_valid - examine and verify a parameter block."

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

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

Function prototype

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

Input

On entry to this routine:

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

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

Output

On successful exit from this routine:

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

Return codes

This function has no return codes.
parse_keys - decrypt and parse the keys from a key block

This function is available only on the coprocessor.

parse_keys takes an operation tracking structure with certain fields containing expected descriptions of
the keys in the key block (in order) and determines whether the keys provided match the expected keys. If
so, the keys will be decrypted and the clear keys, control vectors, and other algorithmic information will be
returned in the provided operation tracking structure.

Function prototype

```c
uint32_t parse_keys ( optx_t *pOptx,
                       uint32_t CV_check_value)
```

Input

On entry to this routine:

pOptx is a pointer to the operation tracking structure, with the fields set as follows.

Within the pOptx structure,

- `optok_cnt` is 1 or more, and the `optok[0]` to `optok[optok_cnt-1]` fields are set as follows:
  - `tok_alg_exp` contains one or more of the following flags:
    - `TOKAE_AES`, `TOKAE DES`, `TOKAE HMAC`, `TOKAE RSA`, `TOKAE ECC`  
  - `tok_type_exp` contains one or more of the following flags:
    - `TOKTE_EMPTY`, `TOKTE NULL`, `TOKTE_RAW_KEY`, `TOKTE_JUNK`, `TOKTE_LABEL`,
      `TOKTE_EXT_KEY`, `TOKTE INT_KEY`, `TOKTE_TBLK`, `TOKTE_RKX`,
      `TOKTE_VAR_EXT`, `TOKTE_VAR_INT`  
  - `in_flags` may contain one or more of the following flags:
    - `TOKI KEK_NEEDED`, `TOKI KEK_HERE`, `TOKI OPT_KEY`, `TOKI CONT ON_ERROR`,
      `TOKI WR_TOKEN`, `TOKI IGN_KEY`, `TOKI IGN CV`, `TOKI NODECR_KEY`,
      `TOKI NO CHECK TOK`, `TOKI CHK MK`, `TOKI IGN KEYPART (and others)`  
  - `algflags` may contain one or more of the following flags:
    - `ALGI IGN_ALGKEY_CHECK`, `ALGI DES MH DOUBLE`, `ALGI DES EXT IGN TM`,
      `ALGI MH ENCR KEY`, `ALGI GET RTND`, `ALGI RSA RTND ALLOW EXT`,
      `ALGI MODMIN ORIG`, `ALGI MODMIN PKE DSV`, `ALGI MODMIN PKD`,
      `ALGI PKA INT MH PVT`, `ALGI PKA EXT MH PVT`, `ALGI RSA IGN PVT`,
      `ALGI PKI INT MH PUB`, `ALGI PKA INT MH PUB`, `ALGI RSA CHK PUB EVEN`,
      `ALGI RSA CHK MAX MOD`, `ALGI_TBLK IGN CHK`, `ALGI_TBLK_TX RSPUB`,
      `ALGI_TBLK NOMACRECOV`, `ALGI_TBLK NOMACVAR`, `ALGI_TBLK_MACINPLC` (and others)
  - `algflags2` may contain one or more of the following constants:
    - `ALGI2 DES MH EXT V1`, `ALGI2 DES MH NULL CV`, `ALGI2 DES EXT XT CHKS`,
      `ALGI2 IGN AES LRC`, `ALGI2 CHK RSA ASYM MK`, `ALGI2 VAR MH KEY`,
      `ALGI2_RKX TRX HERE`, `ALGI2_RKX EKEK HERE`, `ALGI2_RKX_EXP HERE`,
      `ALGI2_RKX_TBLK HERE`  
  - `usage` may contain `TOKU PKA SIGN`
- `tok_alg_in` may contain a rule array element indicating the intended use of the key. 

CV_check_value is set to 0.

**Output**

On successful exit from this routine:

*pOptx->c2.outCPRB->secy_return_code* is set to the proper error, warning, or informational value returned from parse_keys(). See *pOptx->optok[i]->error* below, as well as the Return Codes section.

All *pOptx->optok_cnt* elements of the *pOptx->optok[]* array are set with the following:

- `tok_alg` is one of **TOKA_AES, TOKA_DES, TOKA_DSA, TOKA_HMAC, TOKA_RSA, TOKA_ECC**
- `tok_type` is one of **TOKT_EMPTY, TOKT_NULL, TOKT_RAW_KEY, TOKT_JUNK, TOKT_LABEL, TOKT_EXT_KEY, TOKT_INT_KEY, TOKT_TBLK, TOKT_RKX, TOKT_VAR_INT, TOKT_VAR_EXT**
- `outflags` may contain one or more of the following flags:
  - `TOKO_CKEY_ALLOC`, `TOKO_CKEY_OPT_ALLOC`, `TOKO_TOK_ALLOC`, `TOKO_S390_ZEROCV`, `TOKO_S390_V1`, `TOKO_S390_VAR21`, `TOKO_DESV11_TM_NOT_SINGLE`, `TOKO_XPORT_PROHIB`, `TOKO_NOCV`, `TOKO_CV_MISSING`, `TOKO_MK_CUR`, `TOKO_MK_OLD`, `TOKO_MK_NEW`, `TOKOkeley_USED`, `TOKO_GOT_RTND`, `TOKO_PKA_NO_PVT_KEY`, `TOKO_PKA_NO_PUB_KEY`, `TOKO_PKA_CLR_PVT`, `TOKO_AES_CLR_KEY`  
- `error` contains any warning or informational error which was returned when this key was parsed.
- `wt` contains one of **WRAP_TK_LEGACY, WRAP_TK_ENH_CBC, WRAP_TK_AESKW, WRAP_TK_PKOAEP2**  
- `tok_len` is the length of the field pointed to by `tok`.

**If the token in position j was a DES token:**

- `optok[i]->tok_type = TOKT_INT_KEY` if an internal token, **TOKT_EXT_KEY** if an external token.
- `optok[i]->tok` is a pointer to the encrypted token in the input request block (unless it was set to a specific location before calling parse_keys).
- `optok[i]->ckey` is a pointer to the clear DES key in a **SCA_TRIPLE_KEY** structure.
- `optok[i]->cv` is a pointer to the control vector base (16 bytes).
- `optok[i]->tok_alg_sec` is a pointer to the tokenmarks section of the token.

**If the token in position j was an AES fixed-length token:**

- `optok[i]->tok` is a pointer to the encrypted token in the input request block (unless it was set to a specific location before calling parse_keys).
- `optok[i]->tok_type = TOKT_INT_KEY` if an internal token, **TOKT_EXT_KEY** if it was a clear key token.
- `optok[i]->tok` is a pointer to the encrypted token in the input request block (unless it was set to a specific location before calling parse_keys).
- `optok[j]->ckey` is a pointer to the clear AES key.
- `optok[j]->ckey_bit_length` is the bit length of the clear key.
- `optok[j]->ckey_length` is the byte length of the key in the `optok[j]->ckey` field.
- `optok[j]->cv` is a pointer to the control vector base (8 bytes of 0x00).

If the token in position `j` was an AES or HMAC variable-length token:
- `optok[j]->tok` is a pointer to the encrypted token in the input request block (unless it was set to a specific location before calling `parse_keys`).
- `optok[j]->tok_type = TOKT_VAR_INT` if an internal token, `TOKT_VAR_EXT` if an external token.
- `optok[j]->tok` is a pointer to the encrypted token in the input request block (unless it was set to a specific location before calling `parse_keys`).
- `optok[j]->ckey` is a pointer to the clear AES key.
- `optok[j]->ckey_bit_len` is the bit length of the clear key.
- `optok[j]->ckey_length` is the byte length of the key in the `optok[j]->ckey` field.
- `optok[j]->cv` is a pointer to the KUF fields.
- `optok[j]->cv2` is a pointer to the KMF fields.
- `optok[j]->tok_alg_sec` is a pointer to the key label in the encrypted key token.
- `optok[j]->tok_alg_sec2` is a pointer to the IBM extended associated data in the encrypted key token.
- `optok[j]->tok_alg_sec3` is a pointer to the user associated data in the encrypted key token.
- `optok[j]->ckey_opt` points to the encrypted payload of the key.
- `optok[j]->ckey_opt_len` is the payload length in bytes.

If the token in position `j` was an RSA token:
- `optok[j]->tok` is a pointer to the encrypted token in the input request block (unless it was set to a specific location before calling `parse_keys`).
- `optok[j]->tok_type = TOKT_INT_KEY` if an internal token, `TOKT_EXT_KEY` if an external token.
- `optok[j]->ckey` is a pointer to the RSA key token header of the decrypted token, such as would be returned by `RecoverPkaClearToken`.
- `optok[j]->tok_alg_sec` is a pointer to the private key section of the clear key, such as would be returned by `getPrivateKeySection`.
- `optok[j]->tok_alg_sec2` is a pointer to the public key section of the clear key, such as would be returned by `getPublicKeySection`.
- `optok[j]->tok_alg_val` is a pointer to the modulus bit length in native byte order.

If the token in position `j` was an ECC token:
- `optok[j]->tok` is a pointer to the encrypted token in the input request block (unless it was set to a
specific location before calling parse_keys).

- optok[j]->tok_type = TOKT_INT_KEY if an internal token, TOKT_EXT_KEY if an external token.
- optok[j]->ckey is a pointer to the raw clear private key p (unless ALGI2_PKA_GET_OPK was selected in the algflags2 field of the input token, in which case this is a pointer to the clear OPK for the key.
- optok[j]->ckey2 is a pointer to the raw public key q.
- optok[j]->ckey_opt_len is the length of the OID information.
- optok[j]->ckey_opt is a pointer to the OID information in the private key.
- optok[j]->cv_len is the length of the private key Usage flag.
- optok[j]->cv is a pointer to the private key Usage flag.
- ptok[j]->tok_alg_sec_len is the length of the private key section of the encrypted key, in native byte order.
- optok[j]->tok_alg_sec is a pointer to the private key section of the encrypted key, if it exists.
- optok[j]->tok_alg_sec2_len is the length of the public key section of the encrypted key, in native byte order.
- optok[j]->tok_alg_sec2 is a pointer to the public key section of the encrypted key.
- optok[j]->tok_alg_sec3_len is the length of the key label section of the encrypted key, in native byte order.
- optok[j]->tok_alg_sec3 is a pointer to the key label section of the encrypted key.
- optok[j]->tok_alg_sec4_len is the length of the IBM Extended associated data section of the encrypted key, in native byte order.
- optok[j]->tok_alg_sec4 is a pointer to the IBM Extended associated data section of the encrypted key.
- optok[j]->tok_alg_sec5_len is the length of the User-defined associated data section of the encrypted key, in native byte order.
- optok[j]->tok_alg_sec5 is a pointer to the User-defined associated data section of the encrypted key.
- optok[j]->tok_alg_val is a pointer to the curve length (bit length of p) in native byte order.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>E_INV_CKEY_BITLEN</td>
<td>The bit length of the clear key was invalid. (Typically HMAC keys)</td>
</tr>
<tr>
<td>RT_CV_CONFLICT</td>
<td>There was an issue with the control vector (Typically DES keys)</td>
</tr>
<tr>
<td>RT_SWERR</td>
<td>An error was encountered in the token. This was probably a setup error where the request was badly formed. Check your code.</td>
</tr>
<tr>
<td>RT_TKN_UNUSEABLE</td>
<td>There was an error in the supplied token.</td>
</tr>
</tbody>
</table>

4767 functions  61
3.7.19 rule_check - verify rule array

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

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

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

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

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

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.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyword</td>
<td>This is the eight-character rule array keyword, with a byte of 0x00 added.</td>
</tr>
</tbody>
</table>
| 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. |

Note: If you set the map_value to REQUIRED_GROUP_RULE_MAP, the order_no to the element, and the map_value to 0 for a given element, rule_check() will return E_RULE_ARRAY(REQD_GROUP) when no keyword from that group is present (as long as no invalid keywords or invalid combinations of keywords were encountered before the array was completely processed).

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

Output
On successful exit from this routine:
pReturn_message is a pointer to the location where the function stores the error code, if the rule array is not correct.
pRule_value contains an array of integers, the ith integer is the map value of the keyword from the ith set which is present in the rule array, or INVALID_RULE if there is no keyword from that set.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>E_RULE_ARRAY_KWD</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>E_RULE_ARRAY_COMBINE</td>
<td>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.</td>
</tr>
<tr>
<td>E_RULE_ARRAY_REQD_GROUP</td>
<td>Indicates that no keyword was found in the rule array from a required group of keywords.</td>
</tr>
</tbody>
</table>

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}};

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.
  1. INIT-AC
  2. CHGEXPDT
  3. CHG-AD
  4. 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}};

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.
3.7.20 saf_process_key_label - process key label

This function is available only on the workstation host.

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

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

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

**Function prototype**

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

**Input**

On entry to this routine:

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

**Output**

On successful exit from this routine:

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

**Return codes**

This function has no return codes.
TOK_ACCESS_AND_INIT_BUILD – set up operation tracking
token build structure

This macro is available on the coprocessor.

*TOK_ACCESS_AND_INIT_BUILD* sets the opaque fields of an `optx_tokbld_t` structure properly such that
memory leaks and overruns from improper use are eliminated.

**Macro format**

```c
#define TOK_ACCESS_AND_INIT_BUILD ( optx_tokbld_t *pBldTok,
                                 optx_t *pOptx,
                                 int number )
```

**Input**

On entry to this macro:

- `pBldTok` is a pointer to an `optx_tokbld_t` which has been set to NULL.
- `pOptx` is a pointer to the operation tracking structure for this verb.
- `number` is the number (NOT a variable), between 0 and 2 inclusive, that indicates the position of the key
  in the key block structure (0 based).

**Output**

When this macro completes:

- `pBldTok` points to `pOptx->optokbld[number]`.
- The appropriate data space has been allocated for `pBldTok` and the opaque fields have been set
correctly.

**Return codes**

This macro has no return codes. However, if you use a variable instead of a numeral as the `number`, you
will get a compiler error.
3.7.22   TOK_ACCESS_AND_INIT_PARSE – set up operation tracking
token structure for parse_keys

This macro is available on the coprocessor.

*TOK_ACCESS_AND_INIT_PARSE* sets the opaque fields of an *optx_tok_t* structure properly such that
memory leaks and overruns from improper use are eliminated.

**Macro format**

```c
#define TOK_ACCESS_AND_INIT_PARSE ( optx_tok_t *pNewTok,
                                  optx_t *pOptx,
                                  int    number )
```

**Input**

On entry to this macro:

- *pNewTok* is a pointer to an *optx_tok_t* which has been set to NULL.
- *pOptx* is a pointer to the operation tracking structure for this verb.
- *number* is the number (NOT a variable), between 0 and 10 inclusive, that indicates the position of the key in the key block structure (0 based).

**Output**

On successful exit from this macro:

- *pNewTok* points to *pOptx->optok[number]*.

The appropriate data space has been allocated for *pNewTok* and the opaque fields have been set correctly.

**Return codes**

This macro has no return codes. However, if you use a variable instead of a numeral as the *number*, you will get a compiler error.
4 FCV management functions

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

This chapter 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.

You may load the FCV into a workstation coprocessor using the csulcnm utility. Refer to the IBM 4767 PCIe Cryptographic Coprocessor CCA Support Program Installation Manual for more detailed information.

4.1 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 definition */
#include "cam_fcv.h" /* Function control vector */
```

4.2 Summary of functions

Functions that interact with FCV include the following:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetMaximumCurveOrder</td>
<td>Gets the maximum curve order for ECC keys.</td>
</tr>
<tr>
<td>getSymmetricMaxModulusLength</td>
<td>Gets the maximum RSA key length.</td>
</tr>
<tr>
<td>isFunctionEnabled</td>
<td>Determines whether the FCV allows a particular function.</td>
</tr>
</tbody>
</table>
4.2.1 getMaximumCurveOrder - get ECC key length

This function is available only on the coprocessor.

getMaximumCurveOrder returns the maximum ECC key length (in bits) that can be used for encrypting or signing.

Function prototype

    uint32_t getMaximumCurveOrder( unsigned short *pCurveOrder)

Input

On entry to this routine:

pCurveOrder is a pointer to an unsigned short variable.

Output

On successful exit from this routine:

pCurveOrder contains the maximum authorized curve order.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>srdi_NOT_FOUND</td>
<td>The FCV was not found.</td>
</tr>
<tr>
<td>srdi_GENERAL_ERROR</td>
<td>Failed to access the SRDI manager.</td>
</tr>
<tr>
<td>srdi_ALLOC_ERROR</td>
<td>Could not allocate memory to open the FCV.</td>
</tr>
<tr>
<td>srdi_READ_ERROR</td>
<td>Could not read the FCV.</td>
</tr>
</tbody>
</table>
4.2.2 getSymmetricMaxModulusLength - get RSA key length

This function is available only on the coprocessor.

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

Function prototype

    uint32_t getSymmetricMaxModulusLength( unsigned short *pModLength)

Input

On entry to this routine:

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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>srdi_NOT_FOUND</td>
<td>The FCV was not found.</td>
</tr>
<tr>
<td>srdi_GENERAL_ERROR</td>
<td>Failed to access the SRDI manager.</td>
</tr>
<tr>
<td>srdi_ALLOC_ERROR</td>
<td>Could not allocate memory to open the FCV.</td>
</tr>
<tr>
<td>srdi_READ_ERROR</td>
<td>Could not read the FCV.</td>
</tr>
</tbody>
</table>
4.2.3 isFunctionEnabled - check whether a function is enabled

This function is available only on the coprocessor.

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

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

**Function prototype**

```c
boolean isFunctionEnabled( uint32_t 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 the table of FCV Possible Values below for a list of possible values.

`FunctionBitSelect` is the bit to be checked in the specified Function Control Vector byte. See the table of FCV Possible Values below 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.

<table>
<thead>
<tr>
<th>FCV Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function Byte Index</strong></td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
<tr>
<td>CCA_BASE_FUNCTION_BYTE</td>
</tr>
<tr>
<td>SYMMETRIC_FUNCTION_BYTE</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>SET_FUNCTION_BYTE</td>
</tr>
</tbody>
</table>

**Return codes**

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

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

To determine whether 56-bit DES encryption is allowed:

```c
if ( isFunctionEnabled( DES_FUNCTION_BYTE, FCV_56_BIT_DES) )
{
    /* use 56-bit DES encryption */
}
```
5 Master key manager functions

The CCA Master Key Manager provides access to the CCA master key registers on the PCIe 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 BBRAM.

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

5.1 Header files for master key manager functions

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

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

5.2 Overview of the CCA master keys

The coprocessor uses triple-length master keys to encrypt DES keys and PKA keys, each consisting of three independent eight-byte DES keys. Separate 32-byte AES keys are used to protect AES keys and PKA keys. The master keys are used to protect other data in the following ways.

- Single-length (eight-byte) DES 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.

- An enhanced CBC encryption method for TDES.

- Data encrypted with either of the AES master keys is protected using the AESKW method.

The `mk_set` parameter of the `mk_selectors` variable type identifies the set of master keys to be processed by a function. There is only one set of master keys on a workstation.

Within each master key set, CCA supports four types of master keys, AES and variable length token keys (AES_MK), PKA keys (ASYM_MK) and DES keys (SYM_MK). Each type of key consists of three master key values.

- Old Master Key (OMK) – the version of the master key that was in use prior to the current value. It is maintained to permit recovery of keys that were enciphered under the old master key.

- Current Master Key (CMK) – The current, operational master key. All keys in use in the system are enciphered under this key.

- New Master Key (NMK) – A new master key, which is being entered into the system to replace the current master key. It may be entered in the form of one or more key parts which are combined to form the final key, as a randomly generated value, or as a set of key shares, which were generated on a different coprocessor. AES master keys may only be entered as key parts in
CCA.

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, this value is calculated using MDC-4. For the **AES_MK** and the **APKA_MK**, this value is calculated using SHA-256.

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

- For the NMK register, three status values are possible.
  1. The register is empty. It does not contain any portion of a new master key value.
  2. The register is partially full. The last key part has not yet been combined into the value in the register.
  3. 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.

### 5.2.1 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 a ten character name. The master key data SRDI for DES keys is named **MSTRKEYS**. The master key data SRDI for asymmetric keys is **ASYMKMKKS**. The master key data SRDI for AES and variable token keys is named **AEMKEYS**. The master key data SRDI for RSA keys encrypted with AES is **APKAMKYS**.

### 5.2.2 Initialization of the master key SRDI

When the CCA application is first loaded into a new coprocessor, no master key SRDI exists in the BBRAM. 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, NCK, CMK, and OMK, are all set to binary zeroes.
- The state of CMK and OMK is set to invalid. The state of NCK is set to Empty.
- The master key verification patterns are set to binary zeroes.

### 5.2.3 CCA master key manager interface functions

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

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

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

The common entry processing performs the following functions.

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

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

5.2.4 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 fields:

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

- 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 field can be any of AES_MK, APKA_MK, ASYM_MK, SYM_MK, or BOTH_MK. BOTH_MK operates on the ASYM_MK and the SYM_MK at the same time, if they contain the same master keys. Otherwise, an error is returned.

For deprecated functions – earlier versions of many of the following functions were intended only for the symmetric (DES) master keys. Extended versions (usually beginning with "mkm") are capable of operating on the asymmetric, and AES master keys as well, and the use of these functions is preferred over the earlier versions.

5.3 Summary of functions

The following functions are summarized in this chapter.

<table>
<thead>
<tr>
<th>Function</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes_engine_with_MK</td>
<td>“Functions to encrypt and decrypt using the master key” (page 100)</td>
</tr>
<tr>
<td>aes_unwrap_with_hash_and_MK</td>
<td>“Functions to encrypt and decrypt using the master key” (page 100)</td>
</tr>
<tr>
<td>aes_unwrap_with_MK</td>
<td>“Functions to encrypt and decrypt using the master key” (page 100)</td>
</tr>
<tr>
<td>aes_wrap_with_hash_and_MK</td>
<td>“Functions to encrypt and decrypt using the master key” (page 100)</td>
</tr>
<tr>
<td>aes_wrap_with_MK</td>
<td>“Functions to encrypt and decrypt using the master key” (page 100)</td>
</tr>
<tr>
<td>Function</td>
<td>Section</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>clear_master_keys</td>
<td>&quot;Functions to set and manage the master keys&quot; (page 78)</td>
</tr>
<tr>
<td>combine_mk_parts</td>
<td>&quot;mkmCombineMKParts - combine master key parts&quot; (page 80)</td>
</tr>
<tr>
<td>compute_mk_verification_pattern</td>
<td>&quot;Functions to check master key values and status&quot; (page 93)</td>
</tr>
<tr>
<td>ede3_triple_decrypt_under_master_key</td>
<td>&quot;Functions to encrypt and decrypt using the master key&quot; (page 100)</td>
</tr>
<tr>
<td>ede3_triple_encrypt_under_master_key</td>
<td>&quot;Functions to encrypt and decrypt using the master key&quot; (page 100)</td>
</tr>
<tr>
<td>generate_mk_shares</td>
<td>&quot;mkmGenerateMKShares - generate master key shares&quot; (page 82)</td>
</tr>
<tr>
<td>generate_random_mk</td>
<td>&quot;mkmGenerateRandomMK - generate random master key&quot; (page 84)</td>
</tr>
<tr>
<td>get_master_key_status</td>
<td>&quot;&quot; (page 97)</td>
</tr>
<tr>
<td>get_mk_verification_pattern</td>
<td>&quot;Functions to check master key values and status&quot; (page 80)</td>
</tr>
<tr>
<td>init_master_keys</td>
<td>&quot;Functions to set and manage the master keys&quot; (page 78)</td>
</tr>
<tr>
<td>load_first_master_key_part</td>
<td>&quot;mkmLoadFirstMKPart - load first master key part&quot; (page 87)</td>
</tr>
<tr>
<td>load_mk_from_shares</td>
<td>&quot;mkmLoadMKFromShares - load master key shares&quot; (page 88)</td>
</tr>
<tr>
<td>mkmCombineMKParts</td>
<td>&quot;Functions to set and manage the master keys&quot; (page 78)</td>
</tr>
<tr>
<td>mkmGenerateMKShares</td>
<td>&quot;Functions to set and manage the master keys&quot; (page 78)</td>
</tr>
<tr>
<td>mkmGenerateRandomMK</td>
<td>&quot;Functions to set and manage the master keys&quot; (page 78)</td>
</tr>
<tr>
<td>mkmTDESDecryptUnderMasterKey</td>
<td>&quot;Functions to encrypt and decrypt using the master key&quot; (page 100)</td>
</tr>
<tr>
<td>mkmGetAsymVerificationPattern</td>
<td>&quot;Functions to check master key values and status&quot; (page 93)</td>
</tr>
<tr>
<td>mkmGetMasterKeyStatus</td>
<td>&quot;Functions to check master key values and status&quot; (page 93)</td>
</tr>
<tr>
<td>mkmLoadFirstMKPart</td>
<td>&quot;Functions to set and manage the master keys&quot; (page 78)</td>
</tr>
<tr>
<td>mkmLoadMKFromShares</td>
<td>&quot;Functions to set and manage the master keys&quot; (page 78)</td>
</tr>
<tr>
<td>mkmSetMasterKey</td>
<td>&quot;Functions to set and manage the master keys&quot; (page 78)</td>
</tr>
<tr>
<td>Function</td>
<td>Section</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mkmTDESEncryptUnderMasterKey</td>
<td>“Functions to encrypt and decrypt using the master key” (page 100)</td>
</tr>
<tr>
<td>reinit_master_keys</td>
<td>“Functions to set and manage the master keys” (page 78)</td>
</tr>
<tr>
<td>triple_decrypt_under_master_key</td>
<td>“Functions to encrypt and decrypt using the master key” (page 100)</td>
</tr>
<tr>
<td>triple_decrypt_under_master_key_with_CV</td>
<td>“Functions to encrypt and decrypt using the master key” (page 100)</td>
</tr>
<tr>
<td>triple_encrypt_under_master_key</td>
<td>“Functions to encrypt and decrypt using the master key” (page 100)</td>
</tr>
<tr>
<td>triple_encrypt_under_master_key_with_CV</td>
<td>“Functions to encrypt and decrypt using the master key” (page 100)</td>
</tr>
</tbody>
</table>
5.4 Functions to set and manage the master keys

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

**Note:** CCA calls the functions in these sections only in a command processor running while all other command processors are quiesced. It is strongly recommended that UDXes use the standard CCA master key verbs rather than these subroutines to avoid issues with concurrency. Optionally, users might set the run_type in the `ccax_cp_list` array to RUN_ISOLATED for verbs which call these routines.

### 5.4.1 Summary of functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear_master_keys</td>
<td>Clears a specified master key register.</td>
</tr>
<tr>
<td>combine_mk_parts</td>
<td>XOR an additional part into the DES new master key register.</td>
</tr>
<tr>
<td>generate_mk_shares</td>
<td>Split a 24-byte DES master key into shares.</td>
</tr>
<tr>
<td>generate_random_mk</td>
<td>Generate a random master key and place into the DES new master key register.</td>
</tr>
<tr>
<td>init_master_keys</td>
<td>Create and initialize the master key SRDIs.</td>
</tr>
<tr>
<td>load_first_master_key_part</td>
<td>Load the first part into an empty DES new master key register.</td>
</tr>
<tr>
<td>load_mk_from_shares</td>
<td>Reconstruct a DES master key from shares that were created by the</td>
</tr>
<tr>
<td></td>
<td>generate_mk_shares function and store the result in the new master key</td>
</tr>
<tr>
<td></td>
<td>register.</td>
</tr>
<tr>
<td>mkmCombineMKParts</td>
<td>Combine an additional master key part into the new master key register.</td>
</tr>
<tr>
<td>mkmGenerateMKShares</td>
<td>Splits a 24-byte master key into shares.</td>
</tr>
<tr>
<td>mkmGenerateRandomMK</td>
<td>Generates a random 24-byte master key.</td>
</tr>
<tr>
<td>mkmLoadFirstMKPart</td>
<td>Load the first part of a multi-part master key into the new master key</td>
</tr>
<tr>
<td></td>
<td>register.</td>
</tr>
<tr>
<td>mkmSetMasterKey</td>
<td>Activate the master key</td>
</tr>
<tr>
<td>reinit_master_keys</td>
<td>Delete all master key data and then create and initialize the master key</td>
</tr>
<tr>
<td></td>
<td>SRDIs.</td>
</tr>
<tr>
<td>set_master_key</td>
<td>Activate the DES master key.</td>
</tr>
</tbody>
</table>
5.4.2 clear_master_keys - clear master key

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

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

**Function prototype**

```c
long clear_master_keys ( struct optx *pOptx,
                        mk_selectors *pMKSelector );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `pMKSelector` 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 `AES_MK` if this set of master keys is intended for AES or variable length token key encryption, `ASYM_MK` if this set of master keys is intended for PKA key encryption, or `SYM_MK` if this set of master keys is used for DES key encryption. If `type_mks` is `BOTH_MK` and the state of the specified master key register of both the PKA and DES master key sets is the same, the specified master key register will be cleared and the state set to empty for both PKA and DES master key registers; otherwise an error message will be returned.

**Output**

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

The specified master key register has been cleared.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td>mk_SRDI_OPEN_ERROR</td>
<td>Could not open the master key SRDI.</td>
</tr>
<tr>
<td>mk_SEM_CLAIM_FAILED</td>
<td>Unable to access the SRDI Manager.</td>
</tr>
</tbody>
</table>
5.4.3 mkmCombineMKParts - combine master key parts

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

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

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

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

**Function prototype**

```c
long mkmCombineMKParts ( optx_t *pOptx,
unsigned char *key_part,
mk_selectors MKSelector,
boolean final_part );

long combine_mk_parts ( optx_t *pOptx,
unsigned char *key_part,
boolean final_part );
```

**Input**

On entry to this routine:

*pOptx* is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

*key_part* is a 24-byte (for PKA or DES keys) or 32-byte (for AES keys) clear text key part, which is to be combined into the value in the NMK register.

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

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

*final_part* is a boolean value, which the caller sets to TRUE when the key part is the final part of a new master key.

The NMK register must have been initialized with a call to *mkmLoadFirstMKPart* and zero or more calls to *mkmCombineMKParts*.
Output
This function returns no output. On successful exit from this routine:

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

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_SRDI_OPEN_ERROR</td>
<td>Could not open the master key SRDI.</td>
</tr>
<tr>
<td>mk_SEM_CLAIM_FAILED</td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td>mk_INCORRECT_STATE</td>
<td>The New Master Key register was not in the Partially Full state.</td>
</tr>
<tr>
<td>mk_WEAK_KEY</td>
<td>final_part was TRUE and the resulting key was a weak key. The prior state has been restored.</td>
</tr>
</tbody>
</table>
5.4.4 mkmGenerateMKShares - generate master key shares

*mkmGenerateMKShares* splits a 24-byte master key into shares for the mk_set being processed. The key is split into \( n \) separate shares, where any \( m \) of the shares can be used to recreate the master key value at a later time. This function is not available for the AES master keys.

The *generate_mk_shares* function is the equivalent of calling the *mkmGenerateMKShares* function with the MKSelector parameter set to \( \{ \text{MK_SET_DEFAULT} , \text{current_mk} , \text{SYM_MK} \} \).

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

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

**Function prototype**

\[
\text{long mkmGenerateMKShares ( \text{mk_selectors MKSelector},} \\
\text{ \text{mk_src_t KeySource,}} \\
\text{ \text{UCHAR mShareKey,}} \\
\text{ \text{UCHAR nShareGen,}} \\
\text{ \text{UCHAR *pShares [ ],}} \\
\text{ \text{optx_t *pOptx );}} \\
\]

\[
\text{long generate_mk_shares ( \text{mk_src_t KeySource,}} \\
\text{ \text{UCHAR mShareKey,}} \\
\text{ \text{UCHAR nShareGen,}} \\
\text{ \text{UCHAR *pShares [ ],}} \\
\text{ \text{optx_t *pOptx );}} \\
\]

**Input**

On entry to this routine:

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

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

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

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

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

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

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

$pOptx$ is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

**Output**

On successful exit from this routine:

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

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_SRDI_OPEN_ERROR</td>
<td>Could not open the master key SRDI.</td>
</tr>
<tr>
<td>mk_SEM_CLAIM_FAILED</td>
<td>Could not access the SRDI Manager, the operation cannot be completed.</td>
</tr>
<tr>
<td>mk_INCORRECT_STATE</td>
<td>The master key is not valid.</td>
</tr>
<tr>
<td>mk_UNSUPPORTED_SCHEME</td>
<td>The values of $mShareKey$ and $nShareGen$ were inconsistent.</td>
</tr>
<tr>
<td>mk_KEY_SHARE_SPLIT_FAIL</td>
<td>An error occurred in the splitting process.</td>
</tr>
</tbody>
</table>
5.4.5 mkmGenerateRandomMK - generate random master key

*mkmGenerateRandomMK* generates a random master key, and stores the value in the NMK register for the *mk_set* being processed. This function is not available for AES master keys.

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

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

**Function prototype**

```c
long mkmGenerateRandomMK ( optx_t *pOptx, 
mk_selectors MKSelector );

long generate_random_mk ( optx_t *pOptx );
```

**Input**

On entry to this routine:

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

**Output**

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

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

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>mk_NO_ERROR</em></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><em>mk_SRDI_OPEN_ERROR</em></td>
<td>Could not open the master key SRDI.</td>
</tr>
<tr>
<td><em>mk_SEM_CLAIM_FAILED</em></td>
<td>Could not access the SRDI Manager, the operation cannot be completed.</td>
</tr>
<tr>
<td><em>mk_MKS_MISMATCH</em></td>
<td>BOTH_MK was the value in type_mks, but one or both of the new master key registers was not in the empty state.</td>
</tr>
<tr>
<td>mk_INCORRECT_STATE</td>
<td>The master key is in the incorrect state.</td>
</tr>
</tbody>
</table>
5.4.6 init_master_keys - create and initialize master keys

`init_master_keys` creates and initializes all of the master key SRDIs if they don't already exist.

**Function prototype**

```c
uint32_t init_master_keys ( optx_t *pOptx);
```

**Input**

On input to this function:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

**Output**

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

The master key SRDIs for all types of master keys are generated and initialized.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>srdi_EXISTS</td>
<td>The master keys already exist, and cannot be initialized.</td>
</tr>
<tr>
<td>srdi_GENERAL_ERROR</td>
<td>Failed to access the SRDI manager.</td>
</tr>
<tr>
<td>srdiALLOC_ERROR</td>
<td>Could not allocate memory for the master keys.</td>
</tr>
</tbody>
</table>
5.4.7 mkmLoadFirstMKPart - load first master key part

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

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

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

**Function prototype**

```c
long mkmLoadFirstMKPart ( optx_t *pOptx,
                          unsigned char *key_part,
                          mk_selectors MKSelector );

long load_first_mk_part ( optx_t *pOptx,
                         unsigned char *key_part );
```

**Input**

On entry to this routine:

- *pOptx* is a pointer to the operation tracking structure for this verb.
- *key_part* is a 24-byte clear text key part (if *type_mks* is ASYM_MK or SYM_MK or BOTH_MK) or a 32-byte clear text key part (if *type_mks* is i), which is to be stored in the NMK register.
- *MKSelector* is a parameter of type *mk_selectors*, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - *mk_set* is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
  - *mk_register* should be set to new_mk.
  - *type_mks* should be set to AES_MK if this set of master keys is intended for AES or variable length token key encryption, ASYM_MK if this set of master keys is intended for RSA key encryption, or SYM_MK if this set of master keys is used for DES key encryption. If *type_mks* is BOTH_MK and the new master key register of both the PKA and DES master key sets is empty, the key part will be loaded into both PKA and DES new master key registers; otherwise an error message will be returned.

**Output**

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

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

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_BAD_KEY_PARITY</td>
<td>The input parity has been adjusted to odd parity.</td>
</tr>
</tbody>
</table>
5.4.8  

**mkmLoadMKFromShares - load master key shares**

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

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

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

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

**Function prototype**

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

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

**Input**

On entry to this routine:

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

- *mShareKey* is the number of shares required to reconstruct the master key. This is the number of shares that are provided in the *pShares[ ]* array.
- *nShareGen* is the total number of shares that were generated for this key, by the *mkmGenerateMKShares* function.
- *pShares[ ]* is an array of pointers to the *m* shares that are used to reconstruct the master key. Each share is 25 bytes in length.

88  IBM 4767 CCA UDX Reference
Output
This function has no output. On successful exit from this routine:
The new master key is generated in the N MK register and the state of the N MK register is Full.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_SRDI_OPEN_ERROR</td>
<td>Unable to open the SRDI item.</td>
</tr>
<tr>
<td>mk_SEM_CLAIM_FAILED</td>
<td>Unable to access the SRDI manager.</td>
</tr>
<tr>
<td>mk_INCORRECT_STATE</td>
<td>The N MK register is not in the empty state.</td>
</tr>
<tr>
<td>mk_UNSUPPORTED_SCHEME</td>
<td>MshareKey or nShareGen have invalid values.</td>
</tr>
<tr>
<td>mk_KEY_SHARE_RECOVER_FAIL</td>
<td>Unable to recover the master key.</td>
</tr>
<tr>
<td>mk_VP_CALCULATE_FAIL</td>
<td>SHA calculation error.</td>
</tr>
<tr>
<td>mk_SAVE_ERROR</td>
<td>A SRDI error occurred while attempting to save the new master key in BBRAM.</td>
</tr>
<tr>
<td>mk_VP_MATCHES_EXISTING_KEY</td>
<td>Verification patterns match one of the existing master keys. The key cannot be loaded.</td>
</tr>
<tr>
<td>mk_EQUAL_KEY_HALFS</td>
<td>Two of the three parts of the new key are equal. This is a warning, no action is required.</td>
</tr>
<tr>
<td>mk_WEAK_KEY</td>
<td>One of the key parts is a weak key. The key should be regenerated before use.</td>
</tr>
</tbody>
</table>
5.4.9 mkmSetMasterKey - set master key

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

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

**Function prototype**

```c
long mkmSetMasterKey ( optx_t * pOptx,
                        mk_selectors pMKSelector );
long set_master_key ( optx_t *pOptx);
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `MKSelector` is a parameter of type `mk_selectors`, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is ignored.
  - `type_mks` must be set to `AES_MK` if this set of master keys is intended for AES or variable length token key encryption, `ASYM_MK` if this set of master keys is intended for PKA key encryption, or `SYM_MK` if this set of master keys is used for DES key encryption. If `type_mks` is `BOTH_MK` and the new master key register of both the PKA and DES master key sets is full and the value in the new master key register of both the PKA and DES new master key sets is the same, then the `current_mk` registers will be set to the `new_mk` value; otherwise an error message will be returned.

**Output**

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

The master keys have been changed.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td>mk_SRDI_OPEN_ERROR</td>
<td>Could not open the master key SRDI.</td>
</tr>
<tr>
<td>mk_SEM_CLAIM_FAILED</td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td>mk_INCORRECT_STATE</td>
<td>The new master key register is not in the Full state.</td>
</tr>
<tr>
<td>mk_SAVE_ERROR</td>
<td>Unable to save the master key into bbram.</td>
</tr>
</tbody>
</table>
5.4.10 reinit_master_keys - reinitialize master keys

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

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

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

Function prototype

uint32_t reinit_master_keys ( optx_t *pOptx );

Input
On entry into this function:

pOptx is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

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

All master keys are created and initialized.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>srdi_NO_ERROR</th>
<th>The operation was successful.</th>
</tr>
</thead>
<tbody>
<tr>
<td>srdi_GENERAL_ERROR</td>
<td>Could not open the master key SRDI.</td>
</tr>
</tbody>
</table>
5.5 Functions to check master key values and status
These functions are used to determine the contents of the master key registers.

5.5.1 Summary of functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>compute_mk_verification_pattern</td>
<td>Computes a key verification pattern. See page 94.</td>
</tr>
<tr>
<td>get_mk_verification_pattern</td>
<td>Returns the 20-byte master key verification pattern for a specified master key. See page 96.</td>
</tr>
<tr>
<td>get_master_key_status</td>
<td>Get the status of the three master key registers for the SYM_MK master key. See page 97.</td>
</tr>
<tr>
<td>mkmGetAsymVerificationPattern</td>
<td>Returns the verification pattern for asymmetric master keys. See page 96.</td>
</tr>
<tr>
<td>mkmGetMasterKeyStatus</td>
<td>Returns the status of the three master key registers for the mk_set being processed. See page 97.</td>
</tr>
</tbody>
</table>
5.5.2 compute_mk_verification_pattern - compute a verification pattern

`compute_mk_verification_pattern` computes a key verification pattern for the contents of the specified master key register. The verification pattern can be used to determine which master key (old or current) was used to encrypt a given operational key. The returned verification pattern is 20 bytes in length. For the symmetric-keys master key (`SYM_MK`), if the first and third key parts of the register specified are the same, the OS/390 ICSF algorithm for the generation of the master key verification pattern will be used. If the key parts are not the same, the verification pattern is a SHA-1 hash of the key, combined with a header. The header is a one-byte value which is used to differentiate this hash from any other hash on the master key, which might be computed for a different purpose. The value of the header byte is X'01'. For the asymmetric-keys master key (`ASYM_MK`), the MDC-4 algorithm will be used to compute the verification pattern. For the AES master key (`AES_MK`), the SHA-256 algorithm will be used to compute the verification pattern.

**Function prototype**

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

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `ver_pattern` is a pointer to a 20-byte location where the computed key verification pattern is returned.
- `mk_selector` is a pointer to a variable of type `mk_selectors` which must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is set to either `old_mk`, `current_mk`, or `new_mk`, representing the key whose verification pattern is desired. If `mk_register` is `new_mk` and the new master key register state is partially full, the returned verification pattern will be calculated on the partial key present in the register.
  - `type_mks` should be set to `AES_MK` if this set of master keys is intended for AES or variable length token key encryption, `ASYM_MK` if this set of master keys is intended for PKA key encryption or `SYM_MK` if this set of master keys is used for DES key encryption. If `type_mks` is `BOTH_MK` and the state and value of the specified master key register of both the master key sets is the same, the verification pattern of the specified register will be calculated on the Symmetric Keys master key set; otherwise, an error message will be returned.
- `vp_algorithm` must be one of `VP_ALG_DEFAULT`, `VP_ALG_MDC4`, `VP_ALG_SHA1` or `VP_ALG_SHA256`. Note that the default for PKA tokens is `VP_ALG_MDC4` (for a 16-byte MDC-4 hash), the default for DES tokens is `VP_ALG_SHA1`, and the default for AES tokens is `VP_ALG_SHA_256`.

**Output**

On successful exit from this routine:

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

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_VP_CALCULATE_FAIL</td>
<td>SHA calculation error.</td>
</tr>
<tr>
<td>mk_SEM_CLAIM_FAILED</td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td>mk_SRDI_OPEN_ERROR</td>
<td>Unable to open the SRDI item.</td>
</tr>
</tbody>
</table>
5.5.3 get_mk_verification_pattern - read a verification pattern

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

Function prototype

\[
\text{long get_mk_verification_pattern ( UCHAR *ver_pattern, mk_selectors *mk_selector );}
\]

Input

On entry to this routine:

- \text{ver_pattern} is a pointer to a 20-byte location where the master key verification pattern is returned.
- \text{mk_selector} is a pointer to a variable of type \text{mk_selectors} which must be initialized as follows:
  - \text{mk_set} is an index 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 \text{MK_SET_DEFAULT}.
  - \text{mk_register} is set to either \text{old_mk}, \text{current_mk}, or \text{new_mk}, representing the key whose verification pattern is required.
  - \text{type_mks} should be set to \text{AES_MK} if the request is to get the verification pattern of the AES/variable length token master key, \text{ASYM_MK} if the request is to get the verification pattern of the asymmetric-keys master key, or to \text{SYM_MK} if the request is to get the verification pattern of the symmetric-keys master key. If \text{type_mks} is \text{BOTH_MK} and the state and value of the specified master key register of both the master key sets is the same, the verification pattern of the specified register will be calculated on the Symmetric Keys master key set; otherwise, an error message will be returned.

Output

On successful exit from this routine:

- \text{ver_pattern} contains the verification pattern.

Return codes

Common return codes generated by this routine are:

| \text{mk_NO_ERROR} | The operation was successful. |
| \text{mk_INVALID_KEY_SELECTOR} | The input parameters are not valid. |
| \text{mk_SEM_CLAIM_FAILED} | Unable to access the SRDI Manager. |
| \text{mk_KEY_NOT_VALID} | The selected key was not in a valid state. |
5.5.4 mkmGetAsymVerificationPattern - read the asymmetric keys verification pattern

*mkmGetAsymVerificationPattern* returns the pre-computed verification pattern for a set of keys used to encrypt and decrypt PKA keys. This value is computed and saved when the master keys are first loaded, and may be used to determine which of the master keys was used to encrypt a given operational key.

**Function prototype**

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

**Input**

On entry to this routine:  

*MKSelector* is a variable of type *mk_selectors* which must be initialized as follows:

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

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

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

**Output**

On successful exit from this routine:  

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

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

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>mk_NO_ERROR</em></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><em>mk_INVALID_KEY_SELECTOR</em></td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td><em>mk_SEMCLAIM_FAILED</em></td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td><em>mk_KEY_NOT_VALID</em></td>
<td>The selected key was not in a valid state.</td>
</tr>
</tbody>
</table>
5.5.5 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.

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

**Function prototype**

```c
long mkmGetMasterKeyStatus ( mk_selector MKSelector,
                             mk_status_var *mk_status );

long get_master_key_status ( mk_status_var *mk_status );
```

**Input**

On entry to this routine:

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

- *mk_set* is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
- *mk_register* is ignored.
- *type_mks* should be set to AES_MK if the request is to get the status of the AES/variable length token master key, 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 5.

<table>
<thead>
<tr>
<th>Master Key Status Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0 (LSB)</td>
</tr>
<tr>
<td>Bit 1</td>
</tr>
<tr>
<td>Bit 2</td>
</tr>
<tr>
<td>Bit 3</td>
</tr>
<tr>
<td>Bit 4</td>
</tr>
<tr>
<td>Bit 5-7</td>
</tr>
</tbody>
</table>

Figure 5 Master Key Status Bits

*mk_status* returns a code indicating the success or failure of the operation.

98 IBM 4767 CCA UDX Reference
Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td>mk_SEMCLAIM_FAILED</td>
<td>Unable to access the SRDI Manager.</td>
</tr>
</tbody>
</table>
5.6 Functions to encrypt and decrypt using the master key

These functions in this section are used to make use of the master keys. Since there are no functions which return the master key values, these functions are the only way to use the master keys.

5.6.1 Summary of functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes_engine_with_MK</td>
<td>Encrypts a data string using the AES or APKA master key.</td>
</tr>
<tr>
<td>aes_unwrap_with_hash_and_MK</td>
<td>Decrypts a data string and associated data using the AESKW algorithm with the AES or APKA master key.</td>
</tr>
<tr>
<td>aes_unwrap_with_MK</td>
<td>Decrypts a data string and associated data using the AESKW algorithm with the AES or APKA master key.</td>
</tr>
<tr>
<td>aes_wrap_with_hash_and_MK</td>
<td>Encrypts a data string and associated data using the AESKW algorithm with the AES or APKA master key.</td>
</tr>
<tr>
<td>aes_wrap_with_MK</td>
<td>Encrypts a data string and associated data using the AESKW algorithm with the AES or APKA master key.</td>
</tr>
<tr>
<td>ede3_triple_decrypt_under_master_key</td>
<td>Triple decrypts multiple 8-byte data strings using EDE3 triple DES.</td>
</tr>
<tr>
<td>ede3_triple_encrypt_under_master_key</td>
<td>Triple encrypts multiple 8-byte data strings using EDE3 triple DES.</td>
</tr>
<tr>
<td>TDESDecryptUnderMasterKey</td>
<td>Triple-DES decrypts data using the master key.</td>
</tr>
<tr>
<td>TDESEncryptUnderMasterKey</td>
<td>Triple-DES encrypts data under the master key.</td>
</tr>
<tr>
<td>triple_decrypt_under_master_key</td>
<td>Triple-DES decrypts an 8-byte block of data.</td>
</tr>
<tr>
<td>triple_decrypt_under_master_key_with_CV</td>
<td>Triple-DES decrypts an 8-byte block of data using a control vector.</td>
</tr>
<tr>
<td>triple_encrypt_under_master_key</td>
<td>Triple-DES encrypts an 8-byte block of data.</td>
</tr>
<tr>
<td>triple_encrypt_under_master_key_with_CV</td>
<td>Triple-DES encrypts an 8-byte block of data using a control vector.</td>
</tr>
</tbody>
</table>
5.6.2 aes_engine_with_MK - encrypt or decrypt data using the AES or APKA master key

aes_engine_with_MK encrypts a variable amount of data using either the AES master key or the APKA master key.

Function prototype

```c
uint32_t aes_engine_with_MK ( optx_t *pOptx,
                             mk_selectors *pMK_selectors,
                             uint8_t *pInputText,
                             uint32_t DataLength,
                             uint8_t *pOutputText,
                             boolean cipher_flag );
```

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `pMK_selector` is a pointer to a variable which must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is set to either `old_mk`, `current_mk`, or `new_mk`, representing the key which should be used.
  - `type_mks` must be set to `AES_MK` or `APKA_MK`.
- `pInputText` is a pointer to a buffer containing a multiple of 16 bytes of data to be enciphered.
- `DataLength` is the length of data at `pClearText`.
- `pOutputText` is a pointer to a buffer at least `DataLength` bytes long.
- `cipher_flag` is true is the data is to be enciphered, or false if the data is to be deciphered.

Output

On successful exit from this routine:

- `pOutputText` contains the encrypted data if `cipher_flag` was `TRUE`, or the clear text if `cipher_flag` was `FALSE`. This buffer may be the same as the `pInputText` buffer, if desired.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td>mk_SEM_CLAIM_FAILED</td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td>mk_KEY_NOT_VALID</td>
<td>The designated master key is not in a valid state.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
</tbody>
</table>

Master key manager functions 101
5.6.3 aes_unwrap_with_hash_and_MK - decrypt using the AESKW mechanism

aes_unwrap_with_hash_and_MK decrypts a key and associated data sections using the AESKW key wrapping mechanism, using the master key. This function allows the caller to choose the hashing mechanism.

Function prototype

long aes_unwrap_with_hash_and_MK ( optx_t *pOptx,
                                    mk_selectors *pMK_selector,
                                    UCHAR *pWrappedData,
                                    uint32_t wrapDataLength,
                                    UCHAR *pAssocData,
                                    uint32_t assocDataLength,
                                    uint32_t hashType,
                                    UCHAR *pClearText,
                                    uint32_t *pClearTextLength );

Input

On entry to this routine:

pOptx is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

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

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

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

- type_mks must be set to AES_MK if the AES master key is to be used to wrap, otherwise the APKA master key will be used.

pWrappedData is a pointer to a buffer containing the wrapped data.

wrapDataLength is the length of the wrapped data.

pAssocData is a pointer to the associated data which is included in the wrapped data.

assocDataLength is the length of the buffer pointed to by the pAssocData parameter.

hashType is one of SHA_224_METHOD or SHA_256_METHOD.

pClearText is a pointer to a buffer in which to store the decrypted data.

pClearTextLength is a pointer to the length of the buffer pClearText.

Output

On successful exit from this routine:

pClearText contains the decrypted data, and pClearTextLength is the length of the data.

Return codes

Common return codes generated by this routine are:
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td>mk_SEM_CLAIM_FAILED</td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td>mk_KEY_NOT_VALID</td>
<td>The designated master key is not in a valid state.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
</tbody>
</table>
5.6.4 aes_unwrap_with_MK - decrypt using the AESKW mechanism

aes_unwrap_with_MK decrypts a key and associated data sections using the AESKW key wrapping mechanism, using the master key.

Function prototype

```c
long aes_unwrap_with_MK ( optx_t *pOptx,
    mk_selectors *pMK_selector,
    UCHAR *pWrappedData,
    uint32_t wrapDataLength,
    UCHAR *pAssocData,
    uint32_t assocDataLength,
    UCHAR *pClearText,
    uint32_t *pClearTextLength );
```

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `pMK_selector` is a pointer to a variable which must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is set to either `old_mk`, `current_mk`, or `new_mk`, representing the key which should be used.
  - `type_mks` must be set to `AES_MK` if the AES master key is to be used to wrap, otherwise the APKA master key will be used.
- `pWrappedData` is a pointer to a buffer in which contains the cryptogram.
- `pWrapDataLength` is a pointer to the length of the buffer `pWrappedData`
- `pAssocData` is a pointer to the associated data to be enciphered. Under the AESKW standard, the associated data and some constant data are included in the cryptogram in specified format.
- `assocDataLength` is the length of the buffer pointed to by the `pAssocData` parameter.
- `pClearText` is a pointer to a buffer where the data (key) to be deciphered may be stored.
- `clearTextLength` is the length of the buffer pointed to by `pClearText`.

Output

On successful exit from this routine:

- `pClearText` contains the decrypted data, and `pClearTextLength` is the length of the data.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mk_NO_ERROR</code></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><code>mk_INVALID_KEY_SELECTOR</code></td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>mk_SEM_CLAIM_FAILED</td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td>mk_KEY_NOT_VALID</td>
<td>The designated master key is not in a valid state.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
</tbody>
</table>
5.6.5  aes_wrap_with_hash_and_MK - encrypt using the AESKW mechanism

aes_wrap_with_hash_and_MK encrypts a key and associated data sections using the AESKW key wrapping mechanism, using the master key. This function allows the caller to choose the hashing mechanism.

Function prototype

```c
long aes_wrap_with_hash_and_MK ( optx_t *pOptx,
    mkSelectors *pMK_selector,
    UCHAR *pClearText,
    uint32_t clearTextLength,
    UCHAR *pAssocData,
    uint32_t assocDataLength,
    uint32_t hashType,
    UCHAR *pWrappedData,
    uint32_t *pWrapDataLength );
```

Input

On entry to this routine:

- **pOptx** is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- **pMK_selector** is a pointer to a variable which must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is set to either old_mk, current_mk, or new_mk, representing the key which should be used.
  - `type_mks` must be set to `AES_MK` if the AES master key is to be used to wrap, otherwise the APKA master key will be used.
- **pClearText** is a pointer to a buffer containing the data (key) to be enciphered.
- **clearTextLength** is the length of the clear text.
- **pAssocData** is a pointer to the associated data to be enciphered. Under the AESKW standard, the associated data and some constant data are included in the cryptogram in specified format.
- **assocDataLength** is the length of the buffer pointed to by the **pAssocData** parameter.
- **hashType** is one of **SHA_224_METHOD** or **SHA_256_METHOD**.
- **pWrappedData** is a pointer to a buffer in which to store the resulting cryptogram.
- **pWrapDataLength** is a pointer to the length of the buffer **pWrappedData**

Output

On successful exit from this routine:

- **pWrappedData** contains the cryptogram, and **pWrapDataLength** is the length of the cryptogram.
## Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td>mk_SEM_CLAIM_FAILED</td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td>mk_KEY_NOT_VALID</td>
<td>The designated master key is not in a valid state.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
</tbody>
</table>
5.6.6 aes_wrap_with_MK - encrypt using the AESKW mechanism

`aes_wrap_with_MK` encrypts a key and associated data sections using the AESKW key wrapping mechanism, using the master key.

**Function prototype**

```c
long aes_wrap_with_MK ( optx_t *pOptx,
                      mk_selectors *pMK_selector,
                      UCHAR *pClearText,
                      uint32_t clearTextLength,
                      UCHAR *pAssocData,
                      uint32_t assocDataLength,
                      UCHAR *pWrappedData,
                      uint32_t *pWrapDataLength );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

- `pMK_selector` is a pointer to a variable which must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is set to either `old_mk`, `current_mk`, or `new_mk`, representing the key which should be used.
  - `type_mks` must be set to `AES_MK` if the AES master key is to be used to wrap, otherwise the APKA master key will be used.

- `pClearText` is a pointer to a buffer containing the data (key) to be enciphered.

- `clearTextLength` is the length of the clear text.

- `pAssocData` is a pointer to the associated data to be enciphered. Under the AESKW standard, the associated data and some constant data are included in the cryptogram in specified format.

- `assocDataLength` is the length of the buffer pointed to by the `pAssocData` parameter.

- `pWrappedData` is a pointer to a buffer in which to store the resulting cryptogram.

- `pWrapDataLength` is a pointer to the length of the buffer `pWrappedData`

**Output**

On successful exit from this routine:

- `pWrappedData` contains the cryptogram, and `pWrapDataLength` is the length of the cryptogram.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mk_NO_ERROR</code></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><code>mk_INVALID_KEY_SELECTOR</code></td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td>Code</td>
<td>Message</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>mk_SEM_CLAIM_FAILED</td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td>mk_KEY_NOT_VALID</td>
<td>The designated master key is not in a valid state.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
</tbody>
</table>
5.6.7 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

```c
long ede3_triple_decrypt_under_master_key (  
    optx_t *pOptx,  
    mk_selectors *mk_selector,  
    UCHAR *cleartext,  
    UCHAR *ciphertext,  
    ULONG data_length );
```

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `mk_selector` is a pointer to a variable which must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is set to either `old_mk`, `current_mk`, or `new_mk`, representing the key which should be used.
  - `type_mks` should be set to `ASYM_MK` if the data is to be decrypted with the asymmetric-keys master key or to `SYM_MK` if the data is to be decrypted with the symmetric-keys master key.

- `cleartext` is a pointer to a buffer large enough to store the clear text. This may be the same as the ciphertext buffer.
- `ciphertext` is a pointer to a buffer containing the data to be deciphered.
- `data_length` is the number of bytes of data to be deciphered. This value must be a multiple of eight.

Output

On successful exit from this routine:

- `cleartext` contains the deciphered data. This buffer may be the same as the ciphertext buffer, if desired.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td>mk_INVALID_DATA_LENGTH</td>
<td>The data length is not a multiple of 8.</td>
</tr>
<tr>
<td>mk_SRDI_OPEN_ERROR</td>
<td>Could not open the master key SRDI.</td>
</tr>
<tr>
<td>mk_SEM_CLAIM_FAILED</td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td>mk_KEY_NOT_VALID</td>
<td>The designated master key is not in a valid state.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
</tbody>
</table>
5.6.8 **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**

```c
long ede3_triple_encrypt_under_master_key (
    optx_t *pOptx,
    mk_selectors *mk_selector,
    UCHAR *cleartext,
    UCHAR *ciphertext,
    ULONG data_length );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

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

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

- `ciphertext` is a pointer to a buffer large enough to store the ciphertext. This buffer may be the same as the `cleartext` buffer.

- `data_length` is the number of bytes of data to be enciphered. This value must be a multiple of eight.

**Output**

On successful exit from this routine:

- `ciphertext` contains the enciphered data. This buffer may be the same as the `cleartext` buffer, if desired.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mk_NO_ERROR</code></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><code>mk_INVALID_KEY_SELECTOR</code></td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td><code>mkINVALID_DATA_LENGTH</code></td>
<td>The data length is not a multiple of 8.</td>
</tr>
<tr>
<td><code>mk_SRDI_OPEN_ERROR</code></td>
<td>Could not open the master key SRDI.</td>
</tr>
<tr>
<td><code>mkSEM_CLAIM_FAILED</code></td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td><code>mkKEY_NOT_VALID</code></td>
<td>The designated master key is not in a valid state.</td>
</tr>
<tr>
<td><code>mkINVALID_KEY_SELECTOR</code></td>
<td>The input parameters are not valid.</td>
</tr>
</tbody>
</table>

Master key manager functions 113
5.6.9 mkmTDESDecryptUnderMasterKey

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

**Function prototype**

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

**Input**

On entry to this routine:

`pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

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

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

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

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

`text_length` is the number of bytes of data in the `cipher_text` buffer. This is also the number of bytes of encrypted text returned in the `clear_text` buffer, and it must be a multiple of 8.

**Output**

On successful exit from this routine:

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

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mk_NO_ERROR</code></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><code>mk_INVALID_KEY_SELECTOR</code></td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td><code>mk_INVALID_DATA_LENGTH</code></td>
<td>The data length is not a multiple of 8.</td>
</tr>
<tr>
<td><code>mk_SRDI_OPEN_ERROR</code></td>
<td>Could not open the master key SRDI.</td>
</tr>
<tr>
<td><code>mk_SEM_CLAIM_FAILED</code></td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td>mk_KEY_NOT_VALID</td>
<td>The designated master key is not in a valid state.</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
</tbody>
</table>
5.6.10  mkmTDESEncryptUnderMasterKey

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

**Function prototype**

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

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `MKSelector` is a pointer to a variable which must be initialized as follows:
  - *mk_set* is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - *mk_register* is set to either `old_mk`, `current_mk`, or `new_mk`, representing the key which should be used.
  - *type_mks* should be set to `ASYM_MK` if the data is to be encrypted with the asymmetric-keys master key or to `SYM_MK` if the data is to be encrypted with the symmetric-keys master key.
- `clear_text` is a pointer to the text which you wish to encipher. This text must be a multiple of 8 bytes long.
- `cipher_text` is a pointer to a buffer in which to store the encrypted text. This buffer must be at least as long as the `clear_text`.
- `text_length` is the number of bytes of data in the `clear_text` buffer. This is also the number of bytes of encrypted text returned in the `cipher_text` buffer.

**Output**

On successful exit from this routine:

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

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mk_NO_ERROR</code></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><code>mk_INVALID_KEY_SELECTOR</code></td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td><code>mk_INVALID_DATA_LENGTH</code></td>
<td>The data length is not a multiple of 8.</td>
</tr>
<tr>
<td><code>mk_SRDI_OPEN_ERROR</code></td>
<td>Could not open the master key SRDI.</td>
</tr>
<tr>
<td><code>mk_SEM_CLAIM_FAILED</code></td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td><code>mk_KEY_NOT_VALID</code></td>
<td>The designated master key is not in a valid state.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
</tbody>
</table>
5.6.11 triple_decrypt_under_master_key

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

**Function prototype**

```c
long triple_decrypt_under_master_key ( optx_t *pOptx, 
    mk_selectors *mk_selector, 
    UCHAR *ciphertext, 
    UCHAR *cleartext );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `mk_selector` is a pointer to a variable which must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is set to either `old_mk`, `current_mk`, or `new_mk`, representing the key which should be cleared.
  - `type_mks` should be set to `ASYM_MK` if the data is to be decrypted with the asymmetric-keys master key or to `SYM_MK` if the data is to be decrypted with the symmetric-keys master key.

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

**Output**

On successful exit from this routine:

- `cleartext` contains the deciphered data. This buffer may be the same as the `ciphertext` buffer, if desired.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mk_NO_ERROR</code></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><code>mk_INVALID_KEY_SELECTOR</code></td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td><code>mk_INVALID_DATA_LENGTH</code></td>
<td>The data length is not a multiple of 8.</td>
</tr>
<tr>
<td><code>mk_SRDI_OPEN_ERROR</code></td>
<td>Could not open the master key SRDI.</td>
</tr>
<tr>
<td><code>mk_SEM_CLAIM_FAILED</code></td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td><code>mk_KEY_NOT_VALID</code></td>
<td>The designated master key is not in a valid state.</td>
</tr>
<tr>
<td><code>mk_INVALID_KEY_SELECTOR</code></td>
<td>The input parameters are not valid.</td>
</tr>
</tbody>
</table>
5.6.12 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. Before decrypting the key, this function attempts to find the key in the DES key cache. If it is not present, the clear key will be added to the cache after decryption is performed.

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

Function prototype

```c
long triple_decrypt_under_master_key_with_CV ( optx_t *pOptx,
                                             mk_selectors *mk_selector,
                                             eightbyte *cv,
                                             UCHAR *ciphertext,
                                             UCHAR *cleartext );
```

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `mk_selector` is a pointer to a variable which must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is set to either `old_mk`, `current_mk`, or `new_mk`, representing the key which should be used.
  - `type_mks` should be set to `ASYM_MK` if the data is to be decrypted with the asymmetric-keys master key or to `SYM_MK` if the data is to be decrypted with the symmetric-keys master key.
- `cv` is a pointer to a double-length CCA control vector, which is exclusive-ORed with the specified key value before the key is used.
- `ciphertext` is a pointer to a buffer containing the data to be deciphered.
- `cleartext` is a pointer to a buffer 8 bytes in length. This may be the same as the `ciphertext` buffer.

Output

On successful exit from this routine:

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

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mk_NO_ERROR</code></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><code>mk_INVALID_KEY_SELECTOR</code></td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td><code>mk_SRDI_OPEN_ERROR</code></td>
<td>Could not open the master key SRDI.</td>
</tr>
<tr>
<td><code>mk_SEMCLAIM_FAILED</code></td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>mk_KEY_NOT_VALID</td>
<td>The designated master key is not in a valid state.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
</tbody>
</table>
5.6.13 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

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

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

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

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

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

Output

On successful exit from this routine:

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

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td>mk_SRDI_OPEN_ERROR</td>
<td>Could not open the master key SRDI.</td>
</tr>
<tr>
<td>mk_SEM_CLAIM_FAILED</td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td>mk_KEY_NOT_VALID</td>
<td>The designated master key is not in a valid state.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
</tbody>
</table>
5.6.14  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.

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

Function prototype

```c
long triple_encrypt_under_master_key_with.CV (
    optx_t *pOptx,
    UCHAR *mk_selector,
    eightbyte *cv,
    UCHAR *cleartext,
    UCHAR *ciphertext);
```

Input

On entry to this routine:

pOptx is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

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

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

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

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

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

Output

On successful exit from this routine:

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

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mk_NO_ERROR</code></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><code>mk_INVALID_KEY_SELECTOR</code></td>
<td>The input parameters are not valid.</td>
</tr>
<tr>
<td><code>mk_SRDI_OPEN_ERROR</code></td>
<td>Could not open the master key SRDI.</td>
</tr>
<tr>
<td><code>mk_SEM_CLAIM_FAILED</code></td>
<td>Unable to access the SRDI Manager.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>mk_KEY_NOT_VALID</td>
<td>The designated master key is not in a valid state.</td>
</tr>
<tr>
<td>mk_INVALID_KEY_SELECTOR</td>
<td>The input parameters are not valid.</td>
</tr>
</tbody>
</table>
6 Hashing 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, SHA-224, and SHA-256) as defined in FIPS Publication 180-1 and 180-2.

Each function is labeled with regard to whether it is available on the workstation host, the coprocessor, or both.

6.1 Header files for SHA functions

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

```c
#include "cmncryt2.h" /* Cryptographic types */
#include "cmn_sha.h"  /* SHA external definitions */
#include "cmn_sha2.h" /* SHA 256 and SHA 224 external defs */
```

6.2 Summary of functions

The following functions are described in this chapter:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComputeHMAC_SHA1</td>
<td>Compute a keyed-hash for message authentication code (HMAC) for a given set of data using the Secure Hash Algorithm (SHA-1).</td>
</tr>
<tr>
<td>do_sha_hash_message</td>
<td>Compute the hash of a block of data using the SHA-1 algorithm.</td>
</tr>
<tr>
<td>do_sha_hash_msg_to_bfr</td>
<td>&quot;wrapper&quot; for do_sha_hash_message.</td>
</tr>
<tr>
<td>hw_sha_hash_message</td>
<td>Compute a SHA-1 hash of the requested data on the hashing hardware.</td>
</tr>
<tr>
<td>hw_sha2_hash_message</td>
<td>Compute a SHA-256, SHA-224, SHA-384 or SHA-512 hash of the requested data using the hashing hardware.</td>
</tr>
<tr>
<td>sha_hash_message</td>
<td>Compute the hash of a block of data using the SHA-1 algorithm, in software.</td>
</tr>
<tr>
<td>sha_hash_msg_to_bfr</td>
<td>&quot;wrapper&quot; for sha_hash_message.</td>
</tr>
<tr>
<td>sha224_first</td>
<td>Begin the computation of the hash of a block of data using the SHA-224 algorithm in software.</td>
</tr>
<tr>
<td>sha224_last</td>
<td>Finish the computation of the hash of a block of data using the SHA-224 algorithm in software.</td>
</tr>
<tr>
<td>sha224_middle</td>
<td>Continue computing the hash of a block of data using the SHA-224 algorithm in software.</td>
</tr>
<tr>
<td>sha224_only</td>
<td>Compute the hash of a complete block of data using the SHA-224 algorithm in software.</td>
</tr>
<tr>
<td>sha256_first</td>
<td>Begin the computation of the hash of a block of data using the SHA-256 algorithm in software.</td>
</tr>
<tr>
<td>sha256_last</td>
<td>Finish the computation of the hash of a block of data using the SHA-256 algorithm in software.</td>
</tr>
<tr>
<td>sha_256_middle</td>
<td>Continue computing the hash of a block of data using the SHA-256 algorithm in software.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>sha_256_only</td>
<td>Compute the hash of a complete block of data using the SHA-256 algorithm in software.</td>
</tr>
</tbody>
</table>

### 6.3 Chained operations

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

A chained operation with SHA-1 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 `do_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 `do_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 `do_sha_hash_message` that processed the previous piece of the block. The function hashes the piece and updates `pContext` and `pHash` appropriately.

A chained operation with SHA-224 is initiated by calling `sha224_first`. On return, `state` contains data that must be preserved and passed to `sha224_middle` (or `sha224_last`). A chained operation with SHA-256 is processed similarly.

### Examples

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

```c
BitCount = ( (uint64_t) 150) *8;
memset (( UCHAR *) pContext, 0x00 , sizeof( sha_context) ) ;
do_sha_hash_message ( pBlock, & Hash, & Bitcount, pContext, only) ;
```

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

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

BitCount = (uint64_t) 512;
memset ( ( 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; i++ )
{
    /* it is important that the value in BitCount is divisible by 512 */
    do_sha_hash_message( employee[ i ].name, &Hash,
```

126 Hashing functions
&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 */
6.3.1 computeHMAC_SHA1 - compute HMAC using SHA-1 algorithm

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

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

**Function prototype**

```c
uint32_t computeHMAC_SHA1 ( uint8_t *pBuffer,
                            uint32_t buffer_length,
                            uint8_t *pKey,
                            uint32_t key_length,
                            uint8_t *pHmac,
                            uint32_t hmacLen );
```

**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.
- `hmacLen` is 20, the length of the expected HMAC.

**Output**

On successful exit from this routine:

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

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh_NO_ERROR (0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>E_TXT_LEN</td>
<td>The <code>hmacLen</code> was larger than 20.</td>
</tr>
</tbody>
</table>
6.3.2  computeHMAC_SHA256 - compute HMAC using SHA-256 algorithm
This function is available on both the workstation host and on the coprocessor.

computeHMAC_SHA256 computes a keyed-hash for message authentication code (HMAC) for a given set of data, using the SHA-256 algorithm and a provided key.

**Function prototype**

```c
uint32_t computeHMAC_SHA256 ( uint8_t *pBuffer,
                             uint32_t buffer_length,
                             uint8_t *pKey,
                             uint32_t key_length,
                             uint8_t *pHmac,
                             uint32_t hmacLen );
```

**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 32 bytes long, which will hold the returned data.
- `hmacLen` is 32, the length of the expected HMAC.

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

- `pHmac` contains the HMAC-SHA256 of the data in `pBuffer`.

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>RT_SWERR</td>
<td>The <code>hmacLen</code> was larger than 64.</td>
</tr>
</tbody>
</table>
6.3.3 do_sha_hash_message - calculate SHA-1 hash in hardware/software

This function is available only on the coprocessor.

`do_sha_hash_message` calculates the SHA-1 hash of the data using the coprocessor hardware if the message length is more than 6144 bits (768 bytes). Otherwise, the hash is calculated in software.

**Function prototype**

```
ULONG do_sha_hash_message ( optx_t *pOptx, UCHAR *pBlock, UCHAR *pHash, uint64_t *pBitCount, sha_context *pContext, owh_sequence MsgPart);
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

- `MsgPart` controls the operation of the function and must be one of the following constants:

<table>
<thead>
<tr>
<th><code>MsgPart</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>only</code></td>
<td>The input data constitutes the entire block of data to be hashed. The hash value is computed and returned.</td>
</tr>
<tr>
<td><code>first</code></td>
<td>The input data constitutes the initial portion of a block of data to be hashed. See “Chained operations” on page 126 for details.</td>
</tr>
<tr>
<td><code>middle</code></td>
<td>The input data constitutes an additional portion of a block of data to be hashed. See “Chained operations” on page 126 for details.</td>
</tr>
<tr>
<td><code>final</code></td>
<td>The input data constitutes the final portion of a block of data to be hashed. See “Chained operations” on page 126 for details.</td>
</tr>
</tbody>
</table>

- `pBlock` must contain the address of the block of data that is to be incorporated into the hash.
- `pHash` must contain the address of a buffer to which the hash value may be written. The buffer must be at least 20 bytes long. `pHash` is used only if `MsgPart` specifies `only` or `final`.
- `pBitCount` must contain the address of a buffer that contains the length in bits of the block of data referenced by `pBlock`.
  - If `MsgPart` specifies `first` or `middle`, `pBitCount` must be a multiple of 512, or data will be lost.
- `pContext` must contain the address of a context buffer from which the function may initialize its internal state and to which the function may write its final internal state. See “Chained operations” on page 126 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 specified only or final. In the latter case, the hash value incorporates the initial hash value provided in *pContext.

*pContext has been updated to incorporate changes to the function's internal state caused by incorporating *pBlock into the hash.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh_NO_ERROR (0)</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>sh_MSG_PART_INVALID</td>
<td>The MsgPart argument was not only, first, middle, or final.</td>
</tr>
</tbody>
</table>
6.3.4 do_sha_hash_msg_to_bfr - SHA-1 hash

This function is available only on the coprocessor.

do_sha_hash_msg_to_bfr is a wrapper for do_sha_hash_message that simplifies the interface when chained operations (see “Chained operations” on page 126) are not necessary.

Function prototype

```c
void do_sha_hash_msg_to_bfr ( optx_t *pOptx,
    UCHAR *pBlock,
    UCHAR *pHash,
    uint64_t *pBitCount );
```

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `pBlock` must contain the address of the block of data that is to be hashed.
- `pHash` must contain the address of a buffer to which the hash value may be written. The buffer must be at least 20 bytes long.
- `pBitCount` must contain the address of a buffer that contains the length in bits of the block of data referenced by `pBlock`.

Output

On successful exit from this routine:

The buffer referenced by `pHash` contains the hash value of the input data.

Notes

Function Wraps `do_sha_hash_message`

```c
    do_sha_hash_msg_to_bfr( pBlock, pHash, pBitCount);
```

performs the same function as

```c
{  
    sha_context Context;  
    memset( &Context, 0x00, sizeof(Context) );  
    do_sha_hash_message(pBlock, pHash, pBitCount, &Context, only);  
}
```

Return codes

This function has no return codes.
6.3.5  hw_sha_hash_message - compute SHA-1 hash in hardware

This function is available only on the coprocessor.

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

Function prototype

```c
void hw_sha_hash_message( optx_t *pOptx,
    UCHAR *pText,
    uint32_t text_length,
    sha_context *chain_vector,
    long *pMsg );
```

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls 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, whose fields should be initialized as follows:
  - `sha_state` must be one of `sha_state_FIRST`, `sha_state_MIDDLE`, or `sha_state_FINAL`.
  - If `sha_state` is `sha_state_FIRST`, the remainder of the `chain_vector` should be initialized to zeros.
  - Otherwise, the other fields in this variable should be returned to the next call without change.
- `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. This variable should be passed unchanged to the next call, except for the `sha_state` field.
- If this was the final call to the function, `chain_vector->sha_hash` contains the final hash value, if this call was with the final or only data block.
- `pMsg` contains the return code for the function.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>SHA1_DATA64_ERROR</td>
<td>The options argument specified <code>SHA_MSPART_FIRST</code> or <code>SHA_MSPART_MIDDLE</code> but the length of the data to be processed is not a multiple of 64.</td>
</tr>
<tr>
<td>SHA1_FINAL_ERROR</td>
<td>An error occurred while attempting to pad the input data as directed by the SHA-1 algorithm or while attempting to hash the pad bytes.</td>
</tr>
</tbody>
</table>
6.3.6  hw_sha2_hash_message - compute SHA-2 hash in hardware

This function is available only on the coprocessor.

(hw_sha2_hash_message) computes a SHA-224, SHA-256, SHA-384, or SHA-512 hash of the requested data on the hashing hardware in the coprocessor.

Function prototype

```c
void hw_sha2_hash_message ( void *pOptx,
uint8_t *pText,
uint32_t text_length,
struct SHA256 *chain_vector,
uint32_t options,
LONG *pMsg );
```

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls 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. On the final call, this is the hash.
- `options` must include one of the following chaining constants:
  - SHA2_MSGPART_FIRST
  - SHA2_MSGPART_MIDDLE
  - SHA2_MSGPART_FINAL
  - SHA2_MSGPART_ONLY

exclusive-ORed to one of the following method constants:

- SHA224_METHOD
- SHA256_METHOD
- SHA384_METHOD
- SHA512_METHOD

`final_data` is a pointer to a buffer to hold the result of the hash operation. Its minimum length is determined by the value in the `options` variable.

<table>
<thead>
<tr>
<th>options</th>
<th>Minimum Buffer Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA224_METHOD</td>
<td>28 bytes</td>
</tr>
<tr>
<td>SHA256_METHOD</td>
<td>32 bytes</td>
</tr>
<tr>
<td>SHA384_METHOD</td>
<td>48 bytes</td>
</tr>
<tr>
<td>SHA512_METHOD</td>
<td>64 bytes</td>
</tr>
</tbody>
</table>
pMsg is a pointer to a 4-byte block.

Output
On successful exit from this routine:

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

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

*pMsg* contains the return code for the function.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>ICSF_IV_CV</td>
<td>The options argument specified <code>SHA2_MSGPART_MIDDLE</code> or <code>SHA2_MSGPART_FINAL</code> but the <em>chain_vector</em> parameter failed consistency checking.</td>
</tr>
<tr>
<td>E_TXT_LEN</td>
<td>The options argument specified <code>SHA2_MSGPART_FIRST</code> or <code>SHA2_MSGPART_MIDDLE</code> but the <code>text_length</code> is zero.</td>
</tr>
<tr>
<td>RT_TXT_LEN_NO8</td>
<td>The options argument specified <code>SHA2_MSGPART_FIRST</code> or <code>SHA2_MSGPART_MIDDLE</code> but the <code>text_length</code> is not a multiple of the algorithm block size.</td>
</tr>
</tbody>
</table>
6.3.7 sha512_msg_to_buf - SHA-512 hash
This function is available only on the coprocessor.

sha512_msg_to_buf computes the hash of a single contiguous block of data using the Secure Hash Algorithm (SHA-2). This function calculates the hash in hardware.

Function prototype

    ULONG sha512_msg_to_buf ( void *pOptx,
                              uint8_t *pBlock,
                              uint8_t *pHash,
                              uint32_t ByteLen );

Input
On entry to this routine:

  pOptx is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

  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 64 bytes long.

  ByteLen must contain the length in bytes of the block of data referenced by pBlock.

Output
On successful exit from this routine:

The buffer referenced by pHash contains the hash value of the input data,

The function returns the value of the return code from the xcSHA2 function call.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMGood</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>DMBadParm</td>
<td>The pHash buffer was not writeable or the pBlock buffer was not readable.</td>
</tr>
</tbody>
</table>

Refer to xc_err.h for a comprehensive list of return codes.
6.3.8 sha_hash_message - SHA-1 hash with chaining
This function is available on both the workstation host and the coprocessor.

sha_hash_message computes the hash of a block of data using the Secure Hash Algorithm (SHA-1) and optionally incorporates the result into an initial hash value. This function calculates the hash in software.

Function prototype

ULONG sha_hash_message ( UCHAR *pBlock,
                         UCHAR *pHash,
                         uint64_t *pBitCount,
                         sha_context *pContext,
                         owh_sequence MsgPart );

Input
On entry to this routine:

pBlock must contain the address of the block of data that is to be incorporated into the hash.

pHash must contain the address of a buffer to which the hash value may be written. The buffer must be at least 20 bytes long. pHash is used only if MsgPart specifies only or final.

pBitCount must contain the address of a buffer that contains the length in bits of the block of data referenced by pBlock.

• If MsgPart specifies first or middle, pBitCount must be a multiple of 512, or data will be lost.

pContext must contain the address of a context buffer from which the function may initialize its internal state and to which the function may write its final internal state. See “Chained operations” on page 126 for details.

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 126 for details.

• middle The input data constitutes an additional portion of a block of data to be hashed. See “Chained operations” on page 126 for details.

• final The input data constitutes the final portion of a block of data to be hashed. See “Chained operations” on page 126 for details.

Note: If MsgPart specifies only or first, the initial value of pContext is ignored.

Output
On successful exit from this routine:

The buffer referenced by pHash contains the hash value of the input data if MsgPart specifies only or final. In the latter case, the hash value incorporates the initial hash value provided in pContext.

pContext has been updated to incorporate changes to the function's internal state caused by incorporating pBlock into the hash.
**Return codes**
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>SHA1_DATA64_ERROR</td>
<td>The options argument specified <code>SHA_MSGPART_FIRST</code> or <code>SHA_MSGPART_MIDDLE</code> but the length of the data to be processed is not a multiple of 64.</td>
</tr>
</tbody>
</table>
6.3.9 sha_hash_msg_to_bfr - SHA-1 hash
This function is available on both the workstation host and the coprocessor.

*sha_hash_msg_to_bfr* is a wrapper for *sha_hash_message* that simplifies the interface when chained operations (see "Chained operations" on page 126) are not necessary.

**Function prototype**

```c
void sha_hash_msg_to_bfr( UCHAR *pBlock,
                          UCHAR *pHash,
                          uint64_t *pBitCount );
```

**Input**

On entry to this routine:

*`pBlock`* must contain the address of the block of data that is to be hashed.

*`pHash`* must contain the address of a buffer to which the hash value may be written. The buffer must be at least 20 bytes long.

*`pBitCount`* must contain the address of a buffer that contains the length in bits of the block of data referenced by *`pBlock`*.

**Output**

On successful exit from this routine:

The buffer referenced by *`pHash`* contains the hash value of the input data.

**Notes**

Function Wraps *sha_hash_message*

```
    sha_hash_msg_to_bfr(pBlock,pHash,pBitCount);
```

performs the same function as

```
    {
        sha_context Context;
        memset( &Context, 0x00, sizeof(Context) );
        sha_hash_message(pBlock, pHash, pBitCount, &Context, only);
    }
```

**Return codes**

This function has no return codes.
6.3.10 sha224_first
This function is available on both the workstation host and the coprocessor.

*sha224_first* begins the calculation of a SHA-224 hash on a block of data, using software, not hardware.

**Function prototype**

```c
int sha224_first ( void *state,
                  const unsigned char *data,
                  size_t len );
```

**Input**

On entry to this routine:

*state* is a pointer to a buffer of at least 128 bytes.

*data* is a pointer to the data to be hashed.

*len* is the length in bytes of *data*, and is a multiple of 64.

**Output**

On successful exit from this routine:

*state* contains context information which must be passed unchanged to the function (either *sha224_middle* or *sha224_last*) which will calculate the next part of the hash.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK (0)</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>ERROR (-1)</td>
<td>An error was encountered.</td>
</tr>
</tbody>
</table>
6.3.11 sha224_last
This function is available on both the workstation host and the coprocessor.

sha224_last finishes the calculation of a SHA-224 hash on a block of data, using software.

Function prototype

```c
int sha224_last ( void *state,
                  const unsigned char *data,
                  size_t len,
                  unsigned char *digest );
```

Input
On entry to this routine:

- `state` is a pointer to a buffer of at least 128 bytes, which contains the data returned from a previous call to `sha224_first` or `sha224_middle`.
- `data` is a pointer to the data to be hashed. If `len` is 0, `data` may be NULL.
- `len` is the length in bytes of `data`.
- `digest` is a pointer to a buffer at least 28 bytes long.

Output
On successful exit from this routine:

- `digest` contains the 28-byte hash of the data.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK (0)</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>ERROR (-1)</td>
<td>An error was encountered.</td>
</tr>
</tbody>
</table>
6.3.12 sha224_middle
This function is available on both the workstation host and the coprocessor.

sha224_middle continues the calculation of a SHA-224 hash on a block of data, using software.

Function prototype

```c
int sha224_middle ( void *state,
                   const unsigned char *data,
                   size_t len );
```

Input

On entry to this routine:

- `state` is a pointer to a buffer of at least 128 bytes, which contains the data returned from a previous call to `sha224_first` or `sha224_middle`.
- `data` is a pointer to the data to be hashed.
- `len` is the length in bytes of `data`, and is a multiple of 64.

Output

On successful exit from this routine:

- `state` contains context information which must be passed unchanged to the function (either `sha224_middle` or `sha224_last`) which will calculate the next part of the hash.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK (0)</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>ERROR (-1)</td>
<td>An error was encountered.</td>
</tr>
</tbody>
</table>
6.3.13  sha224_only
This function is available on both the workstation host and the coprocessor.
sha224_only calculates a SHA-224 hash on a block of data using software.

Function prototype

```c
int sha224_only ( const unsigned char *data,
                 size_t len,
                 unsigned char * digest );
```

Input
On entry to this routine:

- `data` is a pointer to the data to be hashed.
- `len` is the length in bytes of `data`, and is a multiple of 64.
- `digest` is a pointer to a buffer at least 28 bytes long.

Output
On successful exit from this routine:

- `digest` contains the 28-byte calculated hash.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>OK (0)</th>
<th>The function completed successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR (-1)</td>
<td>An error was encountered.</td>
</tr>
</tbody>
</table>
6.3.14 sha256_first

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

shaft6_first begins the calculation of a SHA-256 hash on a block of data, using software.

Function prototype

```c
int sha256_first ( void *state,
                  const unsigned char *data,
                  size_t len );
```

Input

On entry to this routine:

- `state` is a pointer to a buffer of at least 128 bytes.
- `data` is a pointer to the data to be hashed.
- `len` is the length in bytes of `data`, and is a multiple of 64.

Output

On successful exit from this routine:

- `state` contains context information which must be passed unchanged to the function (either `sha256_middle` or `sha256_last`) which will calculate the next part of the hash.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK (0)</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>ERROR (-1)</td>
<td>An error was encountered.</td>
</tr>
</tbody>
</table>
6.3.15  sha256_last
This function is available on both the workstation host and the coprocessor.
sha256_last finishes the calculation of a SHA-256 hash on a block of data, using software.

**Function prototype**

```c
int sha256_last ( void *state,
                  const unsigned char *data,
                  size_t len,
                  unsigned char *digest );
```

**Input**

On entry to this routine:

- `state` is a pointer to a buffer of at least 128 bytes, which contains the data returned from a previous call to `sha256_first` or `sha256_middle`.
- `data` is a pointer to the data to be hashed. If `len` is 0, `data` may be NULL.
- `len` is the length in bytes of `data`.
- `digest` is a pointer to a buffer at least 32 bytes long.

**Output**

On successful exit from this routine:

- `digest` contains the 32-byte hash of the data.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK (0)</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>ERROR (-1)</td>
<td>An error was encountered.</td>
</tr>
</tbody>
</table>
6.3.16 sha256_middle
This function is available on both the workstation host and the coprocessor.

*sha256_middle* continues the calculation of a SHA-224 hash on a block of data, using software.

**Function prototype**

```c
int sha256_middle ( void *state,
                    const unsigned char *data,
                    size_t len );
```

**Input**

On entry to this routine:

- `state` is a pointer to a buffer of at least 128 bytes, which contains the data returned from a previous call to *sha256_first* or *sha256_middle*.
- `data` is a pointer to the data to be hashed.
- `len` is the length in bytes of `data`, and is a multiple of 64.

**Output**

On successful exit from this routine:

- `state` contains context information which must be passed unchanged to the function (either *sha256_middle* or *sha256_last*) which will calculate the next part of the hash.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK (0)</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>ERROR (-1)</td>
<td>An error was encountered.</td>
</tr>
</tbody>
</table>
6.3.17  sha256_only
This function is available on both the workstation host and the coprocessor.
sha256_only calculates a SHA-256 hash on a block of data, using software.

Function prototype

```
    int sha256_only ( const unsigned char *data,
                       size_t len,
                       unsigned char * digest );
```

Input
On entry to this routine:

data is a pointer to the data to be hashed.
len is the length in bytes of data, and is a multiple of 64.
digest is a pointer to a buffer at least 32 bytes long.

Output
On successful exit from this routine:
digest contains the 32-byte calculated hash.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK (0)</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>ERROR (-1)</td>
<td>An error was encountered.</td>
</tr>
</tbody>
</table>
7 Symmetric key 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 IBM 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, generate, or store verbs. Bits 4 and 5 of the control vector will not be checked by any standard CCA code. This allows developers to use these three bits to indicate their own, UDX specific keys, which can be used only by UDX verbs. These verbs must be written to test the required bits.

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

Note: AES keys do not use control vectors.

7.1 Header files for DES and AES utility functions

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

```c
#include "cmncryt2.h" /* T2 definitions */
#include "castyped.h" /* Adapter structures */
#include "cassub.h" /* DES 96and AES prototypes */
#include "casfunct.h"
#include "casfunct.h" /* TDES prototypes */
#include "casaes.h" /* AES prototypes */
```

7.2 Summary of functions

DES and AES utility routines include the following functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes_engine_decrypt</td>
<td>CBC deciphers using the AES hardware.</td>
</tr>
<tr>
<td>aes_engine_encrypt</td>
<td>CBC enciphers using the AES hardware.</td>
</tr>
<tr>
<td>aes_unwrap</td>
<td>Decrypts a string using the AESKW algorithm.</td>
</tr>
<tr>
<td>aes_unwrap_with_hash</td>
<td>Decrypts a string using the AESKW algorithm and a specified hash type.</td>
</tr>
<tr>
<td>aes_wrap</td>
<td>Encrypts a string using the AESKW algorithm.</td>
</tr>
<tr>
<td>aes_wrap_with_hash</td>
<td>Encrypts a string using the AESKW algorithm and a specified hash type.</td>
</tr>
<tr>
<td>cas_adjust_parity</td>
<td>Adjusts the parity of a clear DES key.</td>
</tr>
<tr>
<td>cas_aes_key_token_check</td>
<td>Verifies the integrity of an AES key token.</td>
</tr>
<tr>
<td>cas_build_default_cv</td>
<td>Builds a default control vector.</td>
</tr>
<tr>
<td>cas_current_mkvp</td>
<td>Returns the current master key verification pattern from the default master key set.</td>
</tr>
<tr>
<td>cas_des_key_token_check</td>
<td>Verifies the integrity of a DES key token.</td>
</tr>
<tr>
<td>cas_get_key_type</td>
<td>Returns the type of DES key token.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>cas_key_length</td>
<td>Determines the length of a DES key from its control vector.</td>
</tr>
<tr>
<td>cas_key_tokentvv_check</td>
<td>Verifies a DES key token validation value.</td>
</tr>
<tr>
<td>cas_master_key_check</td>
<td>Performs a master key version check using the default master key set.</td>
</tr>
<tr>
<td>cas_old_mkvp</td>
<td>Returns the old master key verification pattern from the default master key set.</td>
</tr>
<tr>
<td>cas_parity_odd</td>
<td>Determines whether a DES key has odd parity.</td>
</tr>
<tr>
<td>CasAesMasterKeyCheck</td>
<td>Performs a master key version check.</td>
</tr>
<tr>
<td>CasBuildCv</td>
<td>Builds a default control vector.</td>
</tr>
<tr>
<td>cas_old_mkvp</td>
<td>Returns the old master key verification pattern from the default master key set.</td>
</tr>
<tr>
<td>CasBuildToken</td>
<td>Builds a DES or AES 64-byte skeleton key token.</td>
</tr>
<tr>
<td>CasCurrentMkvp</td>
<td>Returns the current master key verification pattern.</td>
</tr>
<tr>
<td>CasMasterKeyCheck</td>
<td>Performs a master key version check.</td>
</tr>
<tr>
<td>CasOldMkvp</td>
<td>Returns the old master key verification pattern.</td>
</tr>
<tr>
<td>des_engine_ede2_triple_decrypt</td>
<td>TDES decrypt, using a double length key.</td>
</tr>
<tr>
<td>des_engine_ede2_triple_encrypt</td>
<td>TDES encrypt, using a double length key.</td>
</tr>
<tr>
<td>S390KeyLength</td>
<td>Determine the key length (SINGLE, DOUBLE, or TRIPLE) of a DES key token.</td>
</tr>
</tbody>
</table>

### 7.3 Overview

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

Refer to the *IBM CCA Basic Services Reference and Guide* for more information. Chapter 5, “AES and DES Symmetric-Key Management” includes an in-depth discussion of AES and DES key token management within CCA. You can also refer to Appendix B, “Data Structures” for a description of the AES and DES key token structures and Appendix C, “CCA Control Vector Definitions and Key Encryption” for a discussion of control vectors.

#### 7.3.1 Key types

Keys used in these functions are one of the following **KEY_TYPES**:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA_KEY</td>
<td>For encrypting and decrypting data and generating and verifying Message Authentication Codes.</td>
</tr>
<tr>
<td>CIPHER_KEY</td>
<td>For encrypting and decrypting data.</td>
</tr>
<tr>
<td>ENCIPHER_KEY</td>
<td>For encrypting data.</td>
</tr>
<tr>
<td>DECIPHER_KEY</td>
<td>For decrypting data.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MAC_KEY</td>
<td>For generating and verifying Message Authentication Codes.</td>
</tr>
<tr>
<td>MACVER_KEY</td>
<td>For verifying Message Authentication Codes.</td>
</tr>
<tr>
<td>IMPORTER_KEY</td>
<td>For decoding keys imported from other engines, or translating keys from one encoding to another.</td>
</tr>
<tr>
<td>EXPORTER_KEY</td>
<td>For encoding keys for export (to other engines) or translating keys from one encoding to another.</td>
</tr>
<tr>
<td>IKEYXSLATE_KEY</td>
<td>For inputting a key translation.</td>
</tr>
<tr>
<td>OKEYXSLATE_KEY</td>
<td>For outputting a key translation.</td>
</tr>
<tr>
<td>PINGEN_KEY</td>
<td>For generating PINs.</td>
</tr>
<tr>
<td>PINVER_KEY</td>
<td>For verifying PINs.</td>
</tr>
<tr>
<td>IPINENC_KEY</td>
<td>For importing PINs.</td>
</tr>
<tr>
<td>OPINENC_KEY</td>
<td>For exporting PINs.</td>
</tr>
<tr>
<td>KEYGEN_KEY</td>
<td>For key generation.</td>
</tr>
<tr>
<td>DKYGEN_KEY</td>
<td>For key diversification.</td>
</tr>
<tr>
<td>CVARENC_KEY</td>
<td>For encrypting cryptographic variables.</td>
</tr>
<tr>
<td>SECMMSG_KEY</td>
<td>For encrypting messages containing keys or PINs.</td>
</tr>
<tr>
<td>KEY_TYPE_TOKEN</td>
<td>Not a key type, but the token itself.</td>
</tr>
</tbody>
</table>
7.3.2  aes_engine_decipher - decipher data with AES

aes_engine_encipher deciphers data using the AES algorithm and a 16, 24, or 32 byte AES key, using the cipher block chaining mode with no initialization vector.

Function prototype

```c
long aes_engine_encipher ( optx_t *pOptx,
                          unsigned char *pAESKey,
                          long AESKeyLength,
                          unsigned char *pInputData,
                          long DataLength,
                          unsigned char*pOutputData );
```

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `pAESKey` is a pointer to a buffer containing the AES key.
- `AESKeyLength` is 16, 24, or 32.
- `pInputData` is the data to be deciphered.
- `DataLength` is the length of the data to be deciphered.
- `pOutputData` is a pointer to a buffer as long as `DataLength`.

Output

On successful exit from this routine:

- `pOutputData` contains the clear data from `pInputData`, CBC-deciphered by the key in `pAESKey`, with an initialization vector of 0s.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_INVALID_DATA_LENGTH</td>
<td>The data length is not a multiple of 16.</td>
</tr>
<tr>
<td>Other codes</td>
<td>Other error codes may be returned by the SKCH hardware.</td>
</tr>
<tr>
<td></td>
<td>See the <code>xCAES</code> function description in the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference for more information.</td>
</tr>
</tbody>
</table>
7.3.3 aes_engine_encipher - encipher data with AES

`aes_engine_encipher` enciphers data using the AES algorithm and a 16, 24, or 32 byte AES key, using the cipher block chaining mode with no initialization vector.

**Function prototype**

```c
long aes_engine_encipher ( optx_t *pOptx,
                         unsigned char *pAESKey,
                         long AESKeyLength,
                         unsigned char *pInputData,
                         long DataLength,
                         unsigned char*pOutputData );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `pAESKey` is a pointer to a buffer containing the AES key.
- `AESKeyLength` is 16, 24, or 32.
- `pInputData` is the data to be enciphered.
- `DataLength` is the length of the data to be enciphered.
- `pOutputData` is a pointer to a buffer as long as `DataLength`.

**Output**

On successful exit from this routine:

`pOutputData` contains the data from `pInputData`, CBC-enciphered by the key in `pAESKey`, with an initialization vector of 0s.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mk_NO_ERROR</code></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><code>mk_INVALID_DATA_LENGTH</code></td>
<td>The data length is not a multiple of 16.</td>
</tr>
</tbody>
</table>

**Other codes**

Other error codes may be returned by the SKCH hardware. See the `xcAES` function description in the [IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference](http://example.com) for more information.
7.3.4 aes_unwrap_with_hash - decrypt using the AESKW mechanism

`aes_unwrap_with_hash` decrypts a key and associated data sections using the AESKW key wrapping mechanism, using the provided key. This function allows the caller to choose the hashing mechanism.

**Function prototype**

```c
long aes_unwrap_with_hash ( optx_t *pOptx,
                            UCHAR *pKey,
                            uint32_t KeyLength,
                            UCHAR * pWrappedData,
                            uint32_t wrapDataLength,
                            UCHAR * pAssocData,
                            uint32_t assocDataLength,
                            uint32_t hashType,
                            UCHAR * pClearText,
                            uint32_t *pClearTextLength );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `pKey` is a pointer to an AES key.
- `KeyLength` is the length of the AES key, either 16, 24, or 32.
- `pWrappedData` is a pointer to a buffer containing the wrapped data.
- `wrapDataLength` is the length of the wrapped data.
- `pAssocData` is a pointer to the associated data which is included in the wrapped data.
- `assocDataLength` is the length of the buffer pointed to by the `pAssocData` parameter.
- `hashType` is one of `SHA_224_METHOD` or `SHA_256_METHOD`.
- `pClearText` is a pointer to a buffer in which to store the decrypted data.
- `pClearTextLength` is a pointer to the length of the buffer `pClearText`.

**Output**

On successful exit from this routine:

- `pClearText` contains the decrypted data, and `pClearTextLength` is the length of the data.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>mk_NO_ERROR</th>
<th>The operation was successful.</th>
</tr>
</thead>
</table>

**Other codes**

Other error codes may be returned by the SKCH hardware. See the `xcAES` function description in the (see the `IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference` for more information.
7.3.5 aes_unwrap - decrypt using the AESKW mechanism

aes_unwrap decrypts a key and associated data sections using the AESKW key wrapping mechanism, using the provided key.

Function prototype

```c
long aes_unwrap ( optx_t *pOptx,
                 UCHAR *pKey,
                 uint32_t keyLength,
                 UCHAR *pWrappedData,
                 uint32_t wrapDataLength,
                 UCHAR *pAssocData,
                 uint32_t assocDataLength,
                 UCHAR *pClearText,
                 uint32_t *pClearTextLength );
```

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `pKey` is a pointer to an AES key
- `keyLength` is the length of the AES key, either 16, 24, or 32 bytes.
- `pWrappedData` is a pointer to a buffer in which contains the cryptogram.
- `pWrapDataLength` is a pointer to the length of the buffer `pWrappedData`
- `pAssocData` is a pointer to the associated data to be enciphered. Under the AESKW standard, the associated data and some constant data are included in the cryptogram in specified format.
- `assocDataLength` is the length of the buffer pointed to by the `pAssocData` parameter.
- `pCleartext` is a pointer to a buffer where the data (key) to be deciphered may be stored.
- `clearTextLength` is the length of the buffer pointed to by `pClearText`.

Output

On successful exit from this routine:

- `pClearText` contains the decrypted data, and `pClearTextLength` is the length of the data.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>mk_NO_ERROR</th>
<th>The operation was successful.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other codes</td>
<td>Other error codes may be returned by the SKCH hardware. See the <code>xcAES</code> function description in the <a href="https://www.ibm.com">IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference</a> for more information.</td>
</tr>
</tbody>
</table>

154 IBM 4767 CCA UDX Reference
7.3.6 aes_wrap_with_hash_and_MK - encrypt using the AESKW mechanism

aes_wrap_with_hash encrypts a key and associated data sections using the AESKW key wrapping mechanism, using the provided master key. This function allows the caller to choose the hashing mechanism.

Function prototype

```c
long aes_wrap_with_hash_and_MK ( optx_t *pOptx,
                             mk_selectors *pMK_selector,
                             UCHAR *pClearText,
                             uint32_t clearTextLength,
                             UCHAR *pAssocData,
                             uint32_t assocDataLength,
                             uint32_t hashType,
                             UCHAR *pWrappedData,
                             uint32_t *pWrapDataLength );
```

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `pMK_selector` is a pointer to a variable which must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is set to either `old_mk`, `current_mk`, or `new_mk`, representing the key which should be used.
  - `type_mks` should be set to `ASYM_MK` if the data is to be encrypted with the asymmetric-keys master key or to `SYM_MK` if the data is to be encrypted with the symmetric-keys master key.
- `pClearText` is a pointer to a buffer containing the data (key) to be enciphered.
- `clearTextLength` is the length of the clear text.
- `pAssocData` is a pointer to the associated data to be enciphered. Under the AESKW standard, the associated data and some constant data are included in the cryptogram in specified format.
- `assocDataLength` is the length of the buffer pointed to by the `pAssocData` parameter.
- `hashType` is one of `SHA_224_METHOD` or `SHA_256_METHOD`.
- `pWrappedData` is a pointer to a buffer in which to store the resulting cryptogram.
- `pWrapDataLength` is a pointer to the length of the buffer `pWrappedData`

Output

On successful exit from this routine:

- `pWrappedData` contains the cryptogram, and `pWrapDataLength` is the length of the cryptogram.
**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>Other codes</td>
<td>Other error codes may be returned by the SKCH hardware. See the xcAES function description in the <a href="https://www.ibm.com">IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference</a> for more information.</td>
</tr>
</tbody>
</table>
7.3.7 aes_wrap - encrypt using the AESKW mechanism

aes_wrap encrypts a key and associated data sections using the AESKW key wrapping mechanism, using the provided key.

Function prototype

```c
long aes_wrap ( optx_t *pOptx,
              UCHAR *pKey,
              uint32_t keyLength,
              UCHAR *pClearText,
              uint32_t clearTextLength,
              UCHAR *pAssocData,
              uint32_t assocDataLength,
              UCHAR *pWrappedData,
              uint32_t *pWrapDataLength );
```

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `pKey` is a pointer to an AES key.
- `keyLength` is the length of the AES key, either 16, 24, or 32 bytes.
- `pClearText` is a pointer to a buffer containing the data (key) to be enciphered.
- `clearTextLength` is the length of the clear text.
- `pAssocData` is a pointer to the associated data to be enciphered. Under the AESKW standard, the associated data and some constant data are included in the cryptogram in specified format.
- `assocDataLength` is the length of the buffer pointed to by the `pAssocData` parameter.
- `pWrappedData` is a pointer to a buffer in which to store the resulting cryptogram.
- `pWrapDataLength` is a pointer to the length of the buffer `pWrappedData`

Output

On successful exit from this routine:

- `pWrappedData` contains the cryptogram, and `pWrapDataLength` is the length of the cryptogram.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mk_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>Other codes</td>
<td>Other error codes may be returned by the SKCH hardware. See the <code>xcAES</code> function description in the <a href="https://www.ibm.com/support/docview.wss?uid=swg21922259">IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference</a> for more information.</td>
</tr>
</tbody>
</table>
7.3.8 cas_adjust_parity - adjust parity

cas_adjust_parity adjusts each byte of the passed string, as necessary, so that every byte has odd parity.
This is useful when adjusting DES keys for correct parity.

Function prototype

```c
void cas_adjust_parity ( UCHAR *DataBytes,
                        unsigned int Length );
```

Input

On entry to this routine:

DataBytes is a pointer to the string that is to be parity-adjusted.
Length is the number of bytes in the string at location DataBytes.

Output

On successful exit from this routine:

DataBytes is a pointer to the string that has odd parity.

Return codes

This function has no return codes.
7.3.9 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.

Note: This function is run as part of the `parse_keys` function, so it is not required for keys which were decrypted by the parser.

**Function prototype**

```c
boolean cas_des_key_token_check ( des_key_token_structure *pKeyToken,
                                 DES_TOKEN_CHECK *pMessageFlag );
```

**Input**

On entry to this routine:

- `pKeyToken` is a pointer to the internal DES key token that is to be checked.
- `pMessageFlag` is a pointer to a location where the return code can be stored.

**Output**

On successful exit from this routine:

The buffer at `pMessageFlag` contains one of the values defined in “Return codes” below.

`cas_des_key_token_check` returns a boolean value of **TRUE**, if the value returned in `pMessageFlag` is **DES_TOKEN_CHECK_VALID**, or **FALSE** otherwise.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>DES_TOKEN_CHECK_VALID</th>
<th>The token is valid.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES_TOKEN_CHECK_TOKENFLAG</td>
<td>The token is not an internal DES key token.</td>
</tr>
<tr>
<td>DES_TOKEN_CHECK_RESERVED/</td>
<td>Reserved field i (1 &lt;= i &lt;= 6) is incorrectly set.</td>
</tr>
<tr>
<td>DES_TOKEN_CHECK_VERSION</td>
<td>The version number is incorrect.</td>
</tr>
<tr>
<td>DES_TOKEN_CHECK_FLAG_BYTE</td>
<td>The token flag is incorrect.</td>
</tr>
<tr>
<td>DES_TOKEN_CHECK_FLAG_NOCV</td>
<td>The token has no control vector set.</td>
</tr>
<tr>
<td>DES_TOKEN_CHECK_NOKEY</td>
<td>The token does not contain a key.</td>
</tr>
<tr>
<td>DES_TOKEN_CHECK_KEY_LENGTH</td>
<td>The key length is invalid.</td>
</tr>
<tr>
<td>DES_TOKEN_CHECK_VERSION_ONE</td>
<td>The key is a zero CV key, a double or triple length</td>
</tr>
<tr>
<td>DATA key for S390.</td>
<td>DES_TOKEN_CHECK_INVALID_ZEROCV</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td></td>
<td>The key is a zero CV key but the control vector is a non-zero.</td>
</tr>
</tbody>
</table>
7.3.10 cas_get_key_type - return key type

*cas_get_key_type* returns the key type corresponding to the specified key token.

**Function prototype**

\[ \text{KEY_TYPES cas_get_key_type ( des_key_token_structure *pKeyToken );} \]

**Input**

On entry to this routine:

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

**Output**

On successful exit from this routine:

*cas_get_key_type* returns the key type corresponding to the specified key token. See “Key types” on page 149 for more information.

**Return codes**

This function has no return codes.
7.3.11  **cas_key_length - return key length**

*cas_key_length* determines the length of a key, based on the Control Vector. The key length is returned as the function result.

**Note:** This function does not properly deal with external DATA tokens, or CCF DATA tokens, which have zero-value control vectors. Functions which may be testing external DATA keys or keys with zero-value control vectors, should use the *S390KeyLength* function instead.

**Function prototype**

```
LENGTH_KEYWORD cas_key_length ( eightbyte CvBase,
                                eightbyte CvExtension );
```

**Input**

On entry to this routine:

- *CvBase* is the control vector base.
- *CvExtension* is the control vector extension.

**Output**

On successful exit from this routine:

*cas_key_length* returns **SINGLE** or **DOUBLE**, depending on whether the specified key is single or double length.

**Examples**

To determine the length of the key stored in *DataKey*:

```
switch( cas_key_length( DataKey, cvBase, DataKey. cvExten ) )
{
  case SINGLE:
    /* deal with a single length key */
    break;
  case DOUBLE:
    /* deal with a double length key */
    break;
  default :
    /* return with an error */
    break;
}
```

**Return codes**

This function has no return codes.
7.3.12 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.
7.3.13  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.
7.3.14 CasAesMasterKeyCheck - perform master key version check

CasAesMasterKeyCheck determines which master key an AES key is encrypted under: old, current, or new.

Note: The parse_keys() function will XOR the optok[ ]->outflags field with TOKO_MK_OLD, TOKO_MK_CUR, or TOKO_MK_NEW, whichever is correct, so this function is not needed.

Function prototype

UNDER_MASTER_KEY CasAesMasterKeyCheck( mk_selectors *MKSelector,
                                           aes_token_t *pAESKey );

Input

On entry to this routine:

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

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

pAESKey is a pointer to an internal AES key token.

Output

On successful exit from this routine:

CasAesMasterKeyCheck returns either OLD, CURRENT, or OUT_OF_DATE.

Return codes

This function has no return codes.
7.3.15  **CasBuildCv - build a default control vector**

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

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

**Function prototype**

```c
boolean CasBuildCv ( KEY_TYPES KeyType,
                    CV_LENGTHS CV_Length,
                    UCHAR *pCV );

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

**Input**

On entry to this routine:

- `KeyType` is the type of key your control vector is used with. Possible key types include:
- `pCV` is a pointer to a 16-byte location which will hold the new control vector.
- `CV_Length` is one of `CV_DOUBLE`, `CV_SINGLE`, or `CV_DEFAULT`, depending on what length of key you are building a control vector for.

**Output**

On successful exit from this routine:

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

**Return codes**

This function has no return codes.
7.3.16 CasBuildToken - build a token

CasBuildToken builds a DES or AES 64-byte skeleton key token for the mk_set being processed.

Note: Using the builder, otk_build_a_token on the host or otk_build_tokens on the coprocessor, eliminates the need for this function.

Function prototype

```c
boolean CasBuildToken ( UCHAR TokenFlag,
                        CV_LENGTHS CV_Length,
                        mk_selectors *MKSelector,
                        optx_tokbld_t *mybtok );
```

Input

On entry to this routine:

TokenFlag is the token flag used in constructing the new key token. Legal values for this field are:

- INTERNAL_TOKEN_FLAG An operational key, which will be encrypted by the master key.
- EXTERNAL_TOKEN_FLAG A key which will be encrypted by another key. This is not valid for an AES fixed-length token.

CV_Length is one of the following:

- CV_DOUBLE A double length (16 byte) key.
- CV_SINGLE A single length (8 byte) key.
- CV_DEFAULT The default length for the designated KeyType.

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

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

mybtok is a pointer to a 64-byte structure with the following fields initialized:

- tok_out is a pointer to a 64-byte field to hold the output token.
- kt.des is the key type for the desired token.
- cv is either NULL indicating that the default CV for the key type in kt.des is desired, or a pointer to a valid control vector of the desired key type.

Output

On successful exit from this routine:

mybtok->tok_out contains the token constructed by the function.

CasBuildToken returns a value of TRUE if successful, and FALSE otherwise.
Return codes
This function has no return codes.
7.3.17  CasCurrentMkvp - current master key verification pattern

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

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

Function prototype

```c
boolean CasCurrentMkvp ( mk_selectors *pMKSelector,
                         UCHAR *pMKVP );
```

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

Input

On entry to this routine:

- \( pMKSelector \) is a parameter of type \( mk\_selectors \), indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - \( mk\_set \) is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to \( MK\_SET\_DEFAULT \).
  - \( mk\_register \) must be set to \( current\_mk \).
  - \( type\_mks \) should be set to \( SYM\_MK \).

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

Output

On successful exit from this routine:

- \( pMKVP \) contains the current master key verification pattern.

CasCurrentMkvp returns \texttt{TRUE} if the verification pattern was found, and \texttt{FALSE} otherwise.

Return codes

This function has no return codes.
7.3.18  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.

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

**Note:** The parse_keys() function will XOR the optok[ ]->outflags field with TOKO_MK_OLD, TOKO_MK_CUR, or TOKO_MK_NEW, whichever is correct, so this function is not needed.

**Function prototype**

```c
UNDER_MASTER_KEY CasMasterKeyCheck ( optx_t * pOptx,
                                       mk_selectors *pMKSelector,
                                       des_key_token_structure *pKeyToken );
```

```c
UNDER_MASTER_KEY cas_master_key_check (  optx_t * pOptx,
                                          des_key_token_structure *pKeyToken );
```

**Input**

On entry to this routine:

- **pMKSelector** is a parameter of type *mk_selectors*, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - *mk_set* is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
  - *mk_register* is ignored.
  - *type_mks* should be set to SYM_MK.

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

**Output**

On successful exit from this routine:

CasMasterKeyCheck returns either OLD, CURRENT, or OUT_OF_DATE which identifies which master key (old, current, or no longer available) the key token is encrypted under.

If CasMasterKeyCheck did not return OUT_OF_DATE, the *mk_registers* field of the pMKSelectors parameter has been updated to the appropriate value (old_mk or current_mk).

**Notes**

In CCA, an operational key is a key that has been multiply-enciphered with the master key. In order to use an operational key, it must first be deciphered using the master key.

When the user (security officer, 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 CCA Basic Services Reference and Guide* for more information.

170  IBM 4767 CCA UDX Reference
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.
7.3.19  **CasOldMkvp - old master key verification pattern**

*CasOldMkvp* returns the 20-byte master key verification pattern (MKVP) for the SYM 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.

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

**Function prototype**

```c
boolean CasOldMkvp ( mk_selectors *pMKSelector,
                     UCHAR *pMKVP);

boolean cas_old_mkvp ( UCHAR *pMKVP );
```

**Input**

On entry to this routine:

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

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

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

**Output**

On successful exit from this routine:

*pMKVP* contains the current master key verification pattern.

*CasOldMkvp* returns **TRUE** if the verification pattern was found, and **FALSE** otherwise.

**Return codes**

This function has no return codes.
**7.3.20 S390KeyLength - find the key length of a DES key token**

S390KeyLength finds the length (whether **SINGLE**, **DOUBLE**, or **TRIPLE**) of a DES key in a CCA key token.

**Note:** Only S390 DATA tokens are allowed to be triple-length.

**Note:** This function is deprecated. The key length is available in the optok[ ] structure, either as ckey_len, ckey_bit_len, or in the optok[n]->outflags field as TOKO_EFF_SINGLE_KEY or TOKO_EFF_DOUBLE_KEY.

**Function prototype**

```
LENGTH_KEYWORD S390KeyLength (des_key_token_structure *pKeyToken );
```

**Input**

`pKeyToken` is a pointer to a 64-byte DES key token, either internal or external format key.

**Output**

On successful completion of this function:

S390KeyLength returns either **SINGLE**, **DOUBLE**, or **TRIPLE**, indicating the length of the key contained in `pKeyToken`.

**Return codes**

This function has no return codes.
8 RSA functions

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

Refer to Appendix B of the IBM 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.

8.1 Header files for RSA functions

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

```c
#include "cmncryt2.h" /* T2 CPRB definitions */
#include "scctypes.h" /* Secure Cryptographic Coprocessor structures */
#include "xc_types.h" /* 4767 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 */
```

8.2 Summary of functions

RSA keys and key tokens include the following functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>CalculatenWordLength</td>
<td>Returns the number of 4-byte words in the modulus.</td>
<td>See “Functions to determine field lengths” on page 178.</td>
</tr>
<tr>
<td>CreateInternalKeyToken</td>
<td>Receives a clear key token and creates the operational form with the default master key.</td>
<td>See “Functions to manipulate RSA key tokens” on page 195.</td>
</tr>
<tr>
<td>CreateInternalKeyTokenWithMK</td>
<td>Receives a clear key token and creates the operational form.</td>
<td>See “Functions to manipulate RSA key tokens” on page 195.</td>
</tr>
<tr>
<td>CreateRsaInternalSection</td>
<td>Creates the RSA internal section with the default master key set.</td>
<td>See “Functions to manipulate RSA key tokens” on page 195.</td>
</tr>
<tr>
<td>CreateRsaInternalSectionWithMK</td>
<td>Creates the RSA internal section with the specified master key set.</td>
<td>See Functions to manipulate RSA key tokens” on page 195.</td>
</tr>
<tr>
<td>delete_KeyToken</td>
<td>Removes a registered public or private key from the SRDI where it is stored.</td>
<td>See “Functions to handle the on-board key storage on page 223.</td>
</tr>
<tr>
<td>ExtractRsaPublicSection</td>
<td>Format a public key token from a private/public key token.</td>
<td>See “Functions to manipulate RSA key tokens” on page 195.</td>
</tr>
<tr>
<td>GenerateCcaRsaToken</td>
<td>Generates a CCA RSA key token from an internal format key</td>
<td>See “Functions to manipulate RSA key tokens” on page 195.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>See</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-----</td>
</tr>
<tr>
<td>GenerateRsInternalToken</td>
<td>Creates an internal RSA token from a CCA RSA keytoken.</td>
<td>“Functions to manipulate RSA key tokens” on page 195.</td>
</tr>
<tr>
<td>generate_dSig</td>
<td>Generates a digital signature.</td>
<td>“Functions to use the RSA keys” on page 217.</td>
</tr>
<tr>
<td>GeteLength</td>
<td>Returns the RSA public exponent byte length.</td>
<td>“Functions to manipulate RSA key tokens” on page 195.</td>
</tr>
<tr>
<td>getKeyToken</td>
<td>Retrieves a PKA token from the SRDI where it is stored.</td>
<td>“Functions to handle the on-board key storage” on page 223.</td>
</tr>
<tr>
<td>GetModulus</td>
<td>Extracts and copies the RSA modulus.</td>
<td>“Functions to manipulate RSA key tokens” on page 195.</td>
</tr>
<tr>
<td>GetnBitLength</td>
<td>Returns the bit length of the RSA modulus.</td>
<td>“Functions to determine field lengths” on page 178.</td>
</tr>
<tr>
<td>GetnByteLength</td>
<td>Returns the byte length of the RSA modulus.</td>
<td>“Functions to determine field lengths” on page 178.</td>
</tr>
<tr>
<td>GetPublicExponent</td>
<td>Extracts and copies the RSA key public exponent.</td>
<td>“Functions to manipulate RSA key tokens” on page 195.</td>
</tr>
<tr>
<td>GetRsaKeyNameSection</td>
<td>Returns a pointer to the key name section which holds the name of the key.</td>
<td>“Functions to manipulate RSA key tokens” on page 195.</td>
</tr>
<tr>
<td>GetRsaPrivateKeySection</td>
<td>Returns a pointer to the private key section of an RSA key token.</td>
<td>“Functions to manipulate RSA key tokens” on page 195.</td>
</tr>
<tr>
<td>GetRsaPublicKeySection</td>
<td>Returns a pointer to the public key section of an RSA key token.</td>
<td>“Functions to manipulate RSA key tokens” on page 195.</td>
</tr>
<tr>
<td>GetRsaPuKeyCertSection</td>
<td>Returns a pointer to the public key certificate section.</td>
<td>“Functions to manipulate RSA key tokens” on page 195.</td>
</tr>
<tr>
<td>getTokenLength</td>
<td>Returns the length of the specified token.</td>
<td>“Functions to determine field lengths” on page 178.</td>
</tr>
<tr>
<td>IsPrivateExponentEven</td>
<td>Verifies whether the RSA private exponent is an even valued integer.</td>
<td>“Functions to validate RSA key tokens” on page 184.</td>
</tr>
<tr>
<td>IsPrivateKeyEncrypted</td>
<td>Verifies whether the private key section of the specified key token is encrypted.</td>
<td>“Functions to validate RSA key tokens” on page 184.</td>
</tr>
<tr>
<td>IsPublicExponentEven</td>
<td>Verifies whether the RSA public</td>
<td>“Functions to validate RSA</td>
</tr>
<tr>
<td>Function Name</td>
<td>Description</td>
<td>See “Functions to validate RSA key tokens” on page 184.</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>exponent is an even valued integer</td>
<td>key tokens</td>
<td>184.</td>
</tr>
<tr>
<td>IsRsaToken</td>
<td>Verifies whether the key token contains an RSA key.</td>
<td>184.</td>
</tr>
<tr>
<td>IsS390Token</td>
<td>Verifies whether the key token is a valid type for S390.</td>
<td>184.</td>
</tr>
<tr>
<td>PkaHashQueryWithMK</td>
<td>Returns the master key version used to encrypt the specified token.</td>
<td>184.</td>
</tr>
<tr>
<td>PkaMkvpQuery</td>
<td>Returns a value indicating which master key in the default set was used to encrypt the specified key token.</td>
<td>184.</td>
</tr>
<tr>
<td>PkaMkvpQueryWithMK</td>
<td>Returns a value indicating which master key was used to encrypt the specified key token.</td>
<td>184.</td>
</tr>
<tr>
<td>pka96_tvvgen</td>
<td>Calculates the token validation value (TVV) for the specified key token.</td>
<td>195.</td>
</tr>
<tr>
<td>RecoverPkaClearKeyTokenUnder Mk</td>
<td>Recovers the PKA clear key token under the master key of the default key set.</td>
<td>195.</td>
</tr>
<tr>
<td>RecoverPkaClearKeyTokenUnder Xport</td>
<td>Recovers the PKA clear key token under a DES export key.</td>
<td>195.</td>
</tr>
<tr>
<td>RecoverPkaClearTokenUnderMk WithMK</td>
<td>Recovers the PKA clear key token under the master key.</td>
<td>195.</td>
</tr>
<tr>
<td>RecoverPkaClearTokenUnderMkWithMinModulus</td>
<td>Recovers the PKA clear key token under the specified master key.</td>
<td>195.</td>
</tr>
<tr>
<td>ReEncipherPkaKeyToken</td>
<td>Re-enciphers an internal PKA token from the old master key to the current master key of the default key set.</td>
<td>195.</td>
</tr>
<tr>
<td>ReEncipherPkaKeyTokenWithMK</td>
<td>Re-enciphers an internal PKA key token from the old master key to the current master key of a specified key set.</td>
<td>195.</td>
</tr>
<tr>
<td>RequestRSACrypto</td>
<td>Performs an RSA operation.</td>
<td>217.</td>
</tr>
</tbody>
</table>
store_KeyToken
Saves a public or private key to the SRDI. See “Functions to handle the on-board key storage” on page 223.

TokenMkvpMatchMasterKey
Tests whether the key token was encrypted using a specified version of the master key. See “Functions to validate RSA key tokens” on page 184.

ValidatePkaToken
Verifies that the RSA token is valid for use in the system. See “Functions to validate RSA key tokens” on page 184.

ValidatePkaTokenWithMinModulus
Verifies that the RSA key token is valid for use in the system, also checking for S390 minimum modulus lengths. See “Functions to validate RSA key tokens” on page 184.

VerifyRsaKeyTokenConsistency
Verifies the consistency of a key token. See “Functions to validate RSA key tokens” on page 184.

verify_dSig
Verifies an RSA signature. See “Functions to use the RSA keys” on page 217.

8.3 Overview
An RSA key consists of a public modulus (n) which is the product of two large prime numbers, a public exponent (e) which is relatively prime to the modulus, and a private exponent (d). In the coprocessor, keys may be stored in CCA RSA tokens as in the key storage file or in SCC (Secure Cryptographic Coprocessor) form, and are used in 4767 internal tokens. Each form of key has a public and a private version.

The public version 4767 internal token or SCC token of a key contains the modulus and the public exponent of the key, and the length of each. The private version may be in either modulus exponent or Chinese remainder format, and contains the modulus and public and private exponents for each. The 4767 version of a key is used in the cryptographic engine for xcRSA requests and is the type returned by xcRSAKeyGenerate. SCC tokens are an intermediate version of the key which is used in the translation of keys from one token format to the other.

CCA RSA tokens consist of a token header, followed by:
1. An optional private key section which holds the decrypting information (the private key and the public modulus), verification data, and key-encryption data
2. A required public key section which holds encryption information (the public exponent, the modulus length, and, if there is no private section, the modulus itself).
3. An optional public key certificate section may follow, or
4. Depending on the format, a separate section containing private-key blinding information, called the internal section.

Parts of the private key section may be encrypted under the master key (operational keys) or under a transport key (external keys).
8.4 Functions to determine field lengths

The following functions operate on CCA RSA key tokens to recover the lengths of the internal fields. The key lengths in a CCA RSA key token are all in S390 or big-endian order, and in some cases refer to the length of the field rather than the length of the key part inside the field. Therefore, these functions are provided to make handling the key tokens easier. However, if the parser has been used to decrypt the token, the values may generally be determined from the optx_tok_t structure.

Note: All of these functions operate on CCA RSA key tokens.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CalculatenWordLength</td>
<td>Returns the length of the modulus in 16-bit words.</td>
</tr>
<tr>
<td>GeteLength</td>
<td>Returns the length of the public exponent.</td>
</tr>
<tr>
<td>GetnBitLength</td>
<td>Returns the length of the modulus in bits.</td>
</tr>
<tr>
<td>GetnByteLength</td>
<td>Returns the length of the modulus in bytes.</td>
</tr>
<tr>
<td>GetTokenLength</td>
<td>Returns the length of the CCA RSA key token.</td>
</tr>
</tbody>
</table>
8.4.1 CalculateWordLength - return word length of modulus

*CalculateWordLength* returns the length of the modulus in terms of the number of 16-bit words it occupies.

**Function prototype**

```c
USHORT CalculateWordLength ( RsaKeyTokenHeader *pToken );
```

**Input**

On entry to this routine:

*pToken* is a pointer to the key token.

**Output**

On successful exit from this routine:

*CalculateWordLength* returns the length of the modulus in 16-bit words.

**Return codes**

This function has no return codes.
8.4.2 GeteLength - return RSA public exponent byte length

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

Function prototype

```c
USHORT GeteLength ( RsaKeyTokenHeader *pToken );
```

Input

On entry to this routine:

pToken is a pointer to the key token.

Output

On successful exit from this routine:

GeteLength returns the byte length of the RSA public exponent field.

Return codes

This function has no return codes.
8.4.3 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.

This is the value in the tok_alg_val field of the optx_tok_t structure for an RSA key.

**Function prototype**

```c
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.
8.4.4 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.
8.4.5 GetTokenLength - return key token length

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

**Function prototype**

```c
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.
### 8.5 Functions to validate RSA key tokens

The following functions operate on CCA RSA key tokens to validate the contents of the internal fields.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsPrivateExponentEven</td>
<td>Verifies whether the private exponent of the CCA RSA key token is even.</td>
</tr>
<tr>
<td>IsPrivateKeyEncrypted</td>
<td>Verifies whether the private key section of the CCA RSA key token is encrypted.</td>
</tr>
<tr>
<td>IsPublicExponentEven</td>
<td>Verifies whether the public exponent of the CCA RSA key token is even.</td>
</tr>
<tr>
<td>IsRsaToken</td>
<td>Verifies whether the supplied token is an RSA token.</td>
</tr>
<tr>
<td>IsS390Token</td>
<td>Verifies whether the key token is a valid type for z/OS.</td>
</tr>
<tr>
<td>PkaHashQueryWithMK</td>
<td>Identifies which master key was used to encrypt the specified internal CCA RSA key token.</td>
</tr>
<tr>
<td>PkaMkvpQuery</td>
<td>Identifies which master key of the default key set was used to encrypt the specified internal CCA RSA key token.</td>
</tr>
<tr>
<td>PkaMkvpQueryWithMK</td>
<td>Identifies which master key was used to encrypt the specified internal CCA RSA key token.</td>
</tr>
<tr>
<td>TokenMkvpMatchMasterKey</td>
<td>Tests whether the operational CCA RSA key token was encrypted with the specified master key.</td>
</tr>
<tr>
<td>ValidatePkaToken</td>
<td>Verifies that a CCA RSA key token is valid for use in the system.</td>
</tr>
<tr>
<td>ValidatePkaTokenWithMinModulus</td>
<td>Verifies that the RSA key token is valid for use in the system, also checking for S390 minimum modulus lengths.</td>
</tr>
<tr>
<td>VerifyRsaKeyTokenConsistency</td>
<td>Tests the length fields of the CCA RSA key token, ensuring that they are consistent.</td>
</tr>
</tbody>
</table>

**Note:** the `parse_keys` function uses all of these tests, so that a key which has been decrypted using the parser does not need to have these functions run.
8.5.1 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**

```c
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.
8.5.2 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.
8.5.3 IsPublicExponentEven - verify RSA public exponent

`IsPublicExponentEven` returns **TRUE** if the public exponent in the specified key token is an even valued integer; otherwise, it returns **FALSE**.

**Function prototype**

```c
boolean IsPublicExponentEven ( RsaKeyTokenHeader *pToken );
```

**Input**

On entry to this routine:

`pToken` is a pointer to the key token.

**Output**

On successful exit from this routine:

`IsPublicExponentEven` returns **TRUE** if the public exponent in the specified key token is an even valued integer, and **FALSE** if it is not.

**Return codes**

This function has no return codes.
8.5.4 IsRsaToken - verify RSA key

*IsRsaToken* returns **TRUE** if the specified key token contains an RSA key, or **FALSE** if it does not.

**Function prototype**

```c
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.
8.5.5 IsS390Token - verify z/OS compatibility

*IsS390Token* verifies that the key provided is of a type which may be used in a z/OS system.

**Note**: This function may only be used on a coprocessor with the z/OS load.

**Function prototype**

```c
boolean IsS390Token ( RsaKeyTokenHeader *pToken );
```

**Input**

On entry to this routine:

*pToken* is a pointer to the key token.

**Output**

On successful exit from this routine:

*IsS390Token* returns **TRUE** if the specified key token contains an RSA key, and **FALSE** if it is not an RSA key token.

**Return codes**

This function has no return codes.
8.5.6 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.

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

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

Output
On successful exit from this routine:

PkaHashQueryWithMK returns the version of the master key (MK_CURRENT, MK_OLD, or MK_OUT_OF_DATE) that was used to encrypt the specified key token. If the value returned was not MK_OUT_OF_DATE, pMKSelector->mk_register is set to either old_mk or current_mk, whichever is appropriate.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_KEY_NOT_VALID</td>
<td>The current master key is not valid.</td>
</tr>
<tr>
<td>mk_SEMCLAIM_FAILED</td>
<td>Could not access the master keys.</td>
</tr>
</tbody>
</table>
8.5.7 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.

*PkaMkvpQuery* has the same effect as calling *PkaMkvpQueryWithMK* after setting the pMKSelector parameter to \{MK_SET_DEFAULT, current_mk, ASYM_MK\}.

**Note:** These functions works only with version 0 tokens.

**Function prototype**

\[
\text{MK_VERSION PkaMkvpQueryWithMK ( RsaKeyTokenHeader *pToken, mk_selectors *pMKSelector );}
\]

\[
\text{MK_VERSION PkaMkvpQuery ( RsaKeyTokenHeader *pToken );}
\]

**Input**

On entry to this routine:

- **pToken** is a pointer to the key token that is checked.
- **pMKSelector** is a parameter of type *mk_selectors*, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - *mk_set* is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
  - *mk_register* is ignored.
  - *type_mks* should be set to ASYM_MK.

**Output**

On successful exit from this routine:

*PkaMkvpQueryWithMK* returns the version of the master key (MK_CURRENT, MK_OLD, or MK_OUT_OF_DATE) that was used to encrypt the specified key token. If the value returned was not MK_OUT_OF_DATE, pMKSelector->mk_register is set to either old_mk or current_mk, whichever is appropriate.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_KEY_NOT_VALID</td>
<td>The current master key is not valid.</td>
</tr>
<tr>
<td>mk_SEM_CLAIM_FAILED</td>
<td>Could not access the master keys.</td>
</tr>
</tbody>
</table>
8.5.8 TokenMkvpMatchMasterKey - test encryption of RSA key

TokenMkvpMatchMasterKey tests whether the specified key token was encrypted using a specified version of the master key. The Master Key Verification Pattern (MKVP) of the specified key token is compared to the MKVP for the specified master key. If the two are equal, the function returns TRUE; if not, it returns FALSE.

Function prototype

```c
boolean TokenMkvpMatchMasterKey( mk_selectors *mk_selector,
                                   RsaKeyTokenHeader *pToken );
```

Input

On entry to this routine:

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

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

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

Output

On successful exit from this routine:

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

Return codes

This function has no return codes.
8.5.9 ValidatePkaTokenWithMinModulus - validate RSA key token

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

*ValidatePkaToken* has the same effect as calling *ValidatePkaTokenWithMinModulus* after setting the signal parameter to *ORIGINAL_MIN*.

**Function prototype**

```c
long ValidatePkaTokenWithMinModulus ( RsaKeyTokenHeader *pToken, uint32_t *pErrorCode, long signal );
long ValidatePkaToken ( RsaKeyTokenHeader *pToken, uint32_t *pErrorCode );
```

**Input**

On entry to this routine:

- *pToken* is a pointer to the RSA key token.
- *pErrorCode* is a pointer to the location where the function stores an error code, if a critical error occurs.
- *signal* is a value indicating the type of function the RSA key length is to be validated for:
  - *ORIGINAL_MIN* - Minimum bit length of 512 bits
  - *PKE_DSV_MIN* - Minimum bit length of 128 bits
  - *PKD_MIN* - Minimum bit length of 128 bits for external keys; otherwise, 512 bits is minimum

**Output**

This function returns no output.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>ERROR</td>
<td><em>pErrorCode</em> contains one of the following error codes:</td>
</tr>
<tr>
<td>RSA_KEY_INVALID</td>
<td>The input token is not an internal or external RSA key token.</td>
</tr>
<tr>
<td>RT_TKN_UNUSEABLE</td>
<td>The input token is not an RSA key token.</td>
</tr>
<tr>
<td>E_KEY_TKNVER</td>
<td>Incorrect version data in input token.</td>
</tr>
<tr>
<td>E_PKA_KEYINVALID</td>
<td>An error was found in the token.</td>
</tr>
</tbody>
</table>
8.5.10 VerifyRsaKeyTokenConsistency - verify RSA key token consistency

VerifyRsaKeyTokenConsistency 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**

```c
long VerifyRsaKeyTokenConsistency ( 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**

This function returns no output.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The key token was consistent and the operation was successful.</td>
</tr>
<tr>
<td>ERROR</td>
<td>The key token was inconsistent.</td>
</tr>
</tbody>
</table>
8.6 Functions to manipulate RSA key tokens

The following functions are used to manipulate CCA RSA keys tokens and their sccRsa counterparts, including building and translating the key types.

<table>
<thead>
<tr>
<th>RSA Key Manipulation Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bld_cca_rsa_from_xcrsa_tok</strong></td>
</tr>
<tr>
<td><strong>bld_xcrsa_from_cca_rsa_tok</strong></td>
</tr>
<tr>
<td><strong>CreateInternalKeyToken</strong></td>
</tr>
<tr>
<td><strong>CreateInternalKeyTokenWithMK</strong></td>
</tr>
<tr>
<td><strong>CreateRsaiInternalSection</strong></td>
</tr>
<tr>
<td><strong>CreateRsaiInternalSectionWithMK</strong></td>
</tr>
<tr>
<td><strong>ExtractRsaPublicKeySection</strong></td>
</tr>
<tr>
<td><strong>GenerateCcaRsaToken</strong></td>
</tr>
<tr>
<td><strong>GenerateRsaiInternalToken</strong></td>
</tr>
<tr>
<td><strong>GetModulus</strong></td>
</tr>
<tr>
<td><strong>GetPublicKeyExponent</strong></td>
</tr>
<tr>
<td><strong>GetRsaKeyNameSection</strong></td>
</tr>
<tr>
<td><strong>GetRsaPrivateKeySection</strong></td>
</tr>
<tr>
<td><strong>GetRsaPublicKeySection</strong></td>
</tr>
<tr>
<td><strong>GetRsaPuKeyCertSection</strong></td>
</tr>
<tr>
<td>Function Name</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>pka96_tvvgen</td>
</tr>
<tr>
<td>RecoverPkaClearKeyTokenAESKW</td>
</tr>
<tr>
<td>RecoverPkaClearKeyTokenUnderXport</td>
</tr>
<tr>
<td>RecoverPkaClearTokenUnderMkWithMK WithMinModulus</td>
</tr>
<tr>
<td>ReEncipherPkaKeyToken</td>
</tr>
<tr>
<td>ReEncipherPkaKeyTokenWithMK</td>
</tr>
</tbody>
</table>
8.6.1  bld_cca_rsa_from_xcrsa_tok- create operational key token

*bld_cca_rsa_from_xcrsa_tok* receives a clear RSA key token and creates the CCA form by moving the key parts, padding as required, and calculating the CCA token length and internal part lengths.

**Note:** Use CCA routine CSNBPKB to create the skeleton key token for the output.

**Function prototype**

```c
uint32_t bld_cca_rsa_from_xcrsa_tok ( RsaKeyTokenHeader *pTokenOut, xcRsaKeyToken_t *pTokenIn, uint16_t internal );
```

**Input**

On entry to this routine:

- *pTokenOut* is a CCA RSA skeleton key token whose fields are initialized as if by CSNBPKB:
  - `nextSection` is set to the type of CCA key required:
    - `RSA_PRIVATE_SECTION_NOPT`
    - `RSA_PRIVATE_SECTION_NOPT_NEW`
    - `RSA_PRIVATE_SECTION_CR_NEW`
    - `RSA_PRIVATE_SECTION_NOPT_VAR`
    - `RSA_PRIVATESECTION_NOPT_VAR_OPK`
    - `RSA_PRIVATE_SECTION_AESMK_ME`
    - `RSA_PRIVATE_SECTION_AESMK_CRT`
  - `tokenLength` contains the length of the token, in big-endian format.
  - The section length fields of the following segments (private key section, public key section, and so on) are initialized.

- *pTokenIn* is a pointer to the cleartext xcRSAkey token. The `pTokenIn->type` field must agree with the `pTokenOut->nextSection` field, in that both must be CRT type keys or both must be NOPT/ME keys.

- `internal` is true (non-zero) if the key token is to be an internal token. In particular, a token which is to be a CCA `RSA_PRIVATESECTION_NOPT` will be created as an `RSA_PRIVATESECTION_NOPT_NEW` key `internal` is true, or `RSA_PRIVATESECTION_NOPT` if `internal` is false.

**Output**

On successful exit from this routine:

- *pTokenOut* contains the clear CCA key token.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>ERROR</td>
<td>An error was found in the format of the key (for example, no private</td>
</tr>
</tbody>
</table>

RSA functions 197
### 8.6.2 bld_xc rsa from cca_rsa_tok - create an RSA token in xc format

*bld_xc rsa from cca_rsa_tok* receives a clear CCA RSA key token and creates the hardware compatible form of xCRsaKeyToken_t.

#### Function prototype

```c
uint32_t bld_xc rsa_from_c ca_rsa_tok ( RsaKeyTokenHeader *pTokenIn,
                xCRsaKeyToken_t *pTokenOut);
```

#### Input

On entry to this routine:

- `pTokenIn` is a pointer to the cleartext CCA key token.
- `pTokenOut` is a pointer to a buffer where the new key can be stored. The buffer must be large enough to hold the keytoken structure as well as all of the parts of the key. In general, you should all 2000 bytes for such a token.

#### Output

On successful exit from this routine:

- `pTokenOut` contains the clear key token in xCRsaKeyToken_t format.

#### Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>ERROR</td>
<td>The token was unusable.</td>
</tr>
</tbody>
</table>
8.6.3 CreateInternalKeyTokenWithMK - create operational key token

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

CreateInternalKeyToken has the same effect as calling CreateInternalKeyTokenWithMK after setting the pMKSelector parameter to \(\text{\{MK\_SET\_DEFAULT, current\_mk, ASYM\_MK\}}\).

**Function prototype**

```c
long CreateInternalKeyTokenWithMK ( RsaKeyTokenHeader *pTokenIn, mk_selectors *pMKSelector, RsaKeyTokenHeader *pTokenOut, optx_t *pOptx );

long CreateInternalKeyToken ( RsaKeyTokenHeader *pTokenIn, RsaKeyTokenHeader *pTokenOut, optx_t *pOptx );
```

**Input**

On entry to this routine:

- \(pTokenIn\) is a pointer to the cleartext key token.
- \(pMKSelector\) is a parameter of type \(\text{mk\_selectors}\), 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 \(\text{MK\_SET\_DEFAULT}\).
  - \(mk\_register\) is ignored.
  - \(type\_mks\) should be set to \(\text{ASYM\_MK}\).
- \(pOptx\) is a pointer to the operation tracking structure for this verb.

**Output**

On successful exit from this routine:

- \(pTokenOut\) contains the encrypted operational key token.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_KEY_NOT_VALID</td>
<td>The current master key is not valid.</td>
</tr>
<tr>
<td>mk_SEM_CLAIM_FAILED</td>
<td>Could not access the master keys.</td>
</tr>
</tbody>
</table>
8.6.4 CreateRsaInternalSectionWithMK - create RSA internal section

CreateRsaInternalSectionWithMK receives an xc internal token and creates an internal CCA RSA key token section by copying the blinding values and encrypting under the master key for the *mk_set* being processed.

CreateRsaInternalSection has the same effect as calling CreateRsaInternalSectionWithMK after setting the *pMKSelector* parameter to (MK_SET_DEFAULT, current_mk, ASYM_MK).

**Note:** This function is only useful for type 2 and type 5 key tokens. Other tokens do not have internal sections.

**Function prototype**

```c
long CreateRsaInternalSectionWithMK ( optx_t *pOptx,
    RsaKeyTokenHeader *pTokenOut,
    mk_selectors *pMKSelector,
    xcRSAKeyToken_t *pRsaTokenIn );

long CreateRsaInternalSection ( optx_t *pOptx,
    RsaKeyTokenHeader *pTokenOut,
    xcRSAKeyToken_t *pRsaTokenIn );
```

**Input**

On entry to this routine:

- *pOptx* is a pointer to the operation tracking structure for this verb.

- *pTokenOut* is a pointer to a variable CCA RSA key token whose private section is either type 2 or type 5. *pMKSelector* is a parameter of type *mk_selectors*, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  
  - *mk_set* is an index to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to MK_SET_DEFAULT.
  
  - *mk_register* is ignored.
  
  - *type_mks* should be set to ASYM_MK.

- *pRsaTokenIn* is a pointer to the internal xcrypto key structure.

**Output**

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

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

- *pTokenOut* contains the extended CCA RSA token.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>mk_KEY_NOT_VALID</td>
<td>The current master key is not valid.</td>
</tr>
<tr>
<td>mk_SEM_CLAIM_FAILED</td>
<td>Could not access the master keys.</td>
</tr>
</tbody>
</table>
8.6.5 GenerateCcaRsaToken - create CCA RSA key token (deprecated)

GenerateCcaRsaToken creates a CCA RSA key token from an internal scc (secure cryptographic coprocessor) key token and a CCA PKA skeleton token. The skeleton token must be initialized to indicate the required format of the final token.

Note: The Toolkit function bld_cca_rsa_from_xcrsa_tok() will create a CCA RSA token from a 4767 format key token. To use this function, you must have the input key token in sccRSAKeyToken_t format, which is unlikely. This function is therefore deprecated.

Function prototype

```c
long GenerateCcaRsaToken ( RsaKeyTokenHeader *pPkaToken, sccRSAKeyToken_t *pRsaKeyToken, short int internal);
```

Note: This function is deprecated. SCC tokens are no longer used by the library functions. Use of the bld_cca_rsa_from_xcrsa_tok() routine is recommended instead.

Input

On entry to this routine:

- `pPkaToken` must be a pointer to a CCA RSA key token header whose nextSection field contains one of the following:
  - `RSA_PRIVATE_SECTION_NOPT` - for a version 0 modulus exponent key with a fixed modulus.
  - `RSA_PRIVATE_SECTION_CR` - for a version 0 Chinese Remainder key.
  - `RSA_PRIVATE_SECTION_NOPT_VAR` - for a version 0 key with a variable length modulus field.
  - `RSA_PRIVATE_SECTION_NOPT_NEW` - for a version 1 modulus exponent key.
  - `RSA_PRIVATE_SECTION_CR_NEW` - for a version 1 Chinese Remainder key.

- `pRsaKeyToken` must be a pointer to a valid internal scc RSA key token.

- `internal` must be initialized to `TRUE` if the key is to be an operational key, or `FALSE` otherwise.

Output

On successful exit from this routine:

- `pPkaToken` contains the key parts in CCA RSA token format of the type desired. You must continue the process by initializing the Object Protection key (if needed) and creating the hashes of various sections and encrypting the appropriate sections of the token.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>ERROR</td>
<td>The skeleton token was not initialized.</td>
</tr>
</tbody>
</table>
8.6.6 GenerateRsaInternalToken - create SCC RSA key token (deprecated)

GenerateRsaInternalToken receives a CCA RSA key token and creates a SCC (secure cryptographic coprocessor) token, which may be translated to a 4767 token.

Note: This function is deprecated. SCC tokens are no longer used by the library functions. Use of the bld_xcrsa_from_cca_rsa_tok() routine is recommended instead.

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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>ERROR</td>
<td>The input key token is not an RSA key token.</td>
</tr>
</tbody>
</table>
8.6.7 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.
8.6.8 GetPublicExponent - extract and copy public exponent

GetPublicExponent extracts the RSA key public exponent from the specified key token, and copies it to the provided buffer.

**Function prototype**

```
USHORT GetPublicExponent ( RsaKeyTokenHeader *pToken,
                           UCHAR *pDest );
```

**Input**

On entry to this routine:

- `pToken` is a pointer to the key token.
- `pDest` is a pointer to the buffer provided for the exponent.

**Output**

On successful exit from this routine:

- `pDest` contains the RSA key public exponent.
- `GetPublicExponent` returns the length of the exponent.

**Return codes**

This function has no return codes.
8.6.9 GetRSAKeySection - get the section containing the name of the key

GetRSAKeySection returns a pointer to the section of the token containing the key name.

Function prototype

```c
void * GetRSAKeySection ( RsaKeyTokenHeader *pRsaKeyToken );
```

Input

On entry to this routine:

`pRsaKeyToken` is a pointer to a valid CCA RSA key token.

Output

On successful exit from this routine:

GetRSAKeySection returns a pointer to the key name section, or `NULL` if there is no such section.

Return codes

This function has no return codes.
8.6.10 **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**

```c
void * GetRsaPrivateKeySection ( RsaKeyTokenHeader *pToken );
```

**Input**

On entry to this routine:

*pToken* is a pointer to the CCA RSA key token.

**Output**

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

*GetRsaPrivateKeySection* returns a pointer to the private key section of a CCA key token.

**Notes:**

Refer to Appendix B of the *IBM CCA Basic Services Reference and Guide* for a diagram of the key token structure.

**Return codes**

This function has no return codes.
8.6.11  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.
8.6.12  GetRsaPuKeyCertSection - return public key certificate

*GetRsaPuKeyCertSection* returns a pointer to the public key section of an RSA key token, if it is present. If not, the function returns a null pointer.

**Function prototype**

```c
Void * GetRsaPuKeyCertSection ( RsaKeyTokenHeader *pToken );
```

**Input**

On entry to this routine:

*pToken* is a pointer to the RSA key token.

**Output**

On successful exit from this routine:

*GetRsaPuKeyCertSection* returns a pointer to the public key section of an RSA key token.

**Return codes**

This function has no return codes.
pka96_tvvgen - calculate token validation value

pka96_tvvgen calculates the token validation value (TVV) for the specified key token.

Function prototype

```c
void pka96_tvvgen ( USHORT token_len,
                    UCHAR *key_token_ptr,
                    uint32_t *tvv );
```

Input

On entry to this routine:

- `token_len` is the length of the token specified with parameter `key_token_ptr`.
- `key_token_ptr` is a pointer to the key token whose TVV is calculated.

Output

On successful exit from this routine:

- `tvv` contains the calculated TVV.

Return codes

This function has no return codes.
8.6.14 RecoverPkaClearKeyTokenAESKW

RecoverPkaClearKeyToken receives a PKA key token which is encrypted under an AES transport key, and recovers the clear form by decrypting the private key areas with AESKW and then verifying the hashes contained in those areas.

Note: This function will be called automatically by the parser when the TOKI KEK NEEDED and TOKI KEK HERE flags are set appropriately.

Function prototype

```c
long RecoverPkaClearKeyTokenAESKW ( optx_t *pOptx,
                               RsaKeyTokenHeader *pTokenIn,
                               optx_tok_t *pAesKey,
                               RsaKeyTokenHeader *pTokenOut );
```

Input

On entry to this routine:

- pOptx is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- pTokenIn is a pointer to the encrypted key token.
- pAesKey is a pointer to an operation tracking structure token which has been decrypted by the parse_keys() function.
- 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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>RT_TKN_UNUSEABLE</td>
<td>The pTokenIn was not an RSA token or did not have a private key section, or the unwrapping failed.</td>
</tr>
</tbody>
</table>
8.6.15 RecoverPkaClearKeyTokenUnderXport

RecoverPkaClearKeyTokenUnderXport receives a PKA key token which is encrypted under a DES transport key, and recovers the clear form by decrypting the private key areas and then verifying the SHA-1 hashes contained in those areas.

**Function prototype**

```c
long RecoverPkaClearKeyTokenUnderXport( opxt_t *pOptx,
                                       RsaKeyTokenHeader *pTokenIn,
                                       double_length_key *desKey,
                                       RsaKeyTokenHeader *pTokenOut );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `pTokenIn` is a pointer to the encrypted key token.
- `desKey` is a pointer to 16 bytes of clear key information, which constitute the double length key used to encipher the RSA key.
- `pTokenOut` is a pointer to a location which can store a key token.

**Output**

On successful exit from this routine:

- `pTokenOut` contains the cleartext key token that it recovers.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>ERROR</td>
<td>The operation failed.</td>
</tr>
</tbody>
</table>
RecoverPkaClearTokenUnderMkWithMKWithMinModulus

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

The allowable minimum modulus length is also verified. The clear key is then added to the on-board cache before being returned to the calling function.

Function prototype

```
long RecoverPkaClearTokenUnderMkWithMKWithMinModulus (
    optx_t *pOptx,
    RsaKeyTokenHeader *pTokenIn,
    RsaKeyTokenHeader *pTokenOut,
    mk_selectors *pMKSelector,
    uint32_t *pMsg,
    long signal );
```

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `pTokenIn` is a pointer to the encrypted key token.
- `pMKSelector` is a parameter of type `mk_selectors`, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is ignored.
  - `type_mks` should be set to `ASYM_MK`.
- `signal` is a constant indicating which type of function the key will be used in:
  - `ORIGINAL_MIN` - Minimum bit length of 512 bits
  - `PKE_DSV_MIN` - Minimum bit length of 128 bits
  - `PKD_MIN` - Minimum bit length of 128 bits for external keys; otherwise, 512 bits is minimum.

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:

RecoverPkaClearTokenUnderMkWithMKWithMinModulus determines which master key was used to RSA functions 213.
encipher the PKA key.

This function does not change the value of byte 28 of the private key, the Key Format and Security byte. If you are planning to store this key in clear form, you should change this byte to the appropriate value before storing. Refer to Appendix B of the *IBM 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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>ERROR</td>
<td>pMsg contains one of the following error codes:</td>
</tr>
<tr>
<td>RT_TKN_UNUSEABLE</td>
<td>The token was not an RSA token.</td>
</tr>
<tr>
<td>RT_KEY_INV_MKVN</td>
<td>The key was encrypted with an invalid master key.</td>
</tr>
<tr>
<td>mk_SRDI_OPEN_ERROR</td>
<td>Could not open the master key.</td>
</tr>
<tr>
<td>mk_SEMCLAIM_FAILED</td>
<td>Unable to access the SRDI Manager.</td>
</tr>
</tbody>
</table>
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.

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

Function prototype

```c
long ReEncipherPkaKeyTokenWithMK ( RsaKeyTokenHeader *pToken,
                                    mk_selectors *pMKSelector,
                                    UCHAR *pWorkArea,
                                    optx_t *pOptx );
```

```c
long ReEncipherPkaKeyToken ( RsaKeyTokenHeader *pToken,
                             mk_selectors *pMKSelector,
                             UCHAR *pWorkArea,
                             optx_t *pOptx );
```

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `pToken` is a pointer to the input key token, enciphered under the old master key.
- `pMKSelector` is a parameter of type `mk_selectors`, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` must be set to `new_mk` or `current_mk`.
  - `type_mks` should be set to `ASYM_MK`.
- `pWorkArea` is a pointer to a variable which can hold a private key. This is used as a work area when decrypting.

Output

On successful exit from this routine:

- `pToken` contains the key token, which has been enciphered under the master key specified by the `pMKSelector->mk_register`.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>ERROR</td>
<td>The input token is not an RSA token.</td>
</tr>
<tr>
<td>RT_MKVP_VFY_FAILED</td>
<td>There was a problem with the Master Key Verification Pattern. Check</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mk_INVALID_LENGTH</td>
<td>The data to be decrypted is not a multiple of 8 bytes.</td>
</tr>
<tr>
<td>mk_INVALID_MKS_TYPE</td>
<td>The <code>mk_type</code> field of the master key selector parameter is not valid.</td>
</tr>
</tbody>
</table>
8.7 *Functions to use the RSA keys*

The following functions are used to access the RSA keys.

<table>
<thead>
<tr>
<th>RSA Key Usage Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>generate_dsig</strong></td>
</tr>
<tr>
<td>Generates a digital signature.</td>
</tr>
<tr>
<td><strong>RequestRSACrypto</strong></td>
</tr>
<tr>
<td>Performs an encryption or decryption operation with the CCA RSA key token.</td>
</tr>
<tr>
<td><strong>verify_dSig</strong></td>
</tr>
<tr>
<td>Verifies an RSA signature.</td>
</tr>
</tbody>
</table>
8.7.1 generate_dSig - generate digital signature

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

**Function prototype**

```c
long generate_dSig ( RsaKeyTokenHeader *pTokenIn,
                    UCHAR *pDataIn,
                    long DataLength,
                    mk_selectors *pMKSelector,
                    UCHAR *pSignatureOut,
                    USHORT *pSignatureBitLength,
                    UCHAR SignatureType,
                    optx_t *pOptx );
```

**Input**

On entry to this routine:

- `pTokenIn` is a pointer to the operational key token.
- `pDataIn` is a pointer to the data which is to be signed.
- `DataLength` is the length of the data to be signed, in bytes.
- `pMKSelector` is a parameter of type `mk_selectors`, indicating which set of master keys to use in this function. This variable must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is allowed on this operating system. Where there is only one set of master keys, this must be set to `MK_SET_DEFAULT`.
  - `mk_register` is ignored.
  - `type_mks` should be set to `ASYM_MK`.
- `pSignatureOut` is a pointer to a buffer which is to hold the returned signature.
- `pSignatureBitLength` is a pointer to the length of the buffer `pSignatureOut`, in bits.
- `SignatureType` is one of the following:
  - `M_ISO9796` if the data is to be formatted according to the ISO-9769 standard before signing.
  - `M_PKCS10` if the data is to be formatted as specified in the RSA Data Security, Inc., *Public Key Cryptography Standards #1* block type 00 before signing.
  - `M_PKCS11` if the data is to be formatted as specified in the RSA Data Security, Inc., *Public Key Cryptography Standards #1* block type 01 before signing.
  - `M_ZEROPAD` if the data is to be placed in the low-order bits of a bit-string of the same length as the modulus with all other bit-positions set to zero before signing.
- `pOptx` is a pointer to the operation tracking structure that was input to the command processor that calls this routine. The operation tracking structure is required for the random number generator handle it contains. This parameter is only used for UDX toolkit level 4.3 and above.
Output

On successful exit from this routine:

The `mk_registers` field of the `pMKSelectors` parameter has been set to indicate which master key was used to decrypt the provided `pTokenIn` parameter.

`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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>E_SIZE</td>
<td>The provided buffer was not large enough to contain the signature.</td>
</tr>
<tr>
<td>PKABadAddr</td>
<td>The key token is not valid.</td>
</tr>
</tbody>
</table>
8.7.2 RequestRSACrypto - perform an RSA operation

RequestRSACrypto converts the specified CCA RSA key token to the RSA internal key token format that the RSA engine requires, and then requests that the RSA engine perform the specified RSA function.

Note: Prior to using this routine, ensure that you’ve deciphered the private key (if you’re using it) using the routine RecoverPkaClearTokenUnderMkWithMk() or use the ckey field from the optx_tok type if you have used the parser to decrypt your keys.

Function prototype

```c
uint32_t RequestRSACrypto ( opxt_t *pOptx,
                          void *pInput,
                          RsaKeyTokenHeader *pKeyToken,
                          void *pOutput,
                          uint32_t DataBitLength,
                          uint32_t RsaOperation );
```

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `pInput` is a pointer to the input data for the RSA operation.
- `pKeyToken` is a pointer to the key token for the RSA key. This is a clear CCA format RSA key token.
- `DataBitLength` is the length of the input data, in bits. This number is presumed to be equal to the length of the output data buffer, in bits. If this number is larger than the modulus length in bits, the data which will be operated on is in the rightmost modulus length bits of the input data buffer, and the result will be placed in the rightmost modulus length bits of the output data buffer.
- `RsaOperation` is the requested RSA operation, such as `RSA_ENCRYPT` (public key operation) or `RSA_DECRYPT` (private key operation).

Output

On successful exit from this routine:

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

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>ERROR</td>
<td>Could not create a buffer to receive the RSA key token.</td>
</tr>
<tr>
<td>E_SIZE</td>
<td>The data is larger than the modulus.</td>
</tr>
<tr>
<td>PKABadAddr</td>
<td>The key token is not valid.</td>
</tr>
<tr>
<td>PKANoSpace</td>
<td>Unable to allocate sufficient memory.</td>
</tr>
</tbody>
</table>

220 IBM 4767 CCA UDX Reference
8.7.3 verify_dSig - verify digital 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,
                  mk_selectors *pMKSelector,
                  USHORT SignatureLength,
                  UCHAR Type,
                  optx_t *pOptx );
```

Input

On entry to this routine:

- `pTokenIn` is a pointer to the operational key token.
- `pDataIn` is a pointer to the data which is to be hashed and compared with the encrypted signature.
- `DataLength` is the length of the data to be signed, in bytes.
- `pDigitalSignature` is a pointer to a buffer which contains the signature to be verified.
- `pMKSelector` is a parameter of type `mk_selectors`, indicating which set of master keys to use in this
  function. This variable must be initialized as follows:
  - `mk_set` is a pointer to the set of master keys which is to be accessed, if more than one set is
    allowed on this operating system. Where there is only one set of master keys, this must be set to
    `MK_SET_DEFAULT`.
  - `mk_register` is ignored.
  - `type_mks` should be set to `ASYM_MK`.
- `SignatureLength` is the length of the buffer `pDigitalSignature`, in bytes.
- `Type` is one of the following:
  - `M_ISO9796` if the data was formatted according to the ISO-9796 standard before signing.
  - `M_PKCS1` if the data was formatted as specified in the RSA Data Security, Inc., Public Key
    Cryptography Standards #1 block type 00 before signing.
  - `M_PKCS11` if the data was formatted as specified in the RSA Data Security, Inc., Public Key
    Cryptography Standards #1 block type 01 before signing.
  - `M_ZEROPAD` if the data was 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.
- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls
  this routine. The operation tracking structure is required for the random number generator handle it
  contains. This parameter is only used for UDX toolkit level 4.3 and above.
Output
On output, the `mk_registers` field of the `pMKSelectors` parameter has been set to indicate which master key was used to decrypt the provided `pTokenIn` parameter.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>DSIG_NOT_VERIFIED</td>
<td>The digital signature was not verified.</td>
</tr>
<tr>
<td>E_SIZE</td>
<td>The provided buffer was not large enough to contain the signature.</td>
</tr>
<tr>
<td>PKABadAddr</td>
<td>The key token is not valid.</td>
</tr>
</tbody>
</table>
8.8 Functions to handle the on-board key storage

The following functions are used to store, recover, and delete RSA keys from the coprocessor’s “Retained Keys” file.

<table>
<thead>
<tr>
<th>RSA On-Board Key Storage Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>delete_KeyToken</strong></td>
</tr>
<tr>
<td><strong>getKeyToken</strong></td>
</tr>
<tr>
<td><strong>store_KeyToken</strong></td>
</tr>
</tbody>
</table>
8.8.1 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**

```c
uint32_t delete_KeyToken ( optx_t *pOptx, char *pKeyName, uint16_t *pDomain );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `pKeyName` is a pointer to a 64 byte array containing the name of the key to be deleted.
- `pDomain` is the value in `pOptx->Domain`. For a workstation UDX, this must be 0.

**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:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>KEY_NAME_NOTFOUND</td>
<td>The key was not found in the list.</td>
</tr>
<tr>
<td>CP_MEMORY_NAVAIL</td>
<td>Out of memory error.</td>
</tr>
<tr>
<td>PKEY_SRDI_ERROR</td>
<td>Unable to access the key storage SRDI.</td>
</tr>
</tbody>
</table>
8.8.2 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

```
uint32_t getKeyToken ( optx_t *pOptx, char *pLabel, char *pKey, uint16_t *pFlags, uint16_t *pDomain );
```

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls 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.
- `pDomain` is the value from `pOptx->Domain`. For a workstation UDX, this must be 0.

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:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srdi_NO_ERROR</td>
<td>The command completed successfully.</td>
</tr>
<tr>
<td>PKEY_NOT_REGISTER</td>
<td>The key was not found.</td>
</tr>
<tr>
<td>PKEY_SRDI_ERROR</td>
<td>The registered key manager could not be accessed.</td>
</tr>
</tbody>
</table>

RSA functions 225
8.8.3 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**

```c
uint32_t store_KeyToken ( optx_t *pOptx,
                          KEY_register_data_t *pKey,
                          uint16_t *pDomain );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls 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.
  - `reserved` - This short variable must be initialized to 0.
  - `length` - The length of this record, in big-endian format.
  - `label` - Contains a 64-byte key name.
  - `flags` - Valued to `CCA_CLONE` if this key is allowed to participate in master key cloning operations, or 0 otherwise.
  - `keydata` - The beginning of the actual key token.
- `pDomain` is the value from `pOptx->Domain`. For a workstation UDX, this must be 0.

**Output**

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

The KeyRetain SRDI has been expanded to include the data stored in `pKey`.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>srdf_NO_ERROR</code></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><code>CP_MEMORY_NAVAIL</code></td>
<td>Out of memory error.</td>
</tr>
<tr>
<td><code>PKEY_SRDI_ERROR</code></td>
<td>Unable to access the key storage SRDI.</td>
</tr>
<tr>
<td><code>DUPLICATE_NAME</code></td>
<td>A key with the same name is already registered.</td>
</tr>
</tbody>
</table>
9 CCA SRDI manager functions

This chapter describes the CCA SRDI Manager, which manages the storage and retrieval of persistent data in the coprocessor.

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

9.1 Header files for SRDI manager functions

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

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

9.2 Overview

The security relevant data item (SRDI)\(^1\) Manager is the interface through which CCA-related functions access security related data. Only the SRDI Manager interacts with the BBRAM in which the SRDI data is stored. The CCA verbs and any other CCA code read and write SRDI information through the SRDI Manager interface. In turn, the SRDI Manager accesses the physical SRDI storage through the Linux BBRAM Filesystem Manager, which controls the Battery-Backed RAM memory. This relationship is shown in Figure 6.

---

\(^1\) SRDIs are the sensitive data elements owned by the cryptographic application, and requiring protection. Examples include cryptographic keys and access control profiles.
Encapsulation of the SRDI physical storage mechanism makes it possible to change that mechanism without any effect on the CCA application code.

Each SRDI is identified by a name, much like a file name. The SRDI name is an eight character ASCII string, with no null terminator. SRDIs created by a UDX must have names which begin with the letters “UDX”, followed by a 4-digit number, followed by one ASCII space, for example “UDX0001”. The number must be unique and must not be “0000”. The SRDI name is used to handle the naming of the semaphore associated with the SRDI. Therefore, you must use this naming convention for your SRDIs.

9.3 CCA SRDI manager operation

The Linux JFFS2 Filesystem Manager stores SRDI data in the BBRAM. BBRAM is small, but fast, and it has no limitations on the number of times it can be written.

CCA applications do not have direct access to the SRDI information stored in BBRAM. When a SRDI is opened, the SRDI Manager creates a 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 a 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.

9.3.1 An example: opening a SRDI

Figure 7, Figure 8, and Figure 9, which are interspersed through this section, describe the steps when a 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 a SRDI named UDX0025, which resides in BBRAM. At this time, SRDI UDX0025 is not open. No copy of the SRDI data exists in the CCA application RAM address space.
2. The CCA Manager uses fstat() to determine the length of SRDI UDX0025. It needs to know the length, so it can allocate the required buffer in RAM.
3. The CCA SRDI Manager allocates a buffer to hold UDX0025. This buffer is in RAM addressable by the CCA application.

Persistent memories are those that preserve their contents even when power is turned off. In the coprocessor, the BBRAM is persistent. The main system RAM used for executing programs and their data is not persistent.

IBM 4767 CCA UDX Reference
Step 4. The CCA SRDI Manager sends a request to the Linux Filesystem Manager, asking it to read UDX0025 into the buffer allocated in step 3.

5. The SRDI is read from BBRAM, decrypted, and deposited in the specified buffer.

Figure 7 Master SRDI read illustration part 1

Figure 8 Master SRDI read illustration part 2
Step | Description
--- | ---
6. | The CCA SRDI Manager returns the buffer address to the CCA command processor. The command processor then uses the RAM copy of the SRDI whenever it needs to read or alter UDX0025.

Since the CCA application uses the fiber-management method to allow concurrent operations, different callers may access a 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.

CCA uses the Wait on Quiesce queue to prevent this from occurring. Each verb which modifies a SRDI is created as a "wait on quiesce" command processor, so that it runs only when all other command processors are completed.

For a UDX to add verbs to the wait on quiesce list, set the run_type of in the ccax_cp_list array to RUN_ISOLATED.

If a UDX is to create its own SRDIs, Fiber Mutexes may be employed to handle concurrency issues. Every SRDI user in the UDX application should be required to gain ownership of the fiber mutex for that SRDI 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 fiber mutex is released so that others can use the SRDI.

The fiber mutexes are managed through the xcrypto library functions xFbrMutex_LOCK and xFbrMutex_UNLOCK. See the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference for more information.

9.4 Using the BBRAM without the SRDI manager

The BBRAM medium is mounted as a filesystem at /bbram. It is possible to use the BBRAM memory directly by using the Linux commands open, write, and read on files in this directory. Filenames in /bbram are restricted to 14 characters, and no subdirectories may be created in the /bbram directory. You will
also be responsible for controlling concurrent access to your data. This may be done either by making command processors that modify SRDI data `RUN_ISOLATED` routines or by the use of fiber mutexes.

### 9.5 Using the flash storage

The flash available for user storage is mounted as a filesystem at `/flashS3`. It is possible to use the flash memory directly by using the Linux commands open, write, and read on files in this directory. You will also be responsible for controlling concurrent access to your data.

### 9.6 Summary of functions

These functions are used by the CCA command processor to read and write SRDI data.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>close_cca_srdi</code></td>
<td>Closes the open copy of a SRDI.</td>
</tr>
<tr>
<td><code>create_cca_srdi</code></td>
<td>Creates a SRDI.</td>
</tr>
<tr>
<td><code>delete_cca_srdi</code></td>
<td>Deletes a SRDI from memory.</td>
</tr>
<tr>
<td><code>get_cca_srdi_length</code></td>
<td>Obtains the length of a SRDI, in bytes.</td>
</tr>
<tr>
<td><code>open_cca_srdi</code></td>
<td>Opens and gains access to a SRDI.</td>
</tr>
<tr>
<td><code>resize_cca_srdi</code></td>
<td>Increases or decreases the length of a SRDI, in bytes.</td>
</tr>
<tr>
<td><code>save_cca_srdi</code></td>
<td>Stores SRDI data.</td>
</tr>
</tbody>
</table>
9.6.1 close_cca_SRDI - close CCA SRDI

close_cca_SRDI deactivates the open copy of the SRDI, which is managed by the SRDI Manager. If no other applications are using the SRDI, the RAM which held the working copy of the SRDI is released.

Notes:

- If the working copy of the SRDI has been changed, the application must issue the save_cca_SRDI function in order to have the SRDI saved. SRDI data is not automatically saved when the SRDI is closed.

- The close_cca_SRDI function does not release the semaphore which was locked by the open_cca_SRDI function. This must be done by the user, prior to closing the SRDI.

Function prototype

```c
uint32_t 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:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srdi_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>srdi_NOT_OPEN</td>
<td>The SRDI is not open, and therefore cannot be closed.</td>
</tr>
<tr>
<td>srdi_GENERAL_ERROR</td>
<td>Failed to access the SRDI manager.</td>
</tr>
</tbody>
</table>
9.6.2 `create_cca_srdi` - create CCA SRDI

`create_cca_srdi` creates a SRDI in BBRAM using the specified name.

**Function prototype**

```c
uint32_t create_cca_srdi ( optx_t *pOptx,
                          char *srdi_name,
                          char *srdi_addr,
                          uint32_t srdi_length );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

- `srdi_name` is a pointer to an eight character ASCII string containing the name of the desired SRDI. This string does not contain a null terminator, and should be padded on the right with blanks. It should begin with “UDX”, followed by 4 decimal digits.

- `srdi_addr` is a pointer to the address of the SRDI data. This data is written to the newly created SRDI.

- `srdi_length` is the length of the SRDI data, in bytes.

**Output**

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

`create_cca_srdi` creates a SRDI in BBRAM using the specified name.

Data is encrypted in BBRAM.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srdi_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>srdi_EXISTS</td>
<td>The SRDI already exists.</td>
</tr>
<tr>
<td>srdi_GENERAL_ERROR</td>
<td>Failed to access the SRDI manager.</td>
</tr>
</tbody>
</table>
9.6.3 delete_cca_srdi - delete CCA SRDI

`delete_cca_srdi` deletes a SRDI from the persistent memory area where it is stored. This is equivalent to erasing a file from a hard disk.

**Note:** A SRDI cannot be deleted if it is in the “open” state, since another application may be using it.

**Function prototype**

```c
uint32_t 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:

The SRDI named `srdi_name` no longer exists in the persistent memory area where it was stored.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>srdi_NO_ERROR</th>
<th>The operation was successful.</th>
</tr>
</thead>
<tbody>
<tr>
<td>srdi_NOT_FOUND</td>
<td>The SRDI does not exist.</td>
</tr>
<tr>
<td>srdi_GENERAL_ERROR</td>
<td>Failed to access the SRDI manager.</td>
</tr>
<tr>
<td>srdi_OPEN</td>
<td>The SRDI item is not in the closed state.</td>
</tr>
</tbody>
</table>
9.6.4  get_cca_srdi_length - get CCA SRDI length

get_cca_srdi_length obtains the length of the specified SRDI, in bytes.

Function prototype

```c
uint32_t get_cca_srdi_length ( char *srdi_name,
                                uint32_t *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 `uint32_t` variable in which the length will be stored.

Output

On successful exit from this routine:

- `srdi_length` contains the length of the SRDI data, in bytes.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>srdi_NO_ERROR</th>
<th>The operation was successful.</th>
</tr>
</thead>
<tbody>
<tr>
<td>srdi_READ_ERROR</td>
<td>Unable to read the SRDI item from BBRAM.</td>
</tr>
<tr>
<td>srdi_GENERAL_ERROR</td>
<td>Failed to access the SRDI manager.</td>
</tr>
</tbody>
</table>
9.6.5 open_cca_srdi - open CCA SRDI

open_cca_srdi opens a SRDI, gaining access to its contents. The function returns the address and length of the SRDI data, where the address points to a working copy of the actual SRDI, which is stored in 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, which has been locked by the open_cca_srdi function. The SRDI Manager names the semaphore according to the 4 digits which follow “UDX” in the name of the SRDI item; therefore, these digits should be unique to a given SRDI. This semaphore is used to gain exclusive access to the SRDI, to prevent errors when one thread is writing data, while another is simultaneously either reading or writing that same data.

**Note:** The semaphore for the SRDI is deprecated, since the application runs single-threaded with concurrency handled by the CCA UCT and fiber mutexes. However, you must still use the LIBR_ReleaseSemaphore command to deal with them.

See “Master SRDI read illustration part 3 - controlling concurrent access to a SRDI” on page 230 for further details.

**Function prototype**

```c
uint32_t open_cca_srdi ( optx_t *pOptx, 
    char *srdi_name, 
    char **srdi_addr, 
    uint32_t *srdi_length, 
    uint32_t *semSVid );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

- `srdi_name` is a pointer to an eight character ASCII string containing the name of the desired SRDI. This string does not contain a null terminator. The name should consist of the ASCII string “UDX” concatenated with a string of 4 decimal digits. The final character may be any character which is valid in a file name, or a space.

**Output**

On successful exit from this routine:

- `srdi_addr` is a pointer to a pointer variable, in which the SRDI Manager returns the address of the SRDI. This is an address in RAM, where the SRDI Manager places a copy of the SRDI data.

- `srdi_length` is a pointer to a location where the SRDI Manager stores the length of the SRDI data, in bytes.

- `semSVid` is the SVid for the semaphore assigned to the specified SRDI.

**Return codes**

Common return codes generated by this routine are:

236 IBM 4767 CCA UDX Reference
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srdi_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>srdi_NOT_FOUND</td>
<td>The SRDI does not exist.</td>
</tr>
<tr>
<td>srdi_READ_ERROR</td>
<td>Unable to read the SRDI item from BBRAM.</td>
</tr>
<tr>
<td>srdi_ALLOC_ERROR</td>
<td>Unable to allocate memory for the SRDI item.</td>
</tr>
<tr>
<td>srdi_GENERAL_ERROR</td>
<td>Failed to access the SRDI manager.</td>
</tr>
</tbody>
</table>
9.6.6 resize_cca_srdi - resize CCA SRDI

resize_cca_srdi increases or decreases the length of the specified CCA SRDI, in bytes.

A SRDI can only be resized if you are the only requestor who has it open. If the SRDI is opened by more than one user concurrently, it cannot be resized; the address of the RAM copy changes when it is resized, and there is no way to notify other callers of this.

Function prototype

```c
uint32_t resize_cca_srdi( char *srdi_name,
                          uint32_t 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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srdi_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>srdi_NOT_FOUND</td>
<td>The SRDI does not exist.</td>
</tr>
<tr>
<td>srdi_READ_ERROR</td>
<td>Unable to read the SRDI item from BBRAM.</td>
</tr>
<tr>
<td>srdi_ALLOC_ERROR</td>
<td>Unable to allocate memory for the SRDI item.</td>
</tr>
<tr>
<td>srdi_GENERAL_ERROR</td>
<td>Failed to access the SRDI manager.</td>
</tr>
</tbody>
</table>
9.6.7 save_cca_srdi - save CCA SRDI

save_cca_srdi stores the SRDI data on a persistent storage medium (BBRAM).

This function ensures that no thread is updating the SRDI while it is being stored by gaining exclusive access rights using the SRDI semaphore.

See “Master SRDI read illustration part 3 - controlling concurrent access to a SRDI” on page 230 for details on this semaphore.

Function prototype

\[
\text{uint32_t save_cca_srdi ( optx_t *pOptx, char *srdi_name );}
\]

Input

On entry to this routine:

- \( pOptx \) is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- \( srdi\_name \) is a pointer to an eight character ASCII string containing the name of the desired SRDI. This string does not contain a null terminator.

Note: No two SRDIs can have the same name. The CCA SRDI Manager enforces this restriction.

Output

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

- \( \text{save_cca_srdi} \) stores the SRDI data on a persistent storage medium.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srdi_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>srdi_NOT_OPEN</td>
<td>The SRDI is not in the open state.</td>
</tr>
<tr>
<td>srdi_WRITE_ERROR</td>
<td>Unable to write to BBRAM.</td>
</tr>
<tr>
<td>srdi_ALLOC_ERROR</td>
<td>Unable to allocate memory for the SRDI item.</td>
</tr>
<tr>
<td>srdi_GENERAL_ERROR</td>
<td>Failed to access the SRDI manager.</td>
</tr>
</tbody>
</table>
9.6.8 Example SRDI code

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

```c
#define MY_SRDI_NAME "UDX0001 " /* Name of SRDI we're using */
uint32_t srdi_rc; /* SRDI fcn. return code */
char * my_srdi_addr; /* Pointer to clear SRDI data in RAM */
uint32_t my_srdi_length; /* Length of SRDI data, in bytes */
uint32_t semaphore_id; /* SVid of SRDI access semaphore */

void srdi_stuff ( void )
{
    /* Open the SRDI */
    srdi_rc = open_cca_srdi ( MY_SRDI_NAME,
                             &my_srdi_addr,
                             &my_srdi_length,
                             &semaphore_id);
    if ( srdi_NO_ERROR == srdi_rc ) /* If no errors opening SRDI... */
        { /* do other stuff as needed... */
          /* This is where the code will read and/or write */
          /* to the SRDI data, in the area pointed */
          /* to by my_srdi_addr. */
          memcpy( my_srdi_addr, newdata, my_srdi_length );
          srdi_rc = save_cca_srdi( MY_SRDI_NAME );
          /* Release semaphore, allowing others to access the SRDI */
          LIBR_ReleaseSemaphore( semaphore_id );
          if ( srdi_rc != srdi_NO_ERROR )
              { /* do stuff relating to write error for SRDI */
              }
        } /* do other stuff as needed... */
    /* suppose we later want to read what was in the srdi */
    if ( OK == LIBR_RequestSemaphore( semaphore_id ) )
        { /* do whatever 'no change' means */
        } else
            { /* do whatever 'changed' means */
              LIBR_ReleaseSemaphore( semaphore_id );
            }
    else
        { /* handle semaphore request error */
        }
    /* Close the SRDI */
    srdi_rc = close_cca_srdi( MY_SRDI_NAME );
    if ( srdi_NO_ERROR != srdi_rc )
        { /* handle SRDI close error... */
        }
}
```

240 IBM 4767 CCA UDX Reference
10 Access control manager functions

The functions described in this chapter enable a UDX to initialize or update access control tables, log users on or off of the coprocessor, or check a user’s authority to perform a specified function.

10.1 Header files for access control manager functions

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

```c
#include "camxtrn.h" /* CCA managers */
#include "camacm.h" /* Access Control Manager */
#include "camacm_p.h" /* Access Control Manager */
```

10.2 Summary of functions

Access Control Manager includes the following functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac_check_authorization</td>
<td>Check whether a user is authorized to execute a specified function.</td>
<td>User</td>
</tr>
<tr>
<td>ac_chg_prof_auth_data</td>
<td>Change the authentication data for a user profile.</td>
<td>Profile</td>
</tr>
<tr>
<td>ac_chg_prof_exp_date</td>
<td>Change the expiration date of a user profile.</td>
<td>Profile</td>
</tr>
<tr>
<td>ac_del_profile</td>
<td>Delete a user profile.</td>
<td>Profile</td>
</tr>
<tr>
<td>ac_del_role</td>
<td>Delete a role definition.</td>
<td>Role</td>
</tr>
<tr>
<td>ac_get_list_sizes</td>
<td>Find out how many roles and how many profiles exist on the coprocessor.</td>
<td>General</td>
</tr>
<tr>
<td>ac_get_profile</td>
<td>Read the contents of a specified user profile.</td>
<td>Profile</td>
</tr>
<tr>
<td>ac_get_role</td>
<td>Read the contents of a specified role.</td>
<td>Role</td>
</tr>
<tr>
<td>ac_init</td>
<td>Initialize the access control system, including setup of role and profile tables.</td>
<td>General</td>
</tr>
<tr>
<td>ac_list_profiles</td>
<td>Get a list of all the user IDs for which there are profiles.</td>
<td>Profile</td>
</tr>
<tr>
<td>ac_list_roles</td>
<td>Get a list of all the role IDs.</td>
<td>Role</td>
</tr>
<tr>
<td>ac_load_profiles</td>
<td>Load one or more user profiles.</td>
<td>Profile</td>
</tr>
<tr>
<td>ac_load_roles</td>
<td>Load one or more role definitions.</td>
<td>Role</td>
</tr>
<tr>
<td>ac_lu_add_user</td>
<td>Add a user to the table of logged-on users, and generate a session key for that user.</td>
<td>User</td>
</tr>
<tr>
<td>ac_lu_drop_user</td>
<td>Drop a user from the table of logged-on users.</td>
<td>User</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Type</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>ac_lu_get_ks</td>
<td>Get a copy of a specified user’s session key.</td>
<td>User</td>
</tr>
<tr>
<td>ac_lu_get_num_users</td>
<td>Find out how many users are currently logged on.</td>
<td>User</td>
</tr>
<tr>
<td>ac_lu_get_role</td>
<td>Get the role ID for a specified logged-on user.</td>
<td>User</td>
</tr>
<tr>
<td>ac_lu_ks_dec</td>
<td>Decrypt data with a specified user’s session key.</td>
<td>User</td>
</tr>
<tr>
<td>ac_lu_ks_hmacgen</td>
<td>Computes a HMAC on a message using the session key.</td>
<td>User</td>
</tr>
<tr>
<td>ac_lu_ks_hmacver</td>
<td>Verifies an HMAC using the session key.</td>
<td>User</td>
</tr>
<tr>
<td>ac_lu_list_users</td>
<td>Get a list of all the logged-on users.</td>
<td>User</td>
</tr>
<tr>
<td>ac_lu_query_user</td>
<td>Check whether a specified user is currently logged on (authenticated).</td>
<td>User</td>
</tr>
<tr>
<td>ac_query_profile</td>
<td>Verify that a specified profile exists on the coprocessor, and return its length.</td>
<td>Profile</td>
</tr>
<tr>
<td>ac_query_role</td>
<td>Verify that a specified role exists on the coprocessor.</td>
<td>Role</td>
</tr>
<tr>
<td>ac_reinit</td>
<td>Reinitialize the access control system, to the default state.</td>
<td>General</td>
</tr>
<tr>
<td>ac_reset_logon_fail_cnt</td>
<td>Reset the logon failure count in a user profile.</td>
<td>Profile</td>
</tr>
<tr>
<td>acm_xlt_errorcode</td>
<td>Translates an ACM error code to an SAPI error code.</td>
<td>General</td>
</tr>
<tr>
<td>update_logon_failure_count</td>
<td>Change the logon failure count for a user profile.</td>
<td>Profile</td>
</tr>
</tbody>
</table>
10.2.1   **SRDI files for the access control manager**

The Access Control Manager uses two SRDI files, stored in BBRAM.

- *Profiles* - holds all the user profiles that are enrolled on the coprocessor.
- *Roles* - holds all the roles that have been defined on the coprocessor.

Each of these SRDIs has a similar format. The file begins with a 4-byte header, in which the first two bytes contain an integer specifying how many items are in the file, and the second two bytes are reserved and set to X’0000’. The number of items reflects either the number of profiles or the number of roles, depending on the SRDI in question.

The profiles or roles follow the header. The first one is concatenated to the end of the header, and each successive role or profile is concatenated to its predecessor. Neither roles nor profiles are ordered in any meaningful way.

For quicker access, the Access Control Manager builds indexes in RAM for the roles and the profiles. For each role or profile in the SRDIs, the respective index table holds the name of the role or profile and its offset from the start of the SRDI.
### 10.3 Functions for general use with the access control manager

These functions are used with all of the access control SRDIs and tables.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac_get_list_sizes</td>
<td>Find out how many roles and how many profiles exist on the coprocessor.</td>
<td>General</td>
</tr>
<tr>
<td>ac_init</td>
<td>Initialize the access control system, including setup of role and profile tables.</td>
<td>General</td>
</tr>
<tr>
<td>ac_reinit</td>
<td>Reinitialize the access control system, to the default state.</td>
<td>General</td>
</tr>
<tr>
<td>acm_xlt_errorcode</td>
<td>Translates an ACM error code to an SAPI error code.</td>
<td>General</td>
</tr>
</tbody>
</table>
10.3.1  **ac_get_list_sizes - get sizes of role and profile lists**

`ac_get_list_sizes` is used to find out how much data will be returned by the `ac_list_profiles` and `ac_list_roles` functions. By calling this function first, the application can ensure it has a large enough buffer for the data it will receive.

**Function prototype**

```c
long ac_get_list_sizes ( uint32_t *num_profiles,
                        uint32_t *num_roles );
```

**Input**

On input to this routine:

- `num_profiles` is a pointer to a variable that will receive the number of profiles on the coprocessor.
- `num_roles` is a pointer to a variable that will receive the number of roles on the coprocessor.

**Output**

On successful exit from this routine:

- `num_profiles` contains the number of profiles present.
- `num_roles` contains the number of roles present.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>acm_NO_ERROR</th>
<th>The operation was successful.</th>
</tr>
</thead>
</table>
10.3.2 ac_init - initialize the access control manager

ac_init initializes the Access Control Manager including all data areas and state information. In addition, creates and initializes the role and profile SRDIs if they do not already exist, including creation of a DEFAULT role.

Function prototype

```c
uint32_t ac_init ( optx_t *pOptx );
```

Input

On input to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

Output

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

- `ac_init` creates and initializes the Access Control Manager.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acm_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
</tbody>
</table>
10.3.3 ac_reinit - reinitialize the access control manager

ac_reinit reinitializes the Access Control Manager, deletes all existing roles, profiles, and other data, and reinitializes to the access control state of a new coprocessor.

Function prototype

```c
uint32_t ac_reinit ( optx_t *pOptx );
```

Input

On input to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

Output

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

- `ac_reinit` reinitializes the Access Control Manager.

Return codes

Common return codes generated by this routine are:

| acm_NO_ERROR            | The operation was successful. |
10.3.4  acm_xlt_errorcode - translate an ACM error code to a SAPI error code

acm_xlt_errorcode translates an ACM error code (defined in camacm.h) to an SAPI error code (defined in cmnerrcd.h).

Function prototype

    int acm_xlt_errorcode ( long acm_code,
                                long *sapi_code );

Input

On entry to this routine:

acm_code contains the error code returned by the Access Control Manager.

sapi_code is a pointer to a 4-byte buffer, which will hold the translated code.

Output

On successful exit from this routine:

acm_xlt_errorcode returns TRUE if the provided ACM code was found, and FALSE otherwise.

If acm_xlt_errorcode returned TRUE, sapi_code contains the corresponding SAPI error code.

Return codes

This function has no return codes.
10.4 **Functions to manage roles**

Functions in this section are used to manage the Roles and RoleIDs in the system.

**Note:** since these functions involve writing to the SRDI and a UDX user cannot guarantee that a CCA routine is not accessing the roles or profiles while the operation is taking place, it is recommended that the use of these routines be avoided.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac_del_role</td>
<td>Delete a role definition.</td>
<td>Role</td>
</tr>
<tr>
<td>ac_get_role</td>
<td>Read the contents of a specified role.</td>
<td>Role</td>
</tr>
<tr>
<td>ac_list_roles</td>
<td>Get a list of all the role IDs.</td>
<td>Role</td>
</tr>
<tr>
<td>ac_load_roles</td>
<td>Load one or more role definitions.</td>
<td>Role</td>
</tr>
<tr>
<td>ac_query_role</td>
<td>Verify that a specified role exists on the coprocessor.</td>
<td>Role</td>
</tr>
</tbody>
</table>
10.4.1 ac_del_role - delete role

`ac_del_role` deletes a specified role definition from the coprocessor. In addition, if any logged on users have their authority defined by this role, those users are logged off.

If the request is to delete the DEFAULT role, the role is not actually deleted; the default role must always be present on the coprocessor. Instead, it is reset to the default values for that role.

**Function prototype**

```c
uint32_t ac_del_role ( optx_t *pOptx, role_id_t role_id );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `role_id` is the 8-byte, non-NULL terminated ID of the role to be deleted.

**Output**

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

The specified role has been deleted, unless the role specified was DEFAULT. If the specified role was DEFAULT, the Default role has been reset to its original contents. The list of roles has been saved to BBRAM.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acm_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>acm_ROLE_NOT_FOUND</td>
<td>The specified role was not in the role list.</td>
</tr>
</tbody>
</table>
10.4.2 ac_get_role - get role

ac_get_role returns the information from a specified role.

Function prototype

```c
uint32_t ac_get_role ( role_id_t role_id,
                       USHORT *role_size,
                       void *role_data );
```

Input

On entry to this routine:

- `role_id` is an 8-byte character string non-NULL terminated containing the name of the role to be returned.
- `role_size` is a pointer to a variable which contains the size of the `role_data` buffer, in bytes.
- `role_data` is a pointer to a buffer where the role data will be stored.

Output

On successful exit from this routine:

- `role_size` contains the actual size of the returned role data, or the size needed for the role data if the buffer provided was not sufficient.
- `role_data` contains the role data.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acm_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>acm_BFR_TOO_SMALL</td>
<td>The array is not large enough to hold all the role data. The buffer size</td>
</tr>
<tr>
<td></td>
<td>required is in the <code>role_size</code> variable.</td>
</tr>
<tr>
<td>acm_ROLE_NOT_FOUND</td>
<td>The specified role was not found.</td>
</tr>
</tbody>
</table>
10.4.3 ac_list_roles - list roles

`ac_list_roles` returns a list of the role IDs for each role defined on the coprocessor. This is a series of unordered 8-byte role IDs.

**Function prototype**

```c
long ac_list_roles ( role_id_t role_list [],
                    uint32_t *num_roles );
```

**Input**

On entry to this routine:

- `role_list` is a pointer to an array to hold the list of role IDs.
- `num_roles` is a pointer to a variable containing the maximum number of 8-byte IDs which the array can hold.

**Output**

On successful exit from this routine:

- `role_list` contains the list of role IDs.
- `num_roles` contains the number of role IDs returned.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>acm_NO_ERROR</code></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><code>acm_BFR_TOO_SMALL</code></td>
<td>The array is not large enough to hold all the role IDs.</td>
</tr>
</tbody>
</table>
10.4.4 ac_load_roles - load roles

*ac_load_roles* loads one or more roles into the coprocessor. The roles are passed in an aggregate structure, which can contain any number of concatenated role definitions.

### Function prototype

```c
uint32_t ac_load_roles ( optx_t *pOptx, void *role_list );
```

### Input

On entry to this routine:

- **pOptx** is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- **role_list** is a pointer to a variable length buffer of the following form:
  - a 4-byte variable containing the number of roles in this list followed by
  - a 4-byte variable containing 0 followed by
  - the first role followed by
  - any roles in between followed by
  - the last role

Each role is a variable length role initialized to the following:

A *role_t* variable containing:

- **role_vers** - the role structure version, X'01'
- **role_lth** - the number of bytes in this role
- **comment** - a 20-byte comment, NULL-terminated, describing this role
- **cksum** - checksum, for determining role integrity
- **role_id** - an 8-byte, non-NULL terminated string, the ID for this role.
- **reqd_auth_str** - the required authentication strength for this role. Each method of authentication is assigned a strength value, with 0 being no authentication. A role may be restricted to users who have logged on with a more stringent method (that is, passphrase rather than PIN) if more than one method has been supplied.

- **lower_time_limit**
- **upper_time_limit**
- **valid_dow**

Followed by an access control point list for the role.

### Output

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

The new role is added to the existing set of roles. If the role exists, it is replaced with the new role.
**Return codes**

Common return codes generated by this routine are:

| acm_NO_ERROR        | The operation was successful. |
10.4.5    ac_query_role - return the length of a role
ac_query_role returns the length of a specified role, or zero if the role does not exist on the coprocessor.

Function prototype

```c
uint32_t ac_query_role ( role_id_t role_id,
                       USHORT *role_length );
```

Input

On input to this routine:

- `role_id` is an 8-byte, non-NULL terminated ID of the role to be queried.
- `role_length` is a pointer to a variable which receives the length of the specified role, in bytes.

Output

On successful exit from this routine:

- `role_length` contains the length, in bytes. If the role does not exist, a length of zero is returned.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>acm_NO_ERROR</code></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><code>acm_ROLE_NOT_FOUND</code></td>
<td>The specified role was not in the role list.</td>
</tr>
</tbody>
</table>
## 10.5 Functions to manage profiles

These functions are used to manage user Profiles.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac_chg_prof_auth_data</td>
<td>Change the authentication data for a user profile.</td>
<td>Profile</td>
</tr>
<tr>
<td>ac_chg_prof_exp_date</td>
<td>Change the expiration date of a user profile.</td>
<td>Profile</td>
</tr>
<tr>
<td>ac_del_profile</td>
<td>Delete a user profile.</td>
<td>Profile</td>
</tr>
<tr>
<td>ac_get_profile</td>
<td>Read the contents of a specified user profile.</td>
<td>Profile</td>
</tr>
<tr>
<td>ac_list_profiles</td>
<td>Get a list of all the user IDs for which there are profiles.</td>
<td>Profile</td>
</tr>
<tr>
<td>ac_load_profiles</td>
<td>Load one or more user profiles.</td>
<td>Profile</td>
</tr>
<tr>
<td>ac_query_profile</td>
<td>Verify that a specified profile exists on the coprocessor, and return its length.</td>
<td>Profile</td>
</tr>
<tr>
<td>ac_reset_logon_fail_cnt</td>
<td>Reset the logon failure count in a user profile.</td>
<td>Profile</td>
</tr>
<tr>
<td>update_logon_failure_cnt</td>
<td>Change the logon failure count for a user profile.</td>
<td>Profile</td>
</tr>
</tbody>
</table>
10.5.1  ac_chg_prof_auth_data - change profile authentication data

ac_chg_prof_auth_data replaces the authentication data in a user profile, for a specific authentication mechanism. The old data is replaced with the new data, but only if the Replacable flag is set in the attributes of the old authentication data.

Alternatively, if no data exists in the profile for the specified mechanism, the data is appended to the authentication data already present in that profile.

**Note:** this function updates the PROFILE.srd file. Since there is no method for a UDX to lock out a CCA verb which may be reading this SRDI while this function is writing it, its use should be avoided.

**Function prototype**

```c
uint32_t ac_chg_prof_auth_data ( optx_t *pOptx, profile_id_t profile_id, void *auth_data );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `profile_id` is the 8-byte, non-NULL terminated ID of the profile whose authentication data is to be changed.
- `auth_data` is a pointer to a buffer containing the new authentication data, in the following format:
  - a variable of type `profs_authen_mech_t` which contains:
    - `numbytes` - the number of bytes of data in the buffer.
    - `mech_id` - an identifier which describes the authentication mechanism - for passphrase authentication, the mechanism ID is X'01'.
    - `mech_strength` - an integer which defines the strength of this authentication method. X'00' is reserved for users who have not been authenticated.
    - `exp_date` - a `prof_date_t` variable containing the expiration date for this authentication data. The format is:
      - `year` is a 2-byte integer in big-endian order
      - `month` is a one-byte integer (1-12)
      - `date` is a one-byte integer (1-31)
    - `mech_attr` - a bit-flag to represent any attributes needed to describe the operation and use of this authentication mechanism. The only currently defined value is RENEWABLE, in the most significant bit. The presence of this flag indicates that an authentication method may be updated by the user (for example, a passphrase change).
  - concatenated to the `profs_authen_mech_t` variable, a variable length field containing the authentication data for this user and method. For passphrases, this field contains the 20-byte SHA-1 hash of the user's passphrase. This hash is computed in the host.

**Output**

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

The profile specified in `profile_id` has been updated in one of the following two ways:
If the profile previously had an authentication with the same mechanism ID, its authentication method has been changed.

If the profile previously had NO authentication with an identical mechanism ID, the authentication data has been ADDED to the profile.

The PROFILES SRDI has been saved to BBRAM.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acm_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>acm_INVALID_AUTHENT_DATA</td>
<td>The authentication data was invalid.</td>
</tr>
<tr>
<td>acm_PROFILE_NOT_FOUND</td>
<td>The specified profile was not in the profile list.</td>
</tr>
<tr>
<td>acm_NON_REPLACEABLE_DATA</td>
<td>The REPLACABLE flag was not set in the user's profile.</td>
</tr>
<tr>
<td>acm_MEM_ALLOC_ERROR</td>
<td>There was insufficient memory to store the new data.</td>
</tr>
</tbody>
</table>
10.5.2  ac_chg_prof_exp_date - change profile expiration date

`ac_chg_prof_exp_date` changes the expiration date in a specified user profile. This function is used to re-enable a user whose profile has expired, or to extend the lifetime of a profile that is about to expire.

**Function prototype**

```c
uint32_t ac_chg_prof_exp_date ( optx_t *pOptx, 
                                 profile_id_t profile_id, 
                                 prof_exp_date_t exp_date );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `profile_id` is the ID of the profile whose expiration date should be changed.
- `exp_date` is the new expiration date, in the following format:
  - `year` is a 2-byte integer in big-endian order.
  - `month` is a one-byte number representing the month (1-12).
  - `date` is a one-byte number representing the date (1-31).

**Output**

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

The expiration date of `profile_id` has been changed to `exp_date`, and the PROFILES SRDI has been saved to BBRAM.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acm_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>acmPROFILE_NOT_FOUND</td>
<td>The specified profile was not in the profile list.</td>
</tr>
</tbody>
</table>
10.5.3  ac_del_profile - delete user profile

`ac_del_profile` deletes a user profile from the coprocessor. The specified profile is deleted from the table of enrolled profiles. In addition, if the specified user is logged on, they are logged off.

**Function prototype**

```c
uint32_t ac_del_profile ( optx_t *pOptx,
                        profile_id_t profile_id );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `profile_id` is the 8-byte, non-NULL terminated ID of the profile to be deleted.

**Output**

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

The user specified by `profile_id` has been logged off the system (if needed) and the profile has been removed from the profiles list. In addition, the PROFILES SRDI has been saved to BBRAM.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acm_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>acm_PROFILE_NOT_FOUND</td>
<td>The specified profile was not in the profile list.</td>
</tr>
</tbody>
</table>
10.5.4 ac_get_profile - get profile

`ac_get_profile` returns the information from a specified user profile. The data begins with the fixed profile contents, defined by data type `user_profile_t`. The user's authentication data is concatenated to the end of this structure.

**Function prototype**

```c
uint32_t ac_get_profile ( profile_id_t profile_id, 
                         USHORT *profilesize, 
                         void *profile_data );
```

**Input**

On entry to this routine:

- `profile_id` is an 8-byte character string non-NULL terminated containing the name of the profile to be returned.
- `profile_size` is a pointer to a variable which contains the size of the `profile_data` buffer, in bytes.
- `profile_data` is a pointer to a buffer where the profile will be stored.

**Output**

On successful exit from this routine:

- `profile_size` contains the actual size of the returned profile data.
- `profile_data` contains the profile data.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>acm_NO_ERROR</code></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><code>acm_PROFILE_NOT_FOUND</code></td>
<td>The specified profile was not in the profile list.</td>
</tr>
<tr>
<td><code>acm_BFR_TOO_SMALL</code></td>
<td>The buffer was not large enough to load the profile data. The buffer size required has been returned in the <code>profile_size</code> variable.</td>
</tr>
</tbody>
</table>
10.5.5  **ac_list_profiles - list user profiles**

*ac_list_profiles* returns a list of all the user IDs for the profiles enrolled on the coprocessor. This is a series of unordered 8-byte user IDs.

**Function prototype**

```c
long ac_list_profiles ( profile_id_t profile_list [],
                       uint32_t *num_profiles );
```

**Input**

On entry to this routine:

- `profile_list` is a pointer to an array of 8-byte elements to hold the user IDs.
- `num_profiles` is a pointer to the maximum number of 8-byte values which the array can hold.

**Output**

On successful exit from this routine:

- `profile_list` contains the list of IDs (non-NULL terminated 8-byte strings).
- `num_profiles` contains the number of profile IDs which were returned.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acm_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>acm_BFR_TOO_SMALL</td>
<td>The array is not large enough to hold all the profile IDs.</td>
</tr>
</tbody>
</table>
10.5.6 ac_load_profiles - load user profiles

`ac_load_profiles` loads one or more user profiles into the coprocessor. Existing profiles can be replaced, if new ones have the same name.

The user profiles are an aggregate structure, where any number of profiles can be concatenated into a single message.

**Function prototype**

```c
uint32_t ac_load_profiles ( optx_t *pOptx,
                            void *profile_list,
                            boolean replace_profiles );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `profile_list` is a pointer to a variable length buffer of the following form:
  - a 4-byte variable containing the number of profiles in this list followed by
  - a 4-byte variable containing X'0000' followed by
  - the first profile followed by
  - any profiles in between followed by
  - the last profile

Each of the profiles in the profile list is a `user_profile_t` variable initialized to the following values:

- `profile_vers` - the profile structure version, currently X'01'.
- `profile_lth` - the length of this profile, in bytes.
- `comment` - a descriptive comment, of 19 bytes or less in length, NULL terminated.
- `cksum` - a checksum value to ensure the integrity of the data.
- `failure_cnt` - the number of logon failures, initialized to 0.
- `user_id` - the user ID associated with this profile. An 8-byte, non-NULL terminated string.
- `role_id` - an existing role from the ROLES SRDI. An 8-byte, non-NULL terminated string.
- `act_date` - the activation date of this user, a `prof_date_t` variable with the following values:
  - `year` - a 2-byte variable with the year (4 digits) in big-endian format
  - `month` - a one-byte character representing the month (1-12)
  - `day` - a one-byte character representing the date (1-31)

- `exp_date` - the expiration date of this user, a `prof_date_t` variable with the values as the `act_date` structure.

- authentication data is included at this point. The variable length nature of authentication data is the reason for the `profile_lth` field.

Access control manager functions  263
replace_profiles is a boolean variable with TRUE indicating that existing profiles should be changed, FALSE indicating that they should not.

Output
This function returns no output. On successful exit from this routine:
The PROFILES SRDI has been updated and saved to BBRAM.

Return codes
Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acm_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>acm_FLASH_SPACE_FULL</td>
<td>There is insufficient memory in the BBRAM to store the new profile list.</td>
</tr>
<tr>
<td>acm_MEM_ALLOC_ERROR</td>
<td>The function was unable to allocate sufficient memory to install the new profiles.</td>
</tr>
<tr>
<td>acm_PROFILE_EXISTS</td>
<td>The user_id is already in the PROFILES SRDI, and replace_profiles was FALSE. No profiles have been added.</td>
</tr>
</tbody>
</table>
10.5.7 ac_query_profile - return the length of a user profile

`ac_query_profile` returns the length of a specified user profile, or zero if the profile does not exist on the coprocessor.

**Function prototype**

```c
uint32_t ac_query_profile( profile_id_t profile_id, USHORT *profile_length)
```

**Input**
On entry to this routine:
- `profile_id` is an 8-byte, non-NULL terminated string representing the name of the profile being checked.
- `profile_length` is a pointer to variable that will receive the profile structure length, in bytes. The length will be zero if the profile does not exist.

**Output**
On successful exit from this routine:
- `profile_length` contains the length, in bytes. If the profile does not exist, a length of zero is returned.

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

<table>
<thead>
<tr>
<th>return code</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acm_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>acmPROFILE_NOT_FOUND</td>
<td>The specified profile was not in the profile list.</td>
</tr>
</tbody>
</table>
10.5.8  **ac_reset_logon_fail_cnt - reset logon failure count**  

*ac_reset_logon_fail_cnt* resets the logon failure count in the specified user profile. This is the count of the number of consecutive times the user has tried unsuccessfully to authenticate themselves to the coprocessor. The count is reset to zero.

**Function prototype**

```c
uint32_t ac_reset_logon_fail_cnt ( optx_t *pOptx,
                                    profile_id_t profile_id );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `profile_id` is an 8-byte, non-NULL terminated character string ID for the profile to be reset.

**Output**

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

The logon failure count of the profile specified by `profile_id` has been set to zero.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>acm_NO_ERROR</strong></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><strong>acm_PROFILE_NOT_FOUND</strong></td>
<td>The specified profile was not in the profile list.</td>
</tr>
</tbody>
</table>

266  IBM 4767 CCA UDX Reference
10.5.9  update_logon_failure_count - adjust count of logon failures

*update_logon_failure_count* adjusts the count of the number of times a user has attempted to log on with an invalid password.

If the logon failure count is greater than 2, the user is not be able to log on until an administrator with access control point 0x'0115', Reset User Profile Logon-Attempt-Failure Count, uses the *CSUAACI* verb to reset the user's profile.

**Function prototype**

```c
uint32_t update_logon_failure_count ( optx_t *pOptx,
                                  profile_id_t user_id,
                                  unsigned char new_failure_cnt );
```

**Input**

On entry to this routine:

*pOptx* is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

*user_id* is an 8-byte user ID.

*new_failure_count* is a 1-byte integer indicating the number to be stored as the new failure count.

**Output**

This function has no output. On return, the PROFILE SRDI has been updated.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acm_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>acm_PROFILE_NOT_FOUND</td>
<td>The specified profile was not in the profile list.</td>
</tr>
</tbody>
</table>
10.6 Functions to manage logged on users

The functions in this section are used to find, identify, and count the number of users logged on to the coprocessor, and to track and use the session keys which are used to secure communication with the host machine.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac_check_authorization</td>
<td>Check whether a user is authorized to execute a specified function.</td>
<td>User</td>
</tr>
<tr>
<td>ac.lu_add_user</td>
<td>Add a user to the table of logged-on users, and generate a session key for that user.</td>
<td>User</td>
</tr>
<tr>
<td>ac.lu_drop_user</td>
<td>Drop a user from the table of logged-on users.</td>
<td>User</td>
</tr>
<tr>
<td>ac.lu_get_ks</td>
<td>Get a copy of a specified user’s session key.</td>
<td>User</td>
</tr>
<tr>
<td>ac.lu_get_num_users</td>
<td>Find out how many users are currently logged on.</td>
<td>User</td>
</tr>
<tr>
<td>ac.lu_get_role</td>
<td>Get the role ID for a specified logged-on user.</td>
<td>User</td>
</tr>
<tr>
<td>ac.lu KS_dec</td>
<td>Decrypt data with a specified user’s session key.</td>
<td>User</td>
</tr>
<tr>
<td>ac.lu KS_hmacgen</td>
<td>Computes a HMAC on a message using the session key.</td>
<td>User</td>
</tr>
<tr>
<td>ac.lu KS_hmacver</td>
<td>Verifies an HMAC using the session key.</td>
<td>User</td>
</tr>
<tr>
<td>ac.lu_list_users</td>
<td>Get a list of all the logged-on users.</td>
<td>User</td>
</tr>
<tr>
<td>ac.lu_query_user</td>
<td>Check whether a specified user is currently logged on (authenticated).</td>
<td>User</td>
</tr>
</tbody>
</table>
10.6.1 ac_check_authorization - check authorization to execute function

ac_check_authorization determines if a user is permitted to execute a specified function, based on the user's role, access control state information, and other access control parameters.

**Function prototype**

```c
uint32_t ac_check_authorization ( optx_t *pOptx,
                                 USHORT requested_fcn,
                                 boolean *granted );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine. This contains the role under which the user is operating.

- `requested_fcn` is an index into the Access Control Points table, indicating which function the user wants to execute.

- `granted` is a pointer to a variable that will receive the response, indicating whether the user is allowed to execute the desired function.

**Output**

On successful exit from this routine:

- `ac_check_authorization` returns a boolean value of **TRUE** indicating the user is allowed to use the specified function in the CCA application, and **FALSE** if not.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>acm_NO_ERROR</th>
<th>The operation was successful.</th>
</tr>
</thead>
<tbody>
<tr>
<td>acm_ROLE_NOT_FOUND</td>
<td>The specified role was not in the role list.</td>
</tr>
</tbody>
</table>
10.6.2 ac_lu_add_user - add a user to the list of logged on users

`ac_lu_add_user` informs the Access Control Manager that a specified user has successfully authenticated, and should be added to the list of logged-on users.

The Access Control Manager adds a new entry to the table of logged on users, inserting the specified profile ID as the search key used to identify the entry. It finds the profile and looks up the role, and also puts that in the table entry. This saves the cost of finding the profile each time the user’s role is required for future access control operations. Finally, the Access Control Manager generates a session key for the user, and inserts that in the table with the ID and the role.

If the specified profile ID is already in the table of logged on users, the request is rejected.

**Function prototype**

```c
uint32_t ac_lu_add_user ( profile_id_t profile_id,
                         sess_key_alg_t key_type,
                         optx_t *pOptx );
```

**Input**

On entry to this routine:

- `profile_id` is an 8-byte, non-NULL terminated string representing the user to be logged on.
- `key_type` is one of `SESS_KEY_AES` or `SESS_KEY_DES`.
- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

**Output**

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

The user identified by `profile_id` has been added to the list of logged-on users.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acm_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>acm_ALREADY_LOGGED_ON</td>
<td>The user was already logged on.</td>
</tr>
<tr>
<td>acm_MEM_ALLOC_ERROR</td>
<td>The function was unable to allocate sufficient memory to add the role to the tables.</td>
</tr>
<tr>
<td>acm_PROFILE_NOT_FOUND</td>
<td>The profile_id could not be found in the Profiles SRDI.</td>
</tr>
<tr>
<td>acm_ROLE_NOT_FOUND</td>
<td>The role specified in the Profiles SRDI for the requested profile_id could not be found in the Roles SRDI.</td>
</tr>
</tbody>
</table>
10.6.3  ac.lu.drop_user - remove a user from the logged on list

ac.lu.drop_user informs the Access Control Manager that a user should be logged off, dropping that user from the logged-on users table.

Function prototype

long ac.lu.drop_user ( profile_id_t profile_id );

Input

On entry to this routine:

profile_id is an 8-byte, non-NULL terminated string containing the name of the profile to be removed.

Output

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

ac.lu.drop_user removes the user from the logged-on users table.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acm_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>acm_NOT_LOGGED_ON</td>
<td>The profile_id was not in the logged on users table.</td>
</tr>
</tbody>
</table>
10.6.4 ac_lu_get_ks - get a copy of a session key

ac_lu_get_ks gets a copy of the session key for a specified logged-on user. Session keys are used for Message Authentication codes in the CPRB request as well as for encrypting data to be sent securely across the PCI bus.

Function prototype

```c
long ac_lu_get_ks ( profile_id_t profile_id,
                    sess_key_alg_t *key_type,
                    user_session_key_t *session_key );
```

Input

On entry to this routine:

- `profile_id` is an 8-byte, non-NULL terminated string containing the name of the profile whose session key you want to retrieve.
- `key_type` is one of `SESS_KEY_AES` or `SESS_KEY_DES`.
- `session_key` is a pointer to a buffer which will receive the 8 byte session key for the specified user.

Output

On successful exit from this routine:

- `session_key` contains the cleartext session key KS for the specified user.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acm_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>acm_NOT_LOGGED_ON</td>
<td>The user is not logged on.</td>
</tr>
</tbody>
</table>
10.6.5  **ac_lu_get_num_users - get the number of logged on users**

*ac_lu_get_num_users* returns the number of users who are currently logged on to the coprocessor. This can be used to determine how much data will be returned by the *ac_lu_list_users* function.

**Function prototype**

```c
long ac_lu_get_num_users ( uint32_t *num_users );
```

**Input**

On entry to this routine:

*num_users* is a pointer to a variable which receives the number of logged-on users.

**Output**

On successful exit from this routine:

*num_users* contains the number of users who are currently logged on to the coprocessor.

**Return codes**

Common return codes generated by this routine are:

| acm_NO_ERROR                      | The operation was successful. |
10.6.6 ac_lu_get_role - get role from the logon list

ac_lu_get_role returns the role ID for a specified logged-on user.

**Function prototype**

```
long ac_lu_get_role ( profile_id_t profile_id,
                     role_id_t role,
                     boolean *found );
```

**Input**

On input to this routine:

- `profile_id` is an 8-byte, non-NULL terminated ID of the profile that you want to retrieve.
- `role` is a variable which receives the role ID for the specified user.
- `found` is a pointer to a variable in which a value of type boolean can be stored.

**Output**

On successful exit from this routine:

- If the buffer pointed at by `found` is TRUE, `role` contains the role ID for the specified user.
- If the buffer pointed at by `found` contains FALSE, the contents of `role` are unchanged.

**Return codes**

Common return codes generated by this routine are:

| acm_NO_ERROR | The operation was successful. |
10.6.7  ac_lu_ks_dec - decrypt data with session key

ac_lu_ks_dec decrypts a string of data with a specified user's session key. The data length must be a multiple of eight bytes. Decryption is performed using triple-DES CBC mode, with an initialization vector of X'0000000000000000'.

Function prototype

```
long ac_lu_ks_dec ( optx_t *pOptx,
                   profile_id_t profile_id,
                   UCHAR *ciphertext,
                   UCHAR *cleartext,
                   ULONG text_length );
```

Input

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `profile_id` is an 8-byte, non-NULL terminated string representing the name of the profile whose session key should be used to decrypt the data.
- `ciphertext` is a pointer to the buffer containing the data to be deciphered.
- `cleartext` is a pointer to a buffer where the deciphered data will be stored. This buffer must be at least as large as the `ciphertext` buffer.
- `text_length` is the length of the data to be deciphered, in bytes. This value must be a multiple of eight.

Output

On successful exit from this routine:

- `cleartext` contains the deciphered data.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acm_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>acm_NOT_LOGGED_ON</td>
<td>The user is not logged on.</td>
</tr>
</tbody>
</table>
10.6.8   ac_lu_ks_hmacgen - Compute an HMAC Using Session Key

ac_lu_ks_hmacgen computes the HMAC on a message, using the supplied session key.

Function prototype

    long ac_lu_ks_hmacgen ( optx_t *pOptx,
                           profile_id_t profile_id,
                           UCHAR *mesagetext,
                           ULONG msg_length,
                           mac_t mac );

Input

On entry to this routine:

pOptx is a pointer to the operation tracking structure that was input to the command processor which calls this routine.

profile_id must contain the ID of the profile of the logged-on user.

mesagetext is a pointer to a buffer containing the message.

msg_length is the number of bytes of data in mesagetext.

mac is an 8-byte buffer.

Output

On successful exit from this routine:

mac contains the 8-byte HMAC of the mesagetext, computed with the session key associated with profile_id.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acm_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>acm_NOT_LOGGED_ON</td>
<td>The user is not logged on.</td>
</tr>
</tbody>
</table>
10.6.9 **ac_lu_ks_hmacver** - verify an HMAC using session key

*ac_lu_ks_hmacver* verifies a supplied MAC on a supplied message, using the user's session key.

**Function prototype**

```c
long ac_lu_ks_hmacver ( optx_t *pOptx,
                        profile_id_t profile_id,
                        UCHAR *messagetext,
                        ULONG msg_length,
                        mac_t mac
                        USHORT mac_length,
                        boolean *verified );
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to the operation tracking structure that was input to the command processor which calls this routine.
- `profile_id` should contain the user's profile ID.
- `messagetext` is a pointer to the message that is to be verified.
- `msg_length` is the number of bytes in the message.
- `mac` is the 8-byte HMAC which is to be verified.
- `verified` is a pointer to a boolean variable, in which the answer will be supplied.

**Output**

On successful exit from this routine:

- `verified` contains **TRUE** if the MAC was verified, and **FALSE** if it was not.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM_NO_ERROR</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>ACM_NOT_LOGGED_ON</td>
<td>The user is not logged on.</td>
</tr>
</tbody>
</table>
10.6.10  **ac_lu_list_users - list the IDs of the logged on users**

`ac_lu_list_users` returns a list of the user IDs for each logged-on user.

**Function prototype**

```c
long ac_lu_list_users ( profile_id_t user_list [], uint32_t *num_users);
```

**Input**

On entry to this routine:

- `user_list` is an array which receives the list of logged-on users.
- `num_users` is a pointer to a variable containing the number of eight-byte elements in `user_list`.

**Output**

On successful exit from this routine:

- `user_list` contains the (unordered) list of users.
- `num_users` contains the number of elements returned in the `user_list` array.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>acm_NO_ERROR</code></td>
<td>The operation was successful.</td>
</tr>
<tr>
<td><code>acm_BFR_TOO_SMALL</code></td>
<td>The array is not large enough to hold all the profile IDs.</td>
</tr>
</tbody>
</table>
10.6.11 ac_lu_query_user - check if a user is logged on

`ac_lu_query_user` determines whether a specified user is currently logged on to the coprocessor. This is an indication that the user has successfully authenticated to the coprocessor according to one of the enrolled user profiles.

**Function prototype**

```c
long ac_lu_query_user ( profile_id_t profile_id, 
                        boolean *logged_on);
```

**Input**

On entry to this routine:

- `profile_id` is an 8-byte, non-NULL terminated string representing the name of the user being checked.
- `logged_on` is a pointer to a boolean variable which receives the response to your query.

**Output**

On successful exit from this routine:

- `logged_on` contains a boolean value of `TRUE` if the user is logged on, and `FALSE` if not.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>acm_NO_ERROR</th>
<th>The operation was successful.</th>
</tr>
</thead>
</table>

Access control manager functions  279
11 Cache management functions

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

**Note:** All functions within this chapter are available on the workstation host and the coprocessor.

### 11.1 Header files for caching functions

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

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

### 11.2 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.)

### 11.3 Summary of functions

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache_clear</td>
<td>Remove all entries from a cache.</td>
</tr>
<tr>
<td>cache_delete</td>
<td>Remove an existing cache.</td>
</tr>
<tr>
<td>cache_delete_item</td>
<td>Remove a specific item from the cache.</td>
</tr>
<tr>
<td>cache_get_item</td>
<td>Retrieve an item from the cache, copying it into a user-supplied buffer.</td>
</tr>
<tr>
<td>cache_get_item_b</td>
<td>Retrieve an item from the cache after allocating memory to hold it.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>cache_init</td>
<td>Open a new cache.</td>
</tr>
<tr>
<td>cache_status</td>
<td>Check the status of an existing cache.</td>
</tr>
<tr>
<td>cache_write_item</td>
<td>Store an item in the cache.</td>
</tr>
</tbody>
</table>
11.3.1   cache_clear - remove all data from cache

`cache_clear` clears all data from the specified cache.

**Function prototype**

```c
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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ce_OK</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>ce_BAD_HANDLE</td>
<td>The specified cache does not exist.</td>
</tr>
</tbody>
</table>
11.3.2 cache_delete - delete cache

cache_delete will remove an existing cache, recovering all memory.

Function prototype

    cacheError cache_delete ( cacheHandle handle );

Input

On entry to this routine:

    handle must be set to a valid cacheHandle.

Output

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

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ce_OK</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>ce_BAD_HANDLE</td>
<td>The specified cache does not exist.</td>
</tr>
</tbody>
</table>
11.3.3 cache_delete_item - delete one item from cache

*cache_delete_item* deletes the specified item from the cache.

**Function prototype**

```c
cacheError cache_delete_item ( cacheHandle handle,
                              short_tag_t short_tag,
                              uint32_t full_tag_length,
                              uint8_t *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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ce_OK</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>cd_ITEM_NOT_FOUND</td>
<td>The specified item was not found in the cache, or the specified item in the cache failed an LRC check (if the LRC check was on when the cache was initialized).</td>
</tr>
<tr>
<td>ce_BAD_HANDLE</td>
<td>The specified cache does not exist.</td>
</tr>
</tbody>
</table>
11.3.4 cache_get_item - get an item from cache

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

Function prototype

```c
cacheError cache_get_item ( cacheHandle handle,
uint32_t *bfrSize,
uint8_t *bfr,
short_tag_t short_tag,
uint32_t full_tag_length,
uint8_t *full_tag );
```

Input

On entry to this routine:

*handle* contains the handle for the specified cache.

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

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

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

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

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

Output

On successful exit to this routine:

*bfrSize* is a pointer to the length of the item which was retrieved from the cache.

*bfr* contains the item which was retrieved.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ce_OK</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>cd_ITEM_NOT_FOUND</td>
<td>The specified item was not found in the cache, or the specified item in the cache failed an LRC check (if the LRC check was on when the cache was initialized).</td>
</tr>
<tr>
<td>ce_BAD_HANDLE</td>
<td>The specified cache does not exist.</td>
</tr>
<tr>
<td>ce_BFR_TOO_SMALL</td>
<td>The provided buffer was not large enough to hold the item.</td>
</tr>
</tbody>
</table>
11.3.5  cache_get_item_b - get an item in a new buffer

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

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

**Function prototype**

```c
cacheError cache_get_item_b ( cacheHandle handle,
    uint32_t *bfrSize,
    uint8_t **bfr,
    short_tag_t short_tag,
    uint32_t full_tag_length,
    uint8_t *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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ce_OK</strong></td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td><strong>cd_ITEM_NOT_FOUND</strong></td>
<td>The specified item was not found in the cache, or the specified item in the cache failed an LRC check (if the LRC check was on when the cache was initialized).</td>
</tr>
<tr>
<td><strong>ce_BAD_HANDLE</strong></td>
<td>The specified cache does not exist.</td>
</tr>
<tr>
<td><strong>ce_MEM_ALLOC_ERROR</strong></td>
<td>The function was unable to allocate memory for the buffer to return the item.</td>
</tr>
</tbody>
</table>
11.3.6 cache_init - initialize a cache

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

Function prototype

```c
cacheError cache_init ( uint32_t maxBytes,
                        uint32_t maxItemBytes,
                        uint32_t options,
                        cacheHandle *handle );
```

Input

On entry to this routine:

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

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

- `options` must be set to one of the following values:
  - 0x00000001 to enable an LRC check on the data
  - 0x00000000 to disable the LRC check

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

Output

On successful exit from this routine:

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

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ce_OK</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>ce_NULL_POINTER</td>
<td>One of the parameters was a null pointer.</td>
</tr>
<tr>
<td>ce_ITEM_TOO_LARGE</td>
<td>The <code>maxItemBytes</code> parameter is too large for the <code>maxBytes</code>.</td>
</tr>
<tr>
<td>ce_MEM_ALLOC_ERROR</td>
<td>The function was unable to allocate memory for the cache overhead.</td>
</tr>
</tbody>
</table>

Note: If `options` is 0x00000001 on entry to this routine, the LRC check will be enabled for each access to each item stored in the cache. If the LRC check fails, `ce_ITEM_NOT_FOUND` will be returned on any attempt to access an item.
11.3.7 cache_status - determine cache status

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

**Function prototype**

```c
cacheError cache_status( cacheHandle handle, 
    uint32_t *maxBytes,          
    uint32_t *maxItemBytes,      
    uint32_t *bytesUsed,         
    uint32_t *itemsStored );
```

**Input**

On entry to this routine:

- `handle` is the handle of the cache to be queried.
- `maxBytes` is a pointer to a 4-byte data item.
- `maxItemBytes` is a pointer to a 4-byte data item.
- `bytesUsed` is a pointer to a 4-byte data item.
- `itemsStored` is a pointer to a 4-byte 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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ce_OK</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>ce_NULL_POINTER</td>
<td>One of the parameters was a null pointer.</td>
</tr>
<tr>
<td>ce_BAD_HANDLE</td>
<td>The specified cache does not exist.</td>
</tr>
</tbody>
</table>
11.3.8  cache_write_item - add an item to cache

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

Function prototype

```c
cacheError cache_write_item ( cacheHandle handle,
    uint32_t item_length,
    uint8_t *item,
    short_tag_t short_tag,
    uint32_t full_tag_length,
    uint8_t *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:

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ce_OK</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>ce_NULL_POINTER</td>
<td>One of the parameters was a null pointer.</td>
</tr>
<tr>
<td>ce_BAD_HANDLE</td>
<td>The specified cache does not exist.</td>
</tr>
<tr>
<td>ce_ITEM_TOO_LARGE</td>
<td>The sum of the <code>full_tag_length</code> and the <code>item_length</code> is greater than the maximum item size for the cache.</td>
</tr>
</tbody>
</table>
12 Miscellaneous functions
This chapter describes functions that do not fit into any of the previously described categories.

12.1 Header files for miscellaneous functions
When using these functions, your program must include the following header files.

```c
#include "cassub.h"    /* Common subroutines */
#include "camacm.h"     /* Access control manager */
#include "cmnfunct.h"   /* Functions for host and coprocessor */
#include "cmnlibr.h"    /* Host library functions */
#include "casfunct.h"   /* coprocessor subroutines */
```

12.2 Summary of functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CasGetFileDescriptor</td>
<td>Returns the file descriptor which has been assigned by CCA to the hardware device required for a 4765 function (deprecated).</td>
</tr>
<tr>
<td>CHECK_ACCESS_AUTH</td>
<td>Verifies the user's authority. See check_access_auth_fcn - verify user authority on page 293.</td>
</tr>
<tr>
<td>check_access_auth_fcn</td>
<td>Verifies the user's authority.</td>
</tr>
<tr>
<td>GetAnyKeyLength</td>
<td>Returns the length of a specified key token.</td>
</tr>
<tr>
<td>GetKeyLength</td>
<td>Returns the length of a specified key token. See GetAnyKeyLength - get length of key token on page 294.</td>
</tr>
<tr>
<td>GetVarKeyLength</td>
<td>Returns the length of a specified key token. See GetAnyKeyLength - get length of key token on page 294.</td>
</tr>
<tr>
<td>GetRandomNumber</td>
<td>Returns a random number. See GetRandomNumber – generate random number on page 295.</td>
</tr>
<tr>
<td>intel_long_reverse</td>
<td>Byte-reverses a 4-byte block of data.</td>
</tr>
<tr>
<td>intel_word_reverse</td>
<td>Byte-reverses a 2-byte block of data.</td>
</tr>
<tr>
<td>LIBR_AllocStorage</td>
<td>Allocates memory.</td>
</tr>
<tr>
<td>LIBR_FreeStorage</td>
<td>Frees unneeded memory.</td>
</tr>
<tr>
<td>LIBR_ReleaseSemaphore</td>
<td>Releases a semaphore, to allow others to access the item.</td>
</tr>
<tr>
<td>LIBR_RequestSemaphore</td>
<td>Waits for a semaphore to be free.</td>
</tr>
<tr>
<td>TOKEN_IS_A_LABEL</td>
<td>Determines whether a key identifier is a token or a label.</td>
</tr>
<tr>
<td>TOKEN_LABEL_CHECK</td>
<td>Determines whether a key identifier is a token or a label.</td>
</tr>
<tr>
<td>XcSetClock</td>
<td>Set the coprocessor clock.</td>
</tr>
</tbody>
</table>
### 12.2.1 CasGetFileDescriptor - Deprecated

This function is available only on the coprocessor.

*CasGetFileDescriptor* returns the file descriptor which has been assigned by CCA to the hardware device required for a 4765 function. The 4767 adapter does not use file descriptors, so negative numbers will be returned for all functions.

**Note:** This function is no longer needed and should be removed from your code.

#### Function prototype

```c
int CasGetFileDescriptor ( int hardware_type );
```

#### Input

On entry to this routine:

- `hardware_type` is a constant indicating which file descriptor to return. The following constants are defined:
  - `FD_XCRYPTO` is for accessing the low-battery and intrusion latch reset.
  - `FD_PKA` is for RSA, DSA, and modular math functions.
  - `FD_DES` is for DES and TES functions.
  - `FD_AES` is for AES functions.
  - `FD_SHA` is for SHA-1 functions.
  - `FD_HWRNG` is for the Hardware random number generator.
  - `FD_OA` is for the Outbound Authentication Manager.
  - `FD_RTC` is for the Real-Time Clock.

#### Output

On successful exit from this routine:

*CasGetFileDescriptor* returns **NO_FILE_DESCRIPTOR**

#### Return codes

This function has no return codes.
12.2.2 check_access_auth_fcn - verify user authority

This function is available only on the coprocessor.

`check_access_auth_fcn` examines the role of the user from the table of logged on users and determines whether that role is authorized to perform the function indicated by the provided Access Control Point. Each role contains a list of permitted access control points. The function returns a boolean value to indicate whether the permission was granted.

`CHECK_ACCESS_AUTH` is a macro which does the same thing as `check_access_auth_fcn`.

**Function prototype**

```
ULONG check_access_auth_fcn ( optx_t *pOptx,  
USHORT requested_fcn_code,  
boolean *pGranted );
```

```
#define CHECK_ACCESS_AUTH( V, W, X) check_access_auth_fcn( V, W, X)
```

**Input**

On entry to this routine:

- `pOptx` is a pointer to an Operation Tracking structure. This structure is created by the CCA manager and passed to the command processor with the role ID of the caller as well as pointers to the request and reply CPRB structures.

- `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.

- `pGranted` is a pointer to a buffer where a boolean result may be returned.

**Output**

On successful exit from this routine:

The value stored in `pGranted` is `TRUE` if the user has authorization, and `FALSE` if not.

**Notes**

UDXs built with the `libcsullib.so` file may have access granted to role IDs using the CNM utility, if new access control points are added to the file `csuap.def`.

**Return codes**

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>acm_ROLE_NOT_FOUND</td>
<td>The role is not in the SRDI.</td>
</tr>
</tbody>
</table>
12.2.3 GetAnyKeyLength - get length of key token
This function is available only on the workstation host.

GetAnyKeyLength returns the length of a specified DES, PKA, AES (fixed or variable length) or HMAC key token, or key label. This is the preferred function.

Function prototype

```c
USHORT GetAnyKeyLength ( UCHAR *keyid_ptr,
                          CCAINT *key_parm_length_ptr,
                          CCAINT *message_ptr );

USHORT GetVarKeyLength ( UCHAR *keyid_ptr,
                          CCAINT *key_parm_length_ptr,
                          CCAINT *message_ptr );

USHORT GetKeyLength ( UCHAR *keyid_ptr,
                       CCAINT *key_parm_length_ptr,
                       CCAINT *message_ptr );
```

Input
On entry to this routine:

keyid_ptr is a pointer to the start of the input key identifier data.

key_parm_length_ptr is a pointer to the expected key length, if this is an RSA key token, or NULL if this is a DES or fixed-length AES token. DES tokens in many functions are passed to the host without a parameter length, because they are fixed size. This function returns an error if the supplied token is larger than this expected size.

message_ptr is a pointer to an address that will contain the return code.

Output
On successful exit to this routine:

GetAnyKeyLength, GetVarKeyLength, and GetKeyLength each returns the length of the token, or -1 if an error occurred. If the token is a null token (that is, the first byte is 0x00), the function returns the value 1.

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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_OK</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>E_KEY_LEN</td>
<td>The key has a length less than 1 byte.</td>
</tr>
<tr>
<td>E_SIZE</td>
<td>The key is longer than the expected length in key_parm_length_ptr.</td>
</tr>
<tr>
<td>E_KEY_TOKEN</td>
<td>keyid_ptr was not pointing to a valid key token.</td>
</tr>
</tbody>
</table>
12.2.4 GetRandomNumber – generate random number

This function is available on the coprocessor.

GetRandomNumber is a wrapper for xcDRBGgenerate, which is described in the IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference. This manual is available on the IBM CryptoCards website.

Function prototype

```c
uint32_t GetRandomNumber ( uint8_t  *pRandom,
                           uint32_t  options,
                           uint32_t  length,
                           optx_t   *pOptx );
```

Input

On entry to this routine:

- `pRandom` is a pointer to the variable to receive the random number.
- `options` is the set of random number options required by xcDRBGgenerate. These options are defined in `xc_types.h`, which is in the Toolkit inc directory and are enumerated in the IBM 4767 Custom Software Interface Reference manual.
- `length` is the length of the random number requested, in bytes. The maximum length is 32K bytes.
- `pOptx` is a pointer to the operation tracking structure input to this command processor.

Output

On successful return from this routine:

The variable pointed to by `pRandom` contains the random number generated by the coprocessor.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>ERROR</td>
<td>The call to xcDRBGgenerate failed.</td>
</tr>
</tbody>
</table>
12.2.5        intel_long_reverse - convert 4-byte values

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

*intel_long_reverse* reverses the order of the bytes in a 32-bit (4-byte) integer. This is used to convert 32-bit values between big-endian and little-endian formats.

**Macro format**

```c
uint32_t intel_long_reverse ( uint32_t long_val )
```

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

```c
#ifdef HOST_BIG_ENDIAN
#define xtohl(d) intel_long_reverse((uint32_t)d)
#define htoxl(d) intel_long_reverse((uint32_t)d)
#define atohl(d) intel_long_reverse((uint32_t)d)
#define htoal(d) intel_long_reverse((uint32_t)d)
#else
#define xtohl(d) ((uint32_t)d)
#define htoxl(d) ((uint32_t)d)
#define atohl(d) ((uint32_t)d)
#define htoal(d) ((uint32_t)d)
#endif

#ifdef ADAPTER_BIG_ENDIAN
#define xtoal(d) intel_long_reverse((uint32_t)d)
#define atoxl(d) intel_long_reverse((uint32_t)d)
#define ltoal(d) intel_long_reverse((uint32_t)d)
#define stoal(d) intel_long_reverse((uint32_t)d)
#else
#define xtoal(d) intel_long_reverse((uint32_t)d)
#define atoxl(d) intel_long_reverse((uint32_t)d)
#define ltoal(d) intel_long_reverse((uint32_t)d)
#define stoal(d) ((uint32_t)d)
#endif
```

**Input**

On entry to this macro:

*long_val* is the input value. If the local platform is little-endian, it is reversed in byte order and is returned as the function result. If the local platform is big-endian, the input value is returned unchanged.

**Output**

When this macro completes:

*intel_long_reverse* returns the bytes from *long_val* in big-endian order.

**Return codes**

This function has no return codes.
12.2.6  intel_word_reverse - convert 2-Byte values

This function is available on both the workstation 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.

Macro format

#define intel_word_reverse( USHORT intel_int )

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

#ifndef HOST_BIG_ENDIAN
#define xtohs(d) intel_word_reverse((USHORT)d)
#define htoxs(d) intel_word_reverse((USHORT)d)
#define atohs(d) intel_word_reverse((USHORT)d)
#define htoas(d) intel_word_reverse((USHORT)d)
#else
#define xtohs(d) ((USHORT)d)
#define htoxs(d) ((USHORT)d)
#define atohs(d) ((USHORT)d)
#define htoas(d) ((USHORT)d)
#endif
#ifndef ADAPTER_BIG_ENDIAN
#define xtoas(d) intel_word_reverse((USHORT)d)
#define atoxs(d) intel_word_reverse((USHORT)d)
#define ltoas(d) intel_word_reverse((USHORT)d)
#define stoas(d) intel_word_reverse((USHORT)d)
#else
#define xtoas(d) intel_word_reverse((USHORT)d)
#define atoxs(d) intel_word_reverse((USHORT)d)
#define ltoas(d) intel_word_reverse((USHORT)d)
#define stoas(d) ((USHORT)d)
#endif

Input

On entry to this macro:

intel_int is the input word. If the local platform is little-endian, it is reversed in byte order and is returned as the function result. If the local platform is big-endian, the value is returned unchanged.

Output

When this macro completes:

intel_word_reverse returns the bytes from intel_int in big-endian order.

Return codes

This function has no return codes.
12.2.7 LIBR_AllocStorage - allocate memory
This function is available on both the workstation host and the coprocessor.

*LIBR_AllocStorage* allocates memory from the heap, returning a pointer to the memory.

**Function prototype**

```c
void * LIBR_AllocStorage ( size_t size );
```

**Input**

On entry to this routine:

size is the amount of memory to be allocated, in bytes.

**Output**

On successful return from this routine:

*LIBR_AllocStorage* returns a pointer to the allocated memory, or *NULL* if no memory was allocated.

**Return codes**

This function has no return codes.
12.2.8 LIBR_FreeStorage - free memory

LIBR_FreeStorage de-allocates memory which was allocated by LIBR_AllocStorage.
This function is available on both the workstation host and the coprocessor.

Function prototype

void LIBR_FreeStorage ( void *Address );

Input

On entry to this routine:

Address is the address of the memory to be freed.

Output

This function has no output. On successful return from this function, the memory at Address has been returned to the heap.

Return codes

This function has no return codes.
12.2.9  LIBR_ReleaseSemaphore
This function is available on the workstation host and the coprocessor.

*LIBR_ReleaseSemaphore* releases a semaphore, so that other processes may use the resource it protects.

**Function prototype**

```c
void LIBR_ReleaseSemaphore ( uint32_t SemHandle );
```

**Input**

On entry to this routine:

*SemHandle* is the semaphore handle to be released.

**Output**

This function has no output. On successful exit, the semaphore has been released.

**Return codes**

This function has no return codes.
12.2.10 LIBR_RequestSemaphore

`LIBR_RequestSemaphore` waits until a semaphore is available, to ensure that no other processes may use the resource it protects.

This function is available on the workstation host and the coprocessor.

**Function prototype**

```c
void LIBR_RequestSemaphore ( uint32_t SemHandle );
```

**Input**

`SemHandle` is the semaphore handle which is requested.

**Output**

This function has no output. On successful exit, the semaphore has been obtained.

**Return codes**

This function has no return codes.
12.2.11 SWAP64 - byte-reverse a 64-byte integer

This macro is available on the host and the coprocessor.

SWAP64 reverses the order of the bytes in 64-bits (8 bytes) of data. This is used to convert 8-byte values between big-endian and little-endian formats.

Macro format

```c
#define SWAP64(in)  
   (((0xFF & (in)) << 56) | ((in) >> 56) | 
   ((0xFFF0 & (in)) << 40) | ((0xFFF000000000000 & (in)) >> 40) | 
   ((0xFFF0000 & (in)) << 24) | ((0xFFF000000000000 & (in)) >> 24) | 
   ((0xFFF000000 & (in)) << 8) | ((0xFFF000000000000 & (in)) >> 8))
```

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

```c
#ifndef HOST_BIG_ENDIAN
    #define atohull(d) SWAP64((uint64_t)d)  
    #define htoaull(d) SWAP64((uint64_t)d)
#else
    #define atohull(d) ((uint64_t)d)  
    #define htoaull(d) ((uint64_t)d)
#endif
```

Input
On entry to this macro:

in is a 64-bit buffer (8 bytes).

Output
When this macro completes:

SWAP64 returns the 8 bytes of in in reverse order.

Return codes
This function has no return codes.
12.2.12  **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.

**Macro format**

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

**Input**

On entry to this macro:

*keyid* is a pointer to a CCA key identifier.

**Output**

When this macro completes:

**TOKEN_IS_A_LABEL** returns **TRUE** if the first byte of the key identifier is between 0x20 and 0xFE, otherwise **FALSE**.

**Return codes**

This macro has no return codes.
12.2.13  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.

Macro format

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

Input

On entry to this macro:

`keyid` is the first byte of a CCA key identifier.

Output

When this macro completes:

`TOKEN_LABEL_CHECK` returns TRUE if `keyid` is between 0x20 and 0xFE, otherwise FALSE.

Return codes

This macro has no return codes.
12.2.14  xcSetClock – Set the coprocessor clock

This function sets the clock on the coprocessor to the specified time and date.

NOTE: despite the name, this function is not defined in the xcrypto library, but in CCA itself.

Function prototype

```c
long xcSetClock ( unsigned long day,
                 unsigned long month,
                 unsigned long year,
                 unsigned long hour,
                 unsigned long minute,
                 unsigned long second );
```

Input

On entry to this routine

- `day` is the date (day of the month).
- `month` is the one-based month (for example, January is 0x00000001).
- `year` is the year, a value between 1902 and 2037, inclusive.
- `hour` is the zero-based hour of the day (from 0 to 23).
- `minute` is the minute, from 00 to 59.
- `second` is the second from 00 to 59.

Output

On successful exit from this routine:

The coprocessor clock has been set.

Return codes

Common return codes generated by this routine are:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_OK</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>E_DATE_TIME</td>
<td>One of the parameters is invalid.</td>
</tr>
<tr>
<td>SECURE_CLOCK_FATAL_ERROR</td>
<td>An error was encountered setting the clock.</td>
</tr>
</tbody>
</table>

13 UDX sample code

Sample code for a command processor for a UDX is contained in the Toolkit. It consists of a coprocessor-side piece, a host-side piece, and a test program that demonstrates example round-trip calls to both the host-side and card-side UDX pieces.

The UDX sample is located in the y4tk/<version>/samples/udx/wks directory. This directory contains three directories: card, host, and test.

Three verbs are provided with this sample UDX:

1. a function to read and write SRDI files and check the status of the directory (UDXSRDI),
2. a function to read the memory availability of the coprocessor (UDXMEM), and
3. a function to decrypt and parse CCA key tokens (UDXKPAR).

UDXSRDI uses rules from the user to determine which of four internal requests to satisfy: to read an existing SRDI, to write a new or existing SRDI, to write a file and return statistics about the bbram directory, or to clear the statistics files from the directory.

1. The READDATA rule causes the UDXSRDI routine to read a SRDI and takes the name of the SRDI as input. Fiber mutexes are used to ensure that only one call is accessing the SRDI at a time.
2. The WRITESRD rule causes the UDXSRDI routine to write a SRDI and takes the name of the SRDI as input. Fiber mutexes are used to ensure that only one call is accessing the SRDI at a time.
3. The GETSIZE rule causes the UDXSRDI routine to write a file and return statistics checks for existing names and writes a new one (myfile00 up to myfile79). When this limit is reached an error is returned.
4. The CLEAR rule causes the UDXSRDI routine to clear the directory and erases all files named 'myfilexx' where xx is any two digits from 00 to 79. This will allow the GETSIZE rule (to write a file and return statistics) to complete without error. The SRDI files (named UDX0000 to UDX9999 will not be affected by this call.

UDXMEM returns information about the memory status of the coprocessor. A rule may be used to return this data encrypted with a DES key.

UDXKPAR decrypts various types of CCA key tokens and returns the key parts in the clear in a new structure. This sample is provided to demonstrate how to use the parser and parsed tokens. Rules are used to determine how many keys to parse (the keys are parsed in specific order, so setting the rule array to FIVE will skip HMAC and ECC keys, while setting it to SIX will skip only ECC keys.)

The testcase provided with the sample calls a mixture of UDXSRDI and UDXMEM if started with no command-line parameters or with the command-line parameters "1 -s". It calls UDXSRDI with the READDATA, WRITESRD, and GETSIZE rules and the UDXMEM verb with the CLEAR rule and with the ENCRYPTED rule. If the test function is called this way more than 8 times all calls with the GETDATA rule will return errors.

The testcase provided with the sample calls UDXSRDI with the CLEAR rule if started with the command line parameters "1 -c".

306 IBM 4765 CCA UDX Reference
The testcase provided with the sample calls UDXKEYS with the rule SEVEN.
14 Reserved values

Certain values have been reserved for the use of UDX developers. IBM will not use these values in future upgrades to the CCA, so there will be no overlap between a UDX using these values and an upgrade of the Toolkit. The specific values are as follows:

- For completion codes: The values between 0x5000 and 0x5FFF have been reserved for UDX writers.
- For access control points: For zSeries, the values between 0x8000 and 0xEFFF may be used. (Values between 0xF000 and 0xFFFF will be used on zSeries by IBM UDX writers.)
- For subfunction codes: The values between “XA” (0x5841) to “XZ” (0x585A), “YA” (0x5941) to “YZ” (0x595A), “X0” (0x5830) to “X9” (0x5839), and “Y0” (0x5930) to “Y9” (0x5939) are reserved for UDX writers.
- For DES control vectors: Bits 4, 5, and 61 will not affect or be affected by the import or export of a key. Bits 4 and 5 will be ignored by the CCA at all times. Bit 61 will prevent a key from use in any standard CCA verb, thus reserving a key for use only in a UDX function.
- For VAR tokens (AES or HMAC tokens in variable-length format): KUF Field 1, low order byte, is reserved for UDX tokens. If the high-order bit is set in the low-order byte of KUF field 1, the key may not be used as an operator key in any CCA function. This will not affect its use in CCA function KPI2, KGN2, or other key creation, import or export, or generation functions. Other bits in the low-order byte of KUF field one are ignored by CCA, and may be used by UDX writers to further restrict a UDX key.
15 Data structures
This appendix identifies useful data structures from the Toolkit header files.

15.1 Structures used in communications between the workstation host and the coprocessor
These structures may be used on the coprocessor or on the workstation host machine.

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 61440 bytes (MAX_XCRB_XFER).

<table>
<thead>
<tr>
<th>REQUEST_REPLY_BUF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size</strong></td>
</tr>
<tr>
<td>61440 bytes</td>
</tr>
<tr>
<td>61440 bytes</td>
</tr>
</tbody>
</table>

The REQUEST_REPLY_BUF structure is filled with the following structures (hence they are declared as pointers into the REQUEST_REPLY_BUF structure).

First, a CPRB structure:

Note that the (request) CPRB structure is filled (as completely as it needs to be) by calling CSUC_BULDCPRB with the appropriate lengths and pointers from within the host function. Fields not filled by this function will be filled by the Security Server when the coprocessor is called. Within the coprocessor code, the output CPRB fields are filled by the CCA dispatcher. Changing the values of these fields is not recommended, except for the replied_msg_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 typedef'ed as a pointer to the CPRB_structure.

Because CCA operates on multiple host platforms with different addressing sizes, the following macro has been defined to handle addresses in CCA data structures:

```c
#if defined(HOST32BIT)
  #define HOSTADDR(VARNAME,PADNAME) unsigned char PADNAME[12]; unsigned char *VARNAME
  #define HOSTADDR_RBFPTR(VARNAME,PADNAME) unsigned char PADNAME[12]; RBFPTR VARNAME
#else defined(HOST64BIT)
  #define HOSTADDR(VARNAME,PADNAME) unsigned char PADNAME[8]; unsigned char *VARNAME
  #define HOSTADDR_RBFPTR(VARNAME,PADNAME) unsigned char PADNAME[8]; RBFPTR VARNAME
#else defined(HOST128BIT)
  #define HOSTADDR(VARNAME,PADNAME) unsigned char PADNAME[24]; unsigned char *VARNAME
  #define HOSTADDR_RBFPTR(VARNAME,PADNAME) unsigned char PADNAME[24]; RBFPTR VARNAME
#endif
```
```c
#define HOSTADDR(VARNAME, PADNAME) unsigned char *VARNAME
#define HOSTADDR_RBFPTR(VARNAME, PADNAME) RBFPTR VARNAME
```

### CPRB Structure

<table>
<thead>
<tr>
<th>Size in bytes</th>
<th>Field Name</th>
<th>Description</th>
<th>Auto set</th>
<th>Modified by CCA dispatcher</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cprb_version_id</td>
<td>0, the version</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>cprb_rsvd_01</td>
<td>0 (reserved field)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>cprb_internal_flags[]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>function_id</td>
<td>“T2” to go to coprocessor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>cprb_flags</td>
<td>Indicators for request and reply</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>req_parm_block_length</td>
<td>Request parameter length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>req_data_block_length</td>
<td>Request data block length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>reply_msg_block_length</td>
<td>Provided reply parameter length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>replied_parm_block_length</td>
<td>Actual reply parameter length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>reply_data_block_length</td>
<td>Provided reply data block length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>replied_data_block_length</td>
<td>Actual reply data block length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>request_extension_block_length</td>
<td>Length of request extension block</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>cprb_rsvd_01b</td>
<td>0 (reserved field)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>replied_extension_block_length</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>cprb_rsvd_02</td>
<td>(pad)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>req_parm_block</td>
<td>Address of request parm block</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>cprb_rsvd_03</td>
<td>(pad)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>req_data_block_address</td>
<td>Address of request data block</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>cprb_rsvd_04</td>
<td>(pad)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>reply_parm_block</td>
<td>Address of reply parameter block</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>cprb_rsvd_05</td>
<td>(pad)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>reply_data_block</td>
<td>Address of reply data block</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

310 IBM 4767 CCA UDX Reference
<table>
<thead>
<tr>
<th>Size in bytes</th>
<th>Field Name</th>
<th>Description</th>
<th>Auto set</th>
<th>Modified by CCA dispatcher</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>cprb_rsvd_06</td>
<td>(pad)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>request_extension_block_addr</td>
<td>Address of request extension block block</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>cprb_rsvd_07</td>
<td>(pad)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>reply_extension_block_addr</td>
<td>Address of reply extension block block</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>secy_return_code</td>
<td>Return/Reason code</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>MAC_data_length</td>
<td>Length of MACed data area</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>logon_identifier</td>
<td>Logon User ID</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>MAC_value</td>
<td>Message Authentication Code on CPRB and request parm block, not including this field or the addresses in the structure.</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>MAC_content_flags</td>
<td>Indicate whether user is logged on or (special) if card has been re-initialized, thus user is (automatically) logged off.</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>cprb_rsvd_08</td>
<td>0 (reserved field)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Domain</td>
<td>MK_SET_DEFAULT</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>UsageDomainMask</td>
<td>0 (used for S390)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ControlDomainMask</td>
<td>0 (used for S390)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S390EnforcementMask</td>
<td>0 (used for S390)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>cprb_rsvd_09</td>
<td>0 (reserved)</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

On the host workstation, you will only need one CPRB_ptr, since the request CPRB you build will be replaced by the reply CPRB from the coprocessor during the call to CSNC_SP_SCSRFBSS().

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.
Filling the rule array is easy using the `BuildParmBlock()` function:

```c
BuildParmBlock ( ptr1,
    1,
    SIZE_OF_RULE * (*pRuleCount), pRuleArray );
```

To parse a rule array with the `rule_check()` function, two more structures are used. A pointer to a `RULE_BLOCK` is passed to the function to be parsed. Note that the rule array format within the `ESSS_request_block` structure is, in fact, a `RULE_BLOCK`.

### RULE_BLOCK

<table>
<thead>
<tr>
<th>Size in Bytes</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>length</td>
<td>Total length of the rule array and this field, in big-endian order.</td>
</tr>
<tr>
<td>1</td>
<td>data</td>
<td>First character of first rule array element, if rule_array_length is greater than 2; otherwise, this will be the first byte of the verb unique data length field.</td>
</tr>
</tbody>
</table>

The other structure required is a `RULE_MAP` structure. This maps 8-byte strings into a value array, assigning a unique value to each string, and 1 or more strings to each position in the array, depending on mutual exclusion issues.

### RULE_MAP

<table>
<thead>
<tr>
<th>Size in Bytes</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>keyword</td>
<td>The 8-byte keyword to be tested for, plus a null terminator.</td>
</tr>
<tr>
<td>1</td>
<td>order_no</td>
<td>Group number: All rules which are mutually exclusive to each other should have the same group number.</td>
</tr>
<tr>
<td>4</td>
<td>map_value</td>
<td>The numeric value associated with this rule.</td>
</tr>
</tbody>
</table>
To check the values in the rule array, use the rule_check() function:

```c
rule_check((RULE_BLOCK *)&pReq->rule_array_length,
        sizeof(aRuleMap)/sizeof(RULE_MAP),
        aRuleMap,
        aRuleValue,
        &returnMessage);
```

Immediately following the rule array in the REQUEST_REPLY_BUF is the verb unique data. Two types of structures are supplied for working with verb unique data, the VUD_DATA_RECORD, which is a length/tag/data structure (the data preceded by a DATA_RECORD_HEADER structure), and the verb_unique_data_structure, which is a length/data structure.

### DATA_RECORD_HEADER

<table>
<thead>
<tr>
<th>Size in Bytes</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Length</td>
<td>Total length of this verb unique data (VUD) item and this structure, in big-endian order.</td>
</tr>
<tr>
<td>2</td>
<td>Flag</td>
<td>User-defined type of data</td>
</tr>
</tbody>
</table>

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.

```c
BuildParmBlock( ptr,
        2,
        DATA_HEADER_LENGTH, &DataHeader,
        dataLength, Data );
```

The FindFirstDataBlock() and jFindFirstDataBlock() functions return a pointer to a VUD_DATA_RECORD, so that you can access your data in this format easily.

### VUD_DATA_RECORD

<table>
<thead>
<tr>
<th>Size in Bytes</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Length</td>
<td>Total length of this verb unique data (VUD) item and this structure, in big-endian order.</td>
</tr>
<tr>
<td>2</td>
<td>Flag</td>
<td>User-defined type of data</td>
</tr>
<tr>
<td>1</td>
<td>Data</td>
<td>First byte of data.</td>
</tr>
</tbody>
</table>

```c
FindFirstDataBlock( pCPRB, SEL_REPLY_BLK, &pVerbDataRecord );
```
if(pVerbDataRecord->Flag == EncryptedKey )
{
  memcpy( pKeyParameter, &pVerbDataRecord->Data,
          pVerbDataRecord->Length - DATA_HEADER_LENGTH );
}

To use the length/data format for your verb unique data, use the BuildParmBlock() function to place a two-byte length field before the data (remembering to byte-swap the value).

swappedDataLength = htoas( sizeof(short) + dataLength );
BuildParmBlock( ptr, 2,
                sizeof(short), &swappedDataLength, dataLength, Data);

To retrieve the above data, you must first cast the verb_unique_data_structure as a VUD_DATA_RECORD:

FindFirstDataBlock ( pCPRB, SEL_REPLY_BLK, (VUD_DATA_RECORD **)pVerbUniqueDataStructure);
*pLengthParm = atohs( pVerbUniqueDataStructure->verb_unique_data_length) - LENGTH_FIELD_SIZE;
memcpy( pReturnedData, &pVerbUniqueDataStructure->verb_unique_data, *pLengthParm);

If the only piece of data which is being passed has a fixed length (for example, if it is a structure), you need not use either of the verb structures shown:

BuildParmBlock( ptr, 1,
                sizeof(Structure), &Structure);

Then to access the data:

FindFirstDataBlock( pCPRB, SEL_REPLY_BLOCK, (VUD_DATA_RECORD **)pData);
memcpy( pStructure, pData, sizeof(*pStructure) );

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.

<table>
<thead>
<tr>
<th>Size in Bytes</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Length</td>
<td>Total length of this verb unique data (VUD) item and this structure, in big-endian order.</td>
</tr>
</tbody>
</table>
On the host side, you will need to declare a `KEY_FIELD_HEADER` structure for each key you will be passing to the coprocessor. On the coprocessor, you will need to declare a `KEY_FIELD_HEADER` structure for each key you will be passing to the host. If you are passing a token to be written to the key storage file, you must declare two `KEY_FIELD_HEADER` structures, and pass first the label of the key to write to, then the key token to write into the key storage file.

```c
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()` and `jfind_first_key_block()` functions return a pointer to a `key_data_structure`:

```c
key_data_structure
```

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.
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.

The CCA dispatcher on the coprocessor creates (from the CPRB) an Operation Tracking Structure
(OPTX) which it passes as one of the parameters to the command processor.

<table>
<thead>
<tr>
<th>Size</th>
<th>Name</th>
<th>Description</th>
<th>writeable</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>in_Data</td>
<td>Address of request data block</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>outData</td>
<td>Address of reply data block</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>tData</td>
<td>For you to allocate – common code will free – DEPRECATED</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>fnid</td>
<td>“T2” from input CPRB</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>subfnid</td>
<td>Subfunction code from request parameter block</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>Domain</td>
<td>The domain currently in use, in native byte order (for workstation UDXes, this is 0).</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>roleID</td>
<td>Role of the logged on user</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>flags</td>
<td>bitwise flags</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>c.t2.in_CPRB</td>
<td>pointer to input CPRB</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>c.t2.outCPRB</td>
<td>pointer to output CPRB</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>c.t2.in_parm_block</td>
<td>pointer to request parameter block</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>c.t2.in_ext_block</td>
<td>pointer to request extension block</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>c.t2.in_vud</td>
<td>point to VUD section of request parameter block</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>c.t2.outparm_block</td>
<td>Pointer to reply parameter block: NOT SET on input, set this to outCPRB-&gt;reply_parm_block</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>c.t2.outtext_block</td>
<td>Not for use by UDX user</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>tr</td>
<td>Mapped BBRAM for persistent traces – Data</td>
<td>Yes</td>
</tr>
</tbody>
</table>
## Operation Tracking Structure

<table>
<thead>
<tr>
<th>Size</th>
<th>Name</th>
<th>Description</th>
<th>writeable</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>tok</td>
<td>storage for writeable data</td>
<td>Yes</td>
</tr>
<tr>
<td>128</td>
<td>tok2</td>
<td>storage for writeable data</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>optok_cnt</td>
<td>The number of tokens to parse from the input key block structure.</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Following Fields are handled by fiber macros or CCA base code and should not</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>be disturbed</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>pxcRequestHandle</td>
<td>Value returned by xcInitMappings call, also passed to xcrypto library functions</td>
<td>NO</td>
</tr>
<tr>
<td>4</td>
<td>vbCallBack</td>
<td>Function pointer is set to indicate address to restart this function when an xcrypto library call is completed.</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>vbType</td>
<td>Type of verb: Async or Run Isolated. UDX verbs are all Async</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>QeTrack</td>
<td>Used for tracking structures on the queues</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>reqid</td>
<td>Request ID, not used</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>udxmfte</td>
<td>Local structure to hold information about the UDX values. Set by command decoder, do not use.</td>
<td>No</td>
</tr>
<tr>
<td>16</td>
<td>pmfte</td>
<td>Not for use by UDX user</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>thrData.sysRC</td>
<td>Return code from hardware operation</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>thrData</td>
<td>Further pieces of thread/operation structure data used by library level.</td>
<td>no</td>
</tr>
<tr>
<td>4</td>
<td>myfiber</td>
<td>Used by common code, xCbr function and CALLXC macros to run the fiber system</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>pStack</td>
<td>Virtual address of stack data allocated by comm manager for this fiber. Zeroed by wiper code</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>stackPA</td>
<td>Physical address for stack</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>pDynAlcBase</td>
<td>Base of the heap of data available for allocation</td>
<td>No</td>
</tr>
<tr>
<td>Size</td>
<td>Name</td>
<td>Description</td>
<td>writeable</td>
</tr>
<tr>
<td>------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td></td>
<td>pDynAlcEnd</td>
<td>End of the dynamically allocatable space for this optx.</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>pDynAlcUsed</td>
<td>End of the currently used dynamically allocatable space</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>workrc</td>
<td>Return code from engine</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>pxCOpHandle</td>
<td>Operation handle returned by call to xcrypto library. This is used by library to identify a unique call, and by command processor to identify a call to determine its status</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>xctries</td>
<td>Number of times an attempt has been tried, returning with XC_TIMEOUT</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>pvpkt</td>
<td>Pointer to the virtual Packet</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>useReply</td>
<td>Indicates whether reply area has been filled correctly</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>tlv[2]</td>
<td>Tag-length-pointer structures for reply</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ptlvs[2]</td>
<td>Pointers to TLV structures, either present above or allocated</td>
<td></td>
</tr>
<tr>
<td></td>
<td>putrep</td>
<td>The actual putReply structure to return</td>
<td></td>
</tr>
<tr>
<td></td>
<td>parmlen</td>
<td>Transmit buffer length</td>
<td></td>
</tr>
<tr>
<td></td>
<td>datalen</td>
<td>Transmit data length</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pdbufuf</td>
<td>Pointer to the data buffer to transmit</td>
<td></td>
</tr>
</tbody>
</table>

On the coprocessor, it is important to note that much of the input operation tracking structure is not writeable by the UDX code and an attempt to write to that space will cause the coprocessor to crash. The *tok* and *tok2* blocks within the OPTX structure are intended for this purpose.

The communication functions which begin with "j" -- *jFindFirstDataBlock*, *jFindNextDataBlock*, *jfind_first_key_block*, *jfind_next_key_block*, and *jparm_block_valid* have all been created to use the OPTX fields. All of the "jFind" and "jfind..." take parameters of *VUDPTR*, which is the type of the *c.t2.in_vud* field, while the *jparm_block_valid* function takes a parameter of type *RBFPTR*, which is the type of the *c.t2.in_parm_block* field.
15.2 Structures used to manage keys and key tokens

In addition to the communication structures for key tokens mentioned above, there are several structures used to parse the various key types.

AES key tokens are stored in a 64-byte structure with the following characteristics:

<table>
<thead>
<tr>
<th>Size</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>tokenFlag</td>
<td>0x01 (internal key)</td>
</tr>
<tr>
<td>3</td>
<td>resvd1</td>
<td>0 (reserved field)</td>
</tr>
<tr>
<td>1</td>
<td>version</td>
<td>0x04 (token version)</td>
</tr>
<tr>
<td>1</td>
<td>resvd2</td>
<td>0x00 (reserved field)</td>
</tr>
<tr>
<td>1</td>
<td>flags</td>
<td>XOR of 0x80 (key encrypted), 0x40 (CV present), and 0x20 (key and mkvp absent)</td>
</tr>
<tr>
<td>1</td>
<td>checkSum</td>
<td>LRC checksum of clear key value</td>
</tr>
<tr>
<td>8</td>
<td>mkvp</td>
<td>MKVP= 0 for clear key, otherwise AESMKVP</td>
</tr>
<tr>
<td>32</td>
<td>aesKey</td>
<td>Key left justified and padded on right with zeroes</td>
</tr>
<tr>
<td>8</td>
<td>ctlVec</td>
<td>0 (no control vectors are defined)</td>
</tr>
<tr>
<td>2</td>
<td>bitLenOfClrKey</td>
<td>Length in bits of clear key value</td>
</tr>
<tr>
<td>2</td>
<td>byteLenOfEncKey</td>
<td>Length in bytes of clear key value 0 for clear key, 32 for encrypted key</td>
</tr>
<tr>
<td>4</td>
<td>tvv</td>
<td>Token validation value</td>
</tr>
</tbody>
</table>

DES key tokens are stored in a 64-byte structure with the following characteristics:

<table>
<thead>
<tr>
<th>Size</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>tokenFlag</td>
<td>0x01 (internal key)</td>
</tr>
<tr>
<td>1</td>
<td>reserved1</td>
<td>0 (reserved field)</td>
</tr>
<tr>
<td>2</td>
<td>Oldmkvp</td>
<td>0 (currently unused)</td>
</tr>
<tr>
<td>1</td>
<td>version</td>
<td>0x04 (token version)</td>
</tr>
<tr>
<td>1</td>
<td>reserved2</td>
<td>0x00 (reserved field)</td>
</tr>
<tr>
<td>1</td>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>reserved3</td>
<td>0x00 (reserved field)</td>
</tr>
<tr>
<td>8</td>
<td>mkvp</td>
<td>MKVP= 0 for clear key, otherwise AESMKVP</td>
</tr>
<tr>
<td>8</td>
<td>keyLeft</td>
<td>Key (for a single-length DES key) or first half of a double length DES key.</td>
</tr>
</tbody>
</table>
### Des Key Token Structure

<table>
<thead>
<tr>
<th>Size</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>KeyRight</td>
<td>0 (for a single-length DES key) or second half of a double length DES key.</td>
</tr>
<tr>
<td>8</td>
<td>cvBase</td>
<td>Control Vector (for a single-length DES key) or first half of a double-length DES key.</td>
</tr>
<tr>
<td>8</td>
<td>CvExtn</td>
<td>0 (for a single-length DES key) or second half of a double length DES key.</td>
</tr>
<tr>
<td>8</td>
<td>reserved5</td>
<td>0 (reserved)</td>
</tr>
<tr>
<td>3</td>
<td>reserved6</td>
<td>0 (reserved)</td>
</tr>
<tr>
<td>1</td>
<td>tokenmarks</td>
<td>Token marks – 0 for internal tokens,</td>
</tr>
<tr>
<td>4</td>
<td>tvv</td>
<td>Token validation value</td>
</tr>
</tbody>
</table>

The `RsaRecoverClearKeyTokenUnderXport()` function requires a type of double_length_key.

### Double Length Key

<table>
<thead>
<tr>
<th>Size in Bytes</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>left</td>
<td>First 8 bytes of double length key.</td>
</tr>
<tr>
<td>8</td>
<td>right</td>
<td>Second 8 bytes of double length key.</td>
</tr>
</tbody>
</table>

The functions `load_first_mk_part()` and `combine_mk_parts()` require a TRIPLE_LENGTH_KEY:

### Triple Length Key

<table>
<thead>
<tr>
<th>Size in Bytes</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>first</td>
<td>First 8 bytes of triple length key.</td>
</tr>
<tr>
<td>8</td>
<td>middle</td>
<td>Second 8 bytes of triple length key.</td>
</tr>
<tr>
<td>8</td>
<td>last</td>
<td>Last 8 bytes of triple length key.</td>
</tr>
</tbody>
</table>

For RSA keys, there are two structures defined. For CCA RSA key tokens, the header is:

### RsaKeyTokenHeader

<table>
<thead>
<tr>
<th>Size</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>tokenId</td>
<td>0x01 (internal key)</td>
</tr>
<tr>
<td>1</td>
<td>version</td>
<td>0 (reserved field)</td>
</tr>
<tr>
<td>2</td>
<td>tokenLength</td>
<td>Length of token including this header</td>
</tr>
<tr>
<td>4</td>
<td>reserved</td>
<td>0 (reserved field)</td>
</tr>
</tbody>
</table>

320 IBM 4767 CCA UDX Reference
Each RSA key begins with the preceding structure, and this is the structure that is passed to the majority of the RSA functions.

For generating keys, however, the following structure is used:
<table>
<thead>
<tr>
<th>Size</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>dpOffset</td>
<td>Offset from the beginning of the structure to the dp value (CRT key) or 0 for a modulus exponent key.</td>
</tr>
<tr>
<td>4</td>
<td>dqOffset</td>
<td>Offset from the beginning of the structure to the dq value (CRT key) or 0 for a modulus exponent key.</td>
</tr>
<tr>
<td>4</td>
<td>apOffset</td>
<td>Offset from the beginning of the structure to the ap value (CRT key) or 0 for a modulus exponent key.</td>
</tr>
<tr>
<td>4</td>
<td>aqOffset</td>
<td>Offset from the beginning of the structure to the aq value (CRT key) or 0 for a modulus exponent key.</td>
</tr>
<tr>
<td>4</td>
<td>r_Offset</td>
<td>0 (blinding values not currently used.)</td>
</tr>
<tr>
<td>4</td>
<td>r1Offset</td>
<td>0 (blinding values not currently used.)</td>
</tr>
<tr>
<td>1</td>
<td>tokenData</td>
<td>First byte of the key data.</td>
</tr>
</tbody>
</table>

Valid types for `sccRSAKeyToken_t`:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA_PUBLIC_MODULUS_EXPONENT</td>
<td>A public key (n and e) (not useful for key generation).</td>
</tr>
<tr>
<td>RSA_PRIVATE_MODULUS_EXPONENT</td>
<td>A public key and a private key (n, e, d).</td>
</tr>
<tr>
<td>RSA_X931_PRIVATE_MODULUS_EXPONENT</td>
<td>A public and private key (n, e, d) but n is restrained to X.931 valid lengths.</td>
</tr>
<tr>
<td>RSA_PRIVATE_CHINESE_REMAINDER</td>
<td>A public and private key (n, e, p, q, dp, dq, ap, aq).</td>
</tr>
<tr>
<td>RSA_X931_PRIVATE_CHINESE_REMAINDER</td>
<td>A public and private key (n, e, p, q, dp, dq, ap, aq) but n is restrained to X9.31 valid lengths.</td>
</tr>
</tbody>
</table>
### 15.3 Structures used for handling the master keys

One data structure is required for handling the master keys, the mk_selectors structure.

The mk_selectors data structure is used to indicate which of several master keys to use in a given master key function. There are separate old, current, and new master keys for asymmetric and symmetric master keys in each available domain.

<table>
<thead>
<tr>
<th>Size in Bytes</th>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>mk_set</td>
<td>Domain of master key set. Should be the same value as the Domain field in the CPRB structure, which is <code>MK_SET_DEFAULT</code> on a non-z/OS coprocessor.</td>
</tr>
<tr>
<td>Enumeration :</td>
<td>old_mk</td>
<td></td>
</tr>
<tr>
<td></td>
<td>current_mk</td>
<td></td>
</tr>
<tr>
<td></td>
<td>new_mk</td>
<td></td>
</tr>
<tr>
<td>Enumeration:</td>
<td>mk_register</td>
<td>Which of the three registers to access.</td>
</tr>
<tr>
<td>SYM_MK,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASYM_MK,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES_MK,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>APKA_MK,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOTH_MK</td>
<td>type_mks</td>
<td>Which type of master key to access.</td>
</tr>
</tbody>
</table>

---

Data structures 323
15.4 *Structures used in cache handling*
Only one data structure is required for the use of the cache functions, the short_tag_t:

<table>
<thead>
<tr>
<th>short_tag_t</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size in Bytes</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

You may choose to cast a 2-byte value as a short_tag_t for the function call.
15.5 Constants used in key parsing and building
Constants for the otk_build_a_token and otk_build_tokens functions:

<table>
<thead>
<tr>
<th>BUILD INPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLDI_USE_OLD_MK</td>
</tr>
<tr>
<td>Encrypt the token with the old master key, rather than the current master key.</td>
</tr>
<tr>
<td>BLDI_DES_CHK.CV_EMK</td>
</tr>
<tr>
<td>Adds an extra Control Vector check.</td>
</tr>
<tr>
<td>BLDI_DES_CHK.CV.NULL</td>
</tr>
<tr>
<td>Test whether the control vector is all zeros (DES only)</td>
</tr>
<tr>
<td>BLDI_DES_EXT.CV.NULL</td>
</tr>
<tr>
<td>Extensive test on zero control vector (DES only)</td>
</tr>
<tr>
<td>BLDI_IGN_KEY_TYPE</td>
</tr>
<tr>
<td>Ignore the key type, just encrypt and put into the token.</td>
</tr>
<tr>
<td>BLDI_DES_IGN_PARITY</td>
</tr>
<tr>
<td>Ignore the parity of the key being built (DES only)</td>
</tr>
<tr>
<td>BLDI_AES_IGN_BITLEN</td>
</tr>
<tr>
<td>Ignore the bit length check for an AES key (because the key is empty)</td>
</tr>
<tr>
<td>BLDI_AES_SET.CV.PR</td>
</tr>
<tr>
<td>Builder will set the CV present bit in the token.</td>
</tr>
<tr>
<td>BLDI_CHK_SRC.KEYLN</td>
</tr>
<tr>
<td>Check that the key provided is the correct length for the token.</td>
</tr>
<tr>
<td>BLDI_DES_SINGLE.R</td>
</tr>
<tr>
<td>Take an 8-byte DES clear key and create a 16-byte key with replicated halves.</td>
</tr>
<tr>
<td>BLDI_DES_SET_TM.CVZ</td>
</tr>
<tr>
<td>Create an S390 token with a triple-length key.</td>
</tr>
<tr>
<td>BLDI_DES_SET_EXT.V1</td>
</tr>
<tr>
<td>Create an external DES key token with zero-cv.</td>
</tr>
<tr>
<td>BLDI_KEY_CLEAR</td>
</tr>
<tr>
<td>Create a clear (RSA) key token</td>
</tr>
<tr>
<td>BLDI_CKEY_IS.OPK</td>
</tr>
<tr>
<td>The ckey pointer is pointing to the Object Protection key, not the RSA private key.</td>
</tr>
<tr>
<td>BLDI_CHK.TO KO_SZ</td>
</tr>
<tr>
<td>Return E_SIZE if the tokout size is too small. (Useful on the host).</td>
</tr>
<tr>
<td>BLDI_DES_XLAT_CNTL</td>
</tr>
<tr>
<td>Set the XLATE control vector bits to prevent the (DES) token from being translated.</td>
</tr>
</tbody>
</table>

Constants used for the parse_keys function:

<table>
<thead>
<tr>
<th>TOKEN INPUT FLAGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOKI_OPT_KEY</td>
</tr>
<tr>
<td>The key in this positions is optional, if it it not present do not mark an error.</td>
</tr>
<tr>
<td>TOKI_CONT_ON_ERR</td>
</tr>
<tr>
<td>TOKI_WR_TOK</td>
</tr>
<tr>
<td>TOKI_MUST_BE_TOK</td>
</tr>
<tr>
<td>TOKI_USE_PASSED_TOK</td>
</tr>
<tr>
<td>TOKIKEK_HERE</td>
</tr>
<tr>
<td>TOKIKEK_NEEDED</td>
</tr>
<tr>
<td>TOKIKEK_ON_EMPTY</td>
</tr>
<tr>
<td>TOKIIGN_KEY</td>
</tr>
<tr>
<td>TOKIIGN_CV</td>
</tr>
<tr>
<td>TOKINODECR_KEY</td>
</tr>
<tr>
<td>TOKINOCHECK</td>
</tr>
<tr>
<td>TOKINOWARN_OMK</td>
</tr>
<tr>
<td>TOKINOERR_NMK</td>
</tr>
<tr>
<td>TOKINOERR_OMK</td>
</tr>
<tr>
<td>TOKINOCKTOK</td>
</tr>
<tr>
<td>TOKICHECK MK</td>
</tr>
</tbody>
</table>

**ALGORITHM INPUT**

<p>| ALGIGN_ALGKEY_CHK | DES keys: ignore even parity; AES keys none; RSA keys: call VerifyRsaKeyTokenConsistency before other checks. |</p>
<table>
<thead>
<tr>
<th>ALGI_IGN_S390_ZEROCV</th>
<th>DES: ignore zero CV keys in this position.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALGI_MH_ENCR_KEY</td>
<td>DES: no meaning; AES: key in this position cannot be a clear key; RSA: the key in this position cannot be a clear key.</td>
</tr>
<tr>
<td>ALGI_GET_RTND</td>
<td>If a key label is found here, get the RETAINED key from the on-board storage.</td>
</tr>
<tr>
<td>ALGI_RSA_RTND_ALLOW_EXT</td>
<td>If the retained key is marked as EXTERNAL, don't throw an error, continue processing.</td>
</tr>
<tr>
<td>ALGI_PKA_INT MH_PVT</td>
<td>RSA: Internal key token must have a private key.</td>
</tr>
<tr>
<td>ALGI_PKA_EXT MH_PVT</td>
<td>RSA: External key token must have a private key.</td>
</tr>
<tr>
<td>ALGI_RSA_IGN_PVT</td>
<td>RSA: Ignore the private key, if any.</td>
</tr>
<tr>
<td>ALGI_PKA_INT MH_PUB</td>
<td>RSA: Internal key token must have a public key.</td>
</tr>
<tr>
<td>ALGI_PKA_EXT MH_PUB</td>
<td>RSA: External key token must have a public key.</td>
</tr>
<tr>
<td>ALGI_RSA_CHK_E_LEN</td>
<td>RSA: Key must have non-zero e Length, which is checked early. (Changes order of return codes)</td>
</tr>
<tr>
<td>ALGI_RSA_CHK_PUB_EVEN</td>
<td>RSA: key must have an even public exponent.</td>
</tr>
<tr>
<td>ALGI_RSA_CHK_MAX_MOD</td>
<td>RSA: RSA key must have modulus within the FCV restriction.</td>
</tr>
<tr>
<td>ALGI_TBLK_IGN_CHK</td>
<td>RSA: TBLK token</td>
</tr>
<tr>
<td>ALGI_TBLK_TX_RSAPUB</td>
<td>TBLK and RSA: A Trusted block is expected. Validate the trusted block, translate that to the RSA public key and validate that. Return the RSA clear key in ckey, and the built-up RSA public token in -&gt;tok.</td>
</tr>
<tr>
<td>ALGI_TBLK_NOMACRECOV</td>
<td>Don't verify the MAC over the trusted block once the trusted block itself is validated. Just finish processing the block.</td>
</tr>
<tr>
<td>ALGI_TBLK_MACINPLC</td>
<td>TBLK: A trusted block is expected, decrypt the MAC key in place.</td>
</tr>
<tr>
<td>ALGI2.Des MH Ext V1</td>
<td>DES: the token must have the tokenflags set and a zero CV.</td>
</tr>
<tr>
<td>ALGI2.Des MH NULL CV</td>
<td>DES: the Control vector of the key must be all zeros</td>
</tr>
<tr>
<td>ALGI2.Des EXT XT CHKS</td>
<td>DES: Check that this external token has all the reserved bits zero, the flag byte has no unused bits turned on, and the mkvp and old mkvp fields are zero.</td>
</tr>
<tr>
<td>ALGI2.IGN AES LRC</td>
<td>AES: Ignore the AES Longitudinal Redundancy Check</td>
</tr>
<tr>
<td>ALGI2_CHK_RSA ASYM MK</td>
<td>RSA: test the status on the ASYM master key</td>
</tr>
<tr>
<td>ALGI2_VAR MH KEY</td>
<td>HMAC token must have a key present.</td>
</tr>
<tr>
<td>ALG12_DES_FAKE_LEN</td>
<td>DES: Don't return an error if the DES token length is described as &gt; 64.</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td>ALG12_RKX_TRX_HERE</td>
<td>DES: Parser should run a special parser to translate this RKX key token.</td>
</tr>
<tr>
<td>ALG12_RKX_EKEK_HERE</td>
<td>DES: Parser should run a special parser to translate this RKX key token.</td>
</tr>
<tr>
<td>ALG12_RKX_EXP_HERE</td>
<td>DES: Parser should run a special parser to translate this RKX key token.</td>
</tr>
<tr>
<td>ALG12_RKX_TBLK_HERE</td>
<td>DES: Parser should run a special parser to translate this RKX key token.</td>
</tr>
</tbody>
</table>

Flags which may be returned by the parser:

<table>
<thead>
<tr>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOKO_CKEY_ALLOC</td>
<td>Memory was allocated at the “ckey” pointer to store the key. This memory must be freed after processing. The memory will be freed by the CCA dispatcher. The command processor should not do so and also should not overwrite this flag in the operation tracking structure.</td>
</tr>
<tr>
<td>TOKO_CKEY_OPT_ALLOC</td>
<td>Memory was allocated at the “ckey_opt” pointer to store the key. The memory must be freed after processing. The memory will be freed by the CCA dispatcher. The command processor should not do so and also should not overwrite this flag in the operation tracking structure.</td>
</tr>
<tr>
<td>TOKO_TOKALLOC</td>
<td>Memory was allocated at the “tok” pointer. The memory will be freed by the CCA dispatcher. The command processor should not do so and also should not overwrite this flag in the operation tracking structure.</td>
</tr>
<tr>
<td>TOKO_S390_ZEROCV</td>
<td>This key had a zero CV.</td>
</tr>
<tr>
<td>TOKO_S390_V1</td>
<td>This key was an S390 Version 1 key token.</td>
</tr>
<tr>
<td>TOKO_S390_VAR21</td>
<td>This key was a VARIANT 21 key (S390).</td>
</tr>
<tr>
<td>TOKO_DES_V1_TM_NOT_SINGLE</td>
<td>This key is a Version 1 key (no CV) and the tokenmarks indicate it is not a single length key.</td>
</tr>
<tr>
<td>TOKO_XPORT_PROHIB</td>
<td>This key is not exportable</td>
</tr>
<tr>
<td>TOKO_NOCV</td>
<td>This key is set to be a nocv key (S390 only).</td>
</tr>
<tr>
<td><strong>TOKO.CV MISSING</strong></td>
<td>This keys flags indicate that it has no CV.</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td><strong>TOKO.MK_CUR</strong></td>
<td>Token was encrypted with Current MK.</td>
</tr>
<tr>
<td><strong>TOKO.MK_OLD</strong></td>
<td>Token was encrypted under Old MK. (CPRB-&gt;secy_error_code has been set)</td>
</tr>
<tr>
<td><strong>TOKO.MK_NEW</strong></td>
<td>Only used if <strong>TOKI_NO_ERR_NMK</strong> was in in_flags, otherwise parser returns an error.</td>
</tr>
<tr>
<td><strong>TOKO.KEK_USED</strong></td>
<td>This key is a KEK and it was used to decrypt another key in the list. Only used if <strong>TOKI.KEK HERE</strong> was in in_flags.</td>
</tr>
<tr>
<td><strong>TOKO.GOT_RTND</strong></td>
<td>There was a label in the field, the key has been recovered from the on-board key storage file. Only if <strong>TOKI.GET_RTND</strong> present in in_flags.</td>
</tr>
<tr>
<td><strong>TOKO.PKA_NO_PVT_KEY</strong></td>
<td>This internal RSA key has no private key.</td>
</tr>
<tr>
<td><strong>TOKO.PKA_NO_PUB_KEY</strong></td>
<td>This internal RSA key has no public key.</td>
</tr>
<tr>
<td><strong>TOKO.PKA_CLR_PVT</strong></td>
<td>This internal RSA key has a private key, but that key was not encrypted.</td>
</tr>
<tr>
<td><strong>TOKO.AES_CLR_KEY</strong></td>
<td>This AES key was not encrypted.</td>
</tr>
</tbody>
</table>
16 Modifying a 4765 UDX for the 4767

This chapter discusses the changes needed to convert a UDX written for the IBM 4765 to run on an IBM 4767.

16.1 Makefile changes

On the coprocessor, your code must be compiled with gcc 4.4.6 or greater, rather than gcc 4.1. The jffs2 filesystem must be created by mkfs.jffs2 revision 1.5 or greater. The modified directory structure requires a different set of include directories. See the instructions in “Understanding the UDX environment” on page 1 and “Building a CCA user-defined extension” on page 14 as well as the sample makefile in samples/makefiles/toolkit_sample.inc for more details. The binary files required for code are different, since the UDX is now built as a library which is dynamically loaded rather than building an executable that links with the CCA code in a library.

16.2 Changes to function calls

The operation tracking structure is provided to your command processor is now a required parameter for all functions which eventually call the xcrypto library for services (for example, encryption or decryption or signing.) This has changed the function signature of many existing functions.

Calling the libxccmn.so library directly is not recommended. Any instances you have of direct calls should be replaced by the CALLXC macros to ensure that the fiber management is handled correctly. Take particular notice of the fact that all values which are passed to or from the hardware (keys, vectors, or even 8-byte result fields) are now pointers in the request block structures.
17 Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM’s product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM’s intellectual property rights or other legally protectable rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user’s responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, 10594, USA.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

17.1 Copying and distributing softcopy files

For online versions of this manual, 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.

17.2 Trademarks

IBM, AIX, and z/OS are trademarks of the IBM Corporation in the United States or other countries or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

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

Other company, product, and service names may be trademarks or service marks of others.
## List of abbreviations and acronyms

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

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

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

**adapter.** Synonym for expansion card.

**agent.** (1) An application that runs within the IBM 4767 PCIe Cryptographic Coprocessor. (2) Synonym for secure cryptographic coprocessor application or for Linux cryptographic coprocessor application.

**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.

**battery-backed random access memory (BBRAM).** Random access memory that uses battery power to retain data while the system is powered off. The IBM 4767 PCIe Cryptographic Coprocessor uses BBRAM to store persistent data for xC applications, as well as the coprocessor device key.

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

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

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

**cleartext.** (1) Data that has not been altered by any cryptographic process. (2) Synonym for plaintext. (3) See also ciphertext.
Common Cryptographic Architecture (CCA). A comprehensive set of cryptographic services that furnishes a consistent approach to cryptography on major IBM computing platforms. Application programs can access these services through the CCA application program interface.

Common Cryptographic Architecture (CCA) API. The application program interface used to call Common Cryptographic Architecture functions; it is described in the IBM 4767 CCA Basic Services Reference and Guide.

coprocessor. (1) A supplementary processor that performs operations in conjunction with another processor. (2) A microprocessor on an expansion card that extends the address range of the processor in the host system, or adds specialized instructions to handle a particular category of operations; for example, an I/O coprocessor, math coprocessor, or a network coprocessor.

Coprocessor Load Utility (CLU). A program used to load validated code into the IBM 4767 PCIe Cryptographic Coprocessor.

Cryptographic Coprocessor (IBM 4767). An expansion card that provides a comprehensive set of cryptographic functions to a workstation.

cryptographic node. A node that provides cryptographic services such as key generation and digital signature support.

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

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


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

DES. Data Encryption Standard.

device driver. (1) A file that contains the code needed to use an attached device. (2) A program that enables a computer to communicate with a specific peripheral device; for example, a printer, videodisc player, or a CD drive.

E
cipher. (1) To scramble data or convert it to a secret code that masks its meaning. (2) Contrast with decipher.

enciphered data. (1) Data whose meaning is concealed from unauthorized users or observers. (2) See also ciphertext.

F

feature. A part of an IBM product that can be ordered separately from the essential components of the product.
**flash memory.** A specialized version of erasable programmable read-only memory (EPROM) commonly used to store code in small computers.

**H**

**host.** As regards to the IBM 4767 PCIe Cryptographic Coprocessor, the workstation into which the coprocessor is installed.

**I**

**Interactive Code Analysis Tool (ICAT).** A remote debugger used to debug applications running within the IBM 4767 PCIe Cryptographic Coprocessor.

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

**intrusion latch.** A software-monitored bit. The intrusion latch does not trigger the destruction of data stored within the coprocessor.

**J**

**jumper.** A wire that joins two unconnected circuits.

**K**

**key.** In computer security, a sequence of symbols used with an algorithm to encipher or decipher data.

**M**

**master key.** In computer security, the top-level key in a hierarchy of KEKs.

**message authentication code (MAC).** In computer security, (1) a number of value derived by processing data with an authentication algorithm. (2) The cryptographic result of block cipher operations, on text or data, using the cipher block chain (CBC) mode of operation.

**miniboot.** Software within the IBM 4767 PCIe Cryptographic Coprocessor designed to initialize the operating system and to control updates to flash memory.

**P**

**passphrase.** In computer security, a string of characters known to the computer system and to a user; the user must specify it to gain full or limited access to the system and to the data stored therein.

**Peripheral Component Interconnect (PCI).** A 32-bit parallel computer expansion card standard.

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

**Peripheral Component Interconnect eXtended (PCI-x).** A 64-bit version of the PCI, utilized in the IBM 4764 Cryptographic Adapter.
**private key.** (1) In computer security, a key that is known only to the owner and used with a public key algorithm to decipher data. Data is enciphered using the related public key. (2) Contrast with public key. (3) See also public key algorithm.

**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

**random number generator (RNG).** A system designed to output values that cannot be predicted. Since software-based systems generate predictable, pseudo-random values, the 4767 card uses a hardware-based system to generate true random values for cryptographic use.

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

S

**Security Relevant Data Item (SRDI).** Data that is securely stored by the IBM 4767 Cryptographic Adapter.

V

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

**vital product data (VPD).** A structured description of a device or program that is recorded at the manufacturing site.

**VPD.** Vital product data.

W

**workstation.** A terminal or microcomputer, usually one that is connected to a mainframe or a network, and from which a user can perform applications.

**Numerics**

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

**4765.** IBM 4765 PCIe Cryptographic Coprocessor.

**4767.** IBM 4767 PCIe Cryptographic Coprocessor.
20 Index

4767 functions........................................20
abbreviations........................................333
ac_check_authorization..........................269
ac_chg_prof_auth_data..........................257
ac_chg_prof_exp_date............................259
ac_del_profile....................................260
ac_del_role........................................250
ac_get_list_sizes.................................245
ac_get_profile......................................261
ac_get_role.........................................251
ac_init.............................................246
ac_list_profiles..................................262
ac_list_roles......................................252
ac_load_profiles..................................263
ac_load_roles......................................253
ac_lu_add_user....................................270
ac_lu_drop_user...................................271
ac_lu_get_ks.......................................272
ac_lu_get_num_users...............................273
ac_lu_get_role.....................................274
ac_lu_kd_decr.....................................275
ac_lu_kd_hmacgen................................276
ac_lu_kd_hmacover.................................277
ac_lu_list_users..................................278
ac_lu_query_user..................................279
ac_query_profile..................................265
ac_query_role......................................255
ac_reinit..........................................247
ac_reset_logon_fail_cnt..........................266
access control manager functions..............241
acmx_ctl_errorcode...............................248
aes_engine_encipher.............................151, 152
aes_engine_with_MK..............................101
aes_token_t........................................319
aes_unwrap........................................154
aes_unwrap_with_hash............................153
aes_unwrap_with_hash_and_MK....................103
aes_unwrap_with_MK.............................105
aes_wrap...........................................157
aes_wrap_with_hash...............................155
aes_wrap_with_hash_and_MK.......................107
aes_wrap_with_MK................................109
algorithm input....................................326
build input........................................325
building a CCA UDX.................................14
coprocessor piece..................................17
files used..........................................14
workstation piece..................................16
BuildParmBlock....................................35
cache management functions......................280
cache_clear........................................282
cache_delete........................................283
cache_delete_item.................................284
cache_get_item ....................................285
cache_get_item_b.................................286
cache_init..........................................287
cache_status.......................................288
cache_write_item..................................289
CalculateWordLength.............................179
CALLXC_FIBER.....................................23
CALLXCDES.........................................24
CALLXCDES_3KEY..................................28
CALLXCDES3KEY_ECB.............................29
CALLXSCHA1.........................................32
CALLXSCHA2.........................................31
CALLXCTDES........................................26
cas_adjust parity.................................158
cas_des_key_token_check .......................159
cas_get_key_type..................................161
cas_key_length....................................162
cas_key_token_vvv_check........................163
cas_parity_odd....................................164
Cas_proc_retc.....................................38
CasAesMasterKeyCheck..........................165
CasBuildCv.........................................166
CasBuildToken.....................................167
CasCurrentMKvp...................................169
CasGetFileDescriptor............................292
CasMasterKeyCheck................................170
CasOldMKvp.........................................172
CCA communication structures...................7
CCA dispatcher.....................................5
CCA services.......................................6
CCA SRDI manager functions.....................227
chained operations................................126
changes..............................................
  function call.................................330
  makefile........................................330
check master key functions......................93
check_access_auth_fcn............................293
clear_master_keys................................79
close_cca_sdri....................................232
communication functions........................33
communication structures, CCA..................7
compute_mk_verification_pattern.................94
computeHMAC_SHA1.................................128
computeHMAC_SHA256...............................129
constants.........................................
  key parsing and building........................325
coprocessor 4767 functions.......................20

Index 339