IBM SecureWay Cryptographic Products
IBM 4758 PCI Cryptographic Coprocessor
Custom Software Interface Reference

May 9, 2000

Security Solutions and Technology Department

IBM Corporation
8501 IBM Drive
Charlotte, North Carolina 28262-8563

09-MAY-00, 14:47

Note!

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

Second Edition (May, 2000)

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

IBM does not stock publications at the address given below; requests for IBM publications should be made to your IBM representative or to the IBM branch office that serves your location.

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

Contents

About This Book

The *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference* describes the secure cryptographic coprocessor (SCC) application programming interface (API) function calls that applications running on the cryptographic coprocessor use to obtain cryptographic and communication services from the operating system. It also describes the function calls that applications running on the host use to interact with applications running on the cryptographic coprocessor. Finally, it describes the interface the coprocessor provides to allow a host device driver to initialize the coprocessor, test its operation, and transfer data between the host and the coprocessor.

The primary audience for this manual is developers who are creating applications to use with the coprocessor. This manual should be used in conjunction with the manuals listed under “Custom Software Publications” on page ix.

Prerequisite Knowledge

The reader of this book should understand how to perform basic tasks (including editing, system configuration, file system navigation, and creating application programs) on the host machine. Familiarity with the SCC application development process (as described in the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide*) may also be helpful. Readers who intend to write a host device driver to manage the coprocessor should have a general understanding of the AMCC S5933 PCI Controller (as provided by the *AMCC S5933 PCI Controller Data Book*).

Organization of This Book

Chapter 1, “Overview” discusses the separation of the SCC API into host-side and coprocessor-side components and describes how an application on the host interacts with an application on the cryptographic coprocessor. It includes the source for a sample host application and a sample coprocessor application that illustrate this interaction.

Chapter 2, “Host-Side API” describes the host-side portion of the SCC API in detail.

Chapter 3, “Coprocessor-Side API” describes the coprocessor-side portion of the SCC API in detail.

Chapter 4, “Coprocessor Interface for Host Device Drivers” describes in detail how the host device driver that manages the coprocessor interacts with the coprocessor and transfers information from the host to the coprocessor and from the coprocessor to the host.

Appendix A, “Error Code Formatting” details return codes common to the host and coprocessor APIs.

Appendix B, “DES Weak, Semi-Weak, and Possibly Weak Keys” lists keys that are not suitable for use as DES keys. The random number generator can be instructed not to return any of these numbers.

Appendix D, “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 **bold** type.
- Variable information, parameters, and file names are presented in *italic* type.
- The names of menu items, buttons, and fields displayed in graphical user interface (GUI) applications are presented in **bold** type.
- System response in a non-GUI environment is presented in `monospace` type.
- Function names and return codes are presented in **bold** type.
- Web addresses and file directory locations are presented in *italic* type.

Related Publications

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

General Interest

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

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

CCA Support Program Publications

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

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

PKCS #11 Support Program Publications

The following publication may be of interest to readers who intend to develop applications using PKCS #11 services.

- *IBM 4758 PCI Cryptographic Coprocessor PKCS #11 Support Program Installation Manual*

Custom Software Publications

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

- *IBM 4758 PCI Cryptographic Coprocessor Custom Software Installation Manual*
- *IBM 4758 PCI Cryptographic Coprocessor Interactive Code Analysis Tool (ICAT) User's Guide*
- *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Overview*
- *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference*
- *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System C Runtime Library Reference*
- *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer's Toolkit Guide*
- *IBM 4758 PCI Cryptographic Coprocessor CCA User Defined Extensions Programming Reference*
- *AMCC S5933 PCI Controller Data Book*, available from Applied Micro Circuits Corporation, 6290 Sequence Drive, San Diego, CA 92121-4358. Phone 1-800-755-2622 or 1-619-450-9333. The manual is available online as an Adobe Acrobat** PDF file at *http://www.amcc.com/pdfs/pciprod.pdf*.

Cryptography Publications

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

- "Application Support Architecture for a High-Performance, Programmable Secure Coprocessor," J. Dyer, R. Perez, S.W. Smith, and M. Lindemann, 22nd National Information Systems Security Conference, October 1999.
- "Validating a High-Performance, Programmable Secure Coprocessor," S.W. Smith, R. Perez, S.H. Weingart, and V. Austel, 22nd National Information Systems Security Conference, October 1999.
- "Building a High-Performance, Programmable Secure Coprocessor," S.W. Smith and S.H. Weingart, Research Report RC21102, IBM T.J. Watson Research Center, February 1998. A revised version of this paper appeared in *Computer Networks* 31:831-860, April 1999.
- "Using a High-Performance, Programmable Secure Coprocessor, S.W. Smith, E.R. Palmer, and S.H. Weingart, in *FC98: Proceedings of the Second International Conference on Financial Cryptography*, Anguilla, February 1998. Springer-Verlag LNCS. 1998. ISBN 3-540-64951-4
- "Smart Cards in Hostile Environments," H. Gobioff, S.W. Smith, J.D. Tygar, and B.S. Yee, *Proceedings of the Second USENIX Workshop on Electronic Commerce*, 1996.
- "Secure Coprocessing Research and Application Issues," S.W. Smith, Los Alamos Unclassified Release LA-UR-96-2805, Los Alamos National Laboratory, August 1996.
- "Secure Coprocessing in Electronic Commerce Applications," B.S. Yee and J.D. Tygar, in *Proceedings of the First USENIX Workshop on Electronic Commerce*, New York, July 1995.

- “Transaction Security Systems,” D.G. Abraham, G.M. Dolan, G.P. Double, and J.V. Stevens, in *IBM Systems Journal* Vol. 30 No. 2, 1991, G321-0103.
- “Trusting Trusted Hardware: Towards a Formal Model for Programmable Secure Coprocessors,” S.W. Smith and V. Austel, in *Proceedings of the Third USENIX Workshop on Electronic Commerce*,” Boston, August 1998.
- “Using Secure Coprocessors,” B.S. Yee (Ph.D. thesis), Computer Science Technical Report CMU-CS-94-149, Carnegie-Mellon University, May 1994.
- “Cryptography: It’s Not Just for Electronic Mail Anymore,” J.D. Tygar and B.S. Yee, Computer Science Technical Report, CMU-CS-93-107, Carnegie Mellon University, 1993.
- “Dyad: A System for Using Physically Secure Coprocessors,” J.D. Tygar and B.S. Yee, Harvard-MIT Workshop on Protection of Intellectual Property, April 1993.
- “An Introduction to Citadel—A Secure Crypto Coprocessor for Workstations,” E.R. Palmer, Research Report RC18373, IBM T.J. Watson Research Center, 1992.
- “Introduction to the Citadel Architecture: Security in Physically Exposed Environments,” S.R. White, S.H. Weingart, W.C. Arnold, and E.R. Palmer, Research Report RC16672, IBM T.J. Watson Research Center, 1991.
- “An Evaluation System for the Physical Security of Computing Systems,” S.H. Weingart, S.R. White, W.C. Arnold, and G.P. Double, Sixth Computer Security Applications Conference, 1990.
- “ABYSS: A Trusted Architecture for Software Protection,” S.R. White and L. Comerford, IEEE Security and Privacy, Oakland 1987.
- “Physical Security for the microABYSS System,” S.H. Weingart, IEEE Security and Privacy, Oakland 1987.
- *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*, Bruce Schneier, John Wiley & Sons, Inc. ISBN 0-471-12845-7 or ISBN 0-471-11709-9
- *ANSI X9.31 Public Key Cryptography Using Reversible Algorithms for the Financial Services Industry*
- *IBM Systems Journal* Volume 30 Number 2, 1991, G321-0103
- *IBM Systems Journal* Volume 32 Number 3, 1993, G321-5521
- *IBM Journal of Research and Development* Volume 38 Number 2, 1994, G322-0191
- *USA Federal Information Processing Standard (FIPS):*
  - *Data Encryption Standard,* 46-1-1988
  - *Secure Hash Algorithm,* 180-1, May 31, 1994
  - *Cryptographic Module Security,* 140-1
- *Derived Test Requirements for FIPS PUB 140-1*, W. Havener, R. Medlock, L. Mitchell, and R. Walcott. MITRE Corporation, March 1995.
- *ISO 9796 Digital Signal Standard*
- *Internet Engineering Taskforce RFC 1321*, April 1992, MD5
- *Secure Electronic Transaction Protocol Version 1.0*, May 31, 1997

IBM Research Reports can be obtained from:

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

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

Other IBM Cryptographic Product Publications

The following publications describe products that utilize the IBM Common Cryptographic Architecture (CCA) application program interface (API).

- *IBM Transaction Security System General Information Manual*, GA34-2137
- *IBM Transaction Security System Basic CCA Cryptographic Services*, SA34-2362
- *IBM Transaction Security System I/O Programming Guide*, SA34-2363
- *IBM Transaction Security System Finance Industry CCA Cryptographic Programming*, SA34-2364
- *IBM Transaction Security System Workstation Cryptographic Support Installation and I/O Guide*, GC31-4509
- *IBM 4755 Cryptographic Adapter Installation Instructions*, GC31-4503
- *IBM Transaction Security System Physical Planning Manual*, GC31-4505
- *IBM Common Cryptographic Architecture Services/400 Installation and Operators Guide, Version 2*, SC41-0102
- *IBM Common Cryptographic Architecture Services/400 Installation and Operators Guide, Version 3*, SC41-0102
- *IBM ICSF/MVS General Information*, GC23-0093
- *IBM ICSF/MVS Application Programmer's Guide*, SC23-0098

Summary of Changes

This first edition of the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference* contains product information that is current with the IBM 4758 PCI Cryptographic Coprocessor announcements made in May, 1998.

Changes made to this edition in January, 1999 include:

- Chapter 3, Public Key Algorithm Functions—Clarified that offsets within an RSA key token are offsets from the beginning of the token, not from the beginning of the `tokenData` buffer; clarified the behavior of sccRSA when the length of the input and output buffers is not the same as the length of the modulus; clarified the use of the blinding values in a key token (they are only of use with private keys); and added descriptions of the new function provided by sccDSAKeyGenerate (the caller can specify the values of *p*, *q*, and *g*) and sccDSA (the caller can specify that the input has already been hashed).

  Nonvolatile Memory Functions—Clarified the effects sccCreate4UpdatePPD, sccSavePPD, and sccUpdatePPD have on flash.

Changes made to this edition in March, 1999 include:

- Chapter 3—Added sections on hash functions and large integer modular math functions. Clarified how the RSA X9.31 support works.

Changes made to this edition in July, 1999 include:

- Chapter 3—Documented the enhanced sccGetRandomNumber interface.

Changes made to this edition in September, 1999 include:

- Addition of the APIs provided by the IBM 4758 PCI Cryptographic Coprocessor model 002 hardware (external buffer support in sccSHA1 and new sccTDES and sccTestRandomNumber APIs).

Changes made to this edition in October, 1999 include:

- Chapter 2—Clarified the values returned by sccRequest when an error is detected.
- Chapters 2 and 3—Added sections to the beginning of each chapter to list the required header files, location of the sample code, and so on.

Chapter 1. Overview

The secure cryptographic coprocessor (SCC) application programming interface (API) allows applications running on the host to interact with applications running on the cryptographic coprocessor. The SCC API includes a set of functions an application running on the host may invoke (the host-side API) and a set of functions an application running on the cryptographic coprocessor may invoke (the coprocessor-side API).

This chapter describes how an application on the host interacts with an application on the cryptographic coprocessor and illustrates the flow of data and messages among the various agents involved in the process. It also briefly discusses the message-passing model the cryptographic coprocessor operating system (CP/Q) uses for interprocess communication and how synchronous and asynchronous versions of some coprocessor-side API calls are built on top of this model. Finally, it includes the source for a sample host application and a sample coprocessor application that illustrates the interaction.

Software Architecture

Figure 1-1 on page 1-2 illustrates the principal software agents in the system, with connections between components that directly communicate with one another.

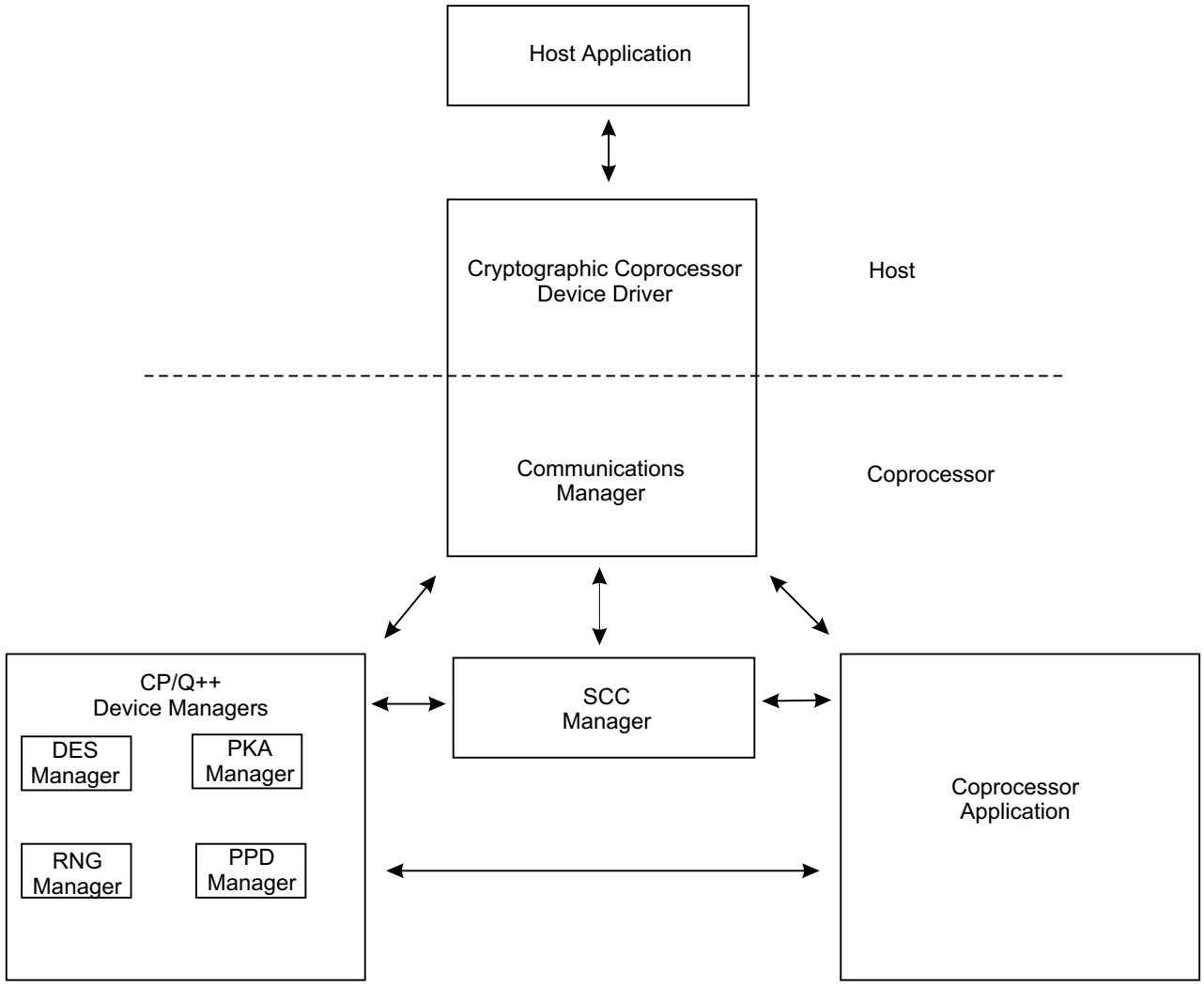

Figure 1-1.

- All requests for service from the host application are sent via the host cryptographic coprocessor device driver. The host device driver forwards requests from the host application across the PCI bus to a special device manager (the Communications Manager) on the cryptographic coprocessor. The host device driver also reads data from and writes data to the host application's address space on behalf of the coprocessor application.
- The Communications Manager forwards requests from the host to the coprocessor application and forwards requests to read or write data from the coprocessor application to the host.
- CP/Q++ device managers control sensitive parts of the coprocessor (for example, the DES encryption hardware).
- The SCC Manager maintains a table of all registered coprocessor applications so that host requests can be routed to the proper destination. The SCC Manager also helps ensure that the CP/Q++ device managers act only on the behalf of properly authorized coprocessor applications.

Host and Coprocessor Interaction

The host application and coprocessor application exchange information as follows:

1. The coprocessor application calls sccSignOn to register with the SCC Manager and passes the SCC Manager a structure of type `sccAgentID_t` that identifies the coprocessor application.
2. The host application calls sccOpenAdapter to establish a communications channel between the host application and the Communications Manager.
3. The coprocessor application calls sccGetNextHeader to await the receipt of a request from the host.
4. The host application sends a request to the Communications Manager. The request includes an `sccAgentID_t` structure that identifies the coprocessor application for which the request is intended. The Communications Manager scans the SCC Manager's list of registered coprocessor applications and then forwards the request to the application whose `sccAgentID_t` structure matches the one in the request.
5. The coprocessor application processes the request. As part of its processing, the coprocessor application may:
   a. Call sccGetBufferData to read data from the host application.
   b. Request services (for example, DES or RSA encryption or decryption, random number generation) from CP/Q++ device managers. These device managers may inspect the table maintained by the SCC Manager to verify the coprocessor application has registered itself and has the proper authority to make the request.
   c. Call sccPutBufferData to write data to the host application.
6. The coprocessor application calls sccEndRequest to notify the host application that the request is complete and supplies a return code for the request and (optionally) the result of the request.

   (Steps 3 through 6 are repeated each time the host generates a request.)
7. The host application calls sccCloseAdapter to close the communications channel between the host application and the Communications Manager.

The host application must initiate all transactions. The coprocessor application cannot interrupt the host application and can transfer data only in response to a request from the host application. Several host applications may interact with the coprocessor at the same time.

Synchronous and Asynchronous Calls

Interprocess communication on CP/Q is most commonly accomplished by passing messages from one process to another. Messages are unidirectional, but each message bears a unique identifier and by convention the reply to a message includes the message's identifier. This allows the original sender to distinguish the reply from other messages.

CP/Q's message passing model greatly simplifies support of asynchronous (nonblocking) interprocess function calls. Associated with each CP/Q task[1] is a queue on which messages destined for that task are placed by default[2]. To make an asynchronous call, an application sends a message to the target process and records the message identifier. At a later time, the application can wait for the reply (a message containing the original message identifier) or examine the message queue to see whether a reply has arrived. For synchronous (blocking) interprocess function calls, the application simply sends a message and immediately waits for the reply.

Refer to the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference*, CPQR-2A01 for detailed information about CP/Q messages and message passing conventions.

Software Attacks and Defensive Coding

Coprocessor applications run in a secure environment and often manipulate or manage sensitive data. To reduce the likelihood that this data will be compromised, a coprocessor application must assume any host application to which it provides service may have been written by an adversary in an attempt to mount an attack on the coprocessor application. For example, the coprocessor application should thoroughly validate any arguments provided by the host application.

CP/Q attempts to limit the amount of damage an errant coprocessor application can cause. If an application terminates (via exit() or by returning from main()) or one of the tasks in the application generates an exception (for example, divide by zero or addressing exception) and the application did not supply a fault handler for the task[3], CP/Q halts the system. No further processing occurs until the coprocessor is rebooted.

Sample Applications

The following applications illustrate the concepts described in this chapter. The applications include the following header and source code files:

OEM_hdr.h	Defines the protocol used between the host and coprocessor applications
OEM_card.c	Coprocessor application source code
OEM_host.c	Host application source code

This simple example illustrates the transport mechanism between the host and the coprocessor; it does not utilize the cryptographic capabilities of the coprocessor.

Various structures can be passed between cooperating host and coprocessor applications. It is important to compile both applications with the same

1 A CP/Q task is a dispatchable unit and is equivalent to a thread on many other operating systems, including Windows NT.

2 A task can create additional queues to which messages can be directed as well.

3 One of the arguments to CPCreateTask identifies the task's fault handler. Refer to the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for details.

structure-packing conventions. This can be controlled with a compiler command-line flag, or by a pragma in common header files.

Because of the 32-bit data boundary alignment imposed by the PCI architecture, host buffers should be aligned to a 32-bit boundary for best performance. All buffer lengths must be multiples of four bytes.

Header File

The following code (*OEM_hdr.h*) defines the protocol used in communication between the host application and the coprocessor application:

```
#ifndef OEM_HDR_H
#define OEM_HDR_H

   /* The first two bytes in an AgentID are assigned by IBM.
   /* The value given here is used with the RSA key supplied in the IBM 4758
      Application Program Development Toolkit. */
   ** The trailing 14 bytes are chosen by the OEM.
   */
   #define OEM_X1   6
   #define OEM_X2   0

   #define OEM_AGENT_1  { { OEM_X1, OEM_X2},\
                          { 0,0,0,0,0,0,0,0,0,0 },\
                          {0},\
                          {0},\
                          {1} }

   /* These are the commands understood between the partners below,
   ** and some of the conversation semantics.
   **
   */
   #define OEM_CMD_HELLO   0
   #define OEM_CMD_DONE    1

   #define OEM_BUFLEN      32  /* must be a 4-byte multiple */
   #define OEM_TOCARD_BUF   0
   #define OEM_TOHOST_BUF   0

   #define OEM_HELLO_MSG   "Hello from the 4758."
   #define OEM_DONE_MSG    "Bye, you said you were done."

   #define OEM_GOOD        0
   #define OEM_BAD         1

#endif  /*OEM_HDR_H*/
```

Coprocessor Application Code

The following code (*OEM_card.c*) signs on with the SCC Manager and sets up a test buffer:

```
/*start-of-c-file*/

#include <string.h>
#include <stdlib.h>
#include <cpqlib.h>
#include <scc_int.h>
/* scctypes.h is included by scc_int.h; it defines the req'd types */
#include "OEM_hdr.h"
```

```
sccAgentID_t      agentID_OEM = OEM_AGENT_1;
sccRequestHeader_t reqHdr;
char               buf[OEM_BUFLEN];
char               testBuf[OEM_BUFLEN];

void main(void)
{
   long  rc;
   long  status;

   /* The first thing we do is signon with the SCC_Mgr.
   ** Until this is done, any host-side program trying to send a message
   ** to me (using my agentID) will get the equivalent of
   ** "addressee unknown".
   **
   ** This signon asks that the SCC_Mgr use the default message queue
   ** (which is my task queue) in order to pass requests to this program.
   ** Subsequent calls to GetNextHeader will then use 0 for msgq_id.
   */
   rc = sccSignOn( &agentID_OEM, NULL );
   if( rc )
      exit( rc );

   /* For this simple program, we just set up our testBuf.
   ** Other activities, such as acquiring memory or getting
   ** previously saved data (keys, prior configuration)
   ** would be done as needed before waiting for work.
   */
   strcpy( testBuf, OEM_HELLO_MSG );

   /* The main loop...pick up one work item at a time */
   for(;;)
   {
      /* Wait until a request comes our way */
      rc = sccGetNextHeader( &reqHdr, 0, SVCWAITFOREVER );
      if( rc )
         exit( rc );

      /* Assume the normal case, all will be well */
      status = OEM_GOOD;

      /* Switch on the commands we accept.
      ** By convention, the command is stored in the UserDefined
      ** field of the request header.
      */
      switch( reqHdr.UserDefined )
      {
         case OEM_CMD_HELLO:
            /* We expect to receive a single buffer from the
            ** host, into which we are to put our message
            */
            if( reqHdr.InBufferLength[OEM_TOHOST_BUF]!=OEM_BUFLEN )
            {
               status = OEM_BAD;
               break;
            }
            memcpy( buf, testBuf, OEM_BUFLEN );
            break;

         case OEM_CMD_DONE:
            /* We expect to receive a buffer from the
            ** host, which holds a copy of our hello, and
            ** a buffer into which we are to put our goodby.
            */
```

```
            if( reqHdr.OutBufferLength[OEM_TOCARD_BUF]!=OEM_BUFLEN
                || reqHdr.InBufferLength[OEM_TOHOST_BUF]!=OEM_BUFLEN )
            {
               status = OEM_BAD;
               break;
            }
            /* Get the buffer being sent to us (OUT from host) */
            rc = sccGetBufferData( reqHdr.RequestID, OEM_FROMHOST_BUF,
                                   buf, OEM_BUFLEN );
            if( rc )
            {
               status = rc;
               break;
            }
            /* Verify the echo */
            if( 0!=memcmp( buf, testBuf, OEM_BUFLEN ) )
            {
               status = OEM_BAD;
               break;
            }
            /* Copy answer to results buffer */
            strcpy( buf, OEM_DONE_MSG );
            break;

         default:
            status = OEM_BAD;
            break;

      }/*switch*/

      /* End this request, sending the host the content
      ** of 'buf' if we have good status.
      ** We must specify the request ID (there may be other
      ** agents at work in the system, or we may choose to
      ** juggle several requests at once).
      */
      if( status== OEM_GOOD )
         rc = sccEndRequest( reqHdr.RequestID, OEM_TOHOST_BUF,
                                       buf, OEM_BUFLEN, OEM_GOOD );
      else
         rc = sccEndRequest( reqHdr.RequestID, 0, NULL, 0, status );

      if( rc )
         exit( rc );

   } /*for*/

} /*main()*/

/*end-of-c-file*/
```

Host Application Code

The following code (*OEM_host.c*) opens the coprocessor and requests a “hello” message from the coprocessor application.

```
/*start-of-c-file*/
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <scc_host.h>
/* scctypes.h is included by scc_host; It defines the req'd types. */
#include "OEM_hdr.h"
```

```
sccAgentID_t agent = OEM_AGENT_1
sccAdapterHandle_t  adpt;
sccRB_t             rb_OEM;

/* This buffer should be on a 32-bit boundary */
char                buf[OEM_BUFLEN];

long main(int argc, char **argv )
{
   long   rc;
   long   iters;

   /* Open the earliest adapter.
   ** We could ask how many there are, and try them one-by-one
   ** until we find the first adapter containing our companion,
   ** agent OEM_AGENT_1.
   */
   rc = sccOpenAdapter( 0L, &adpt );
   if( rc )
   {
      printf( "sccOpenAdapter 0 returned %#X\n", rc );
      return( rc );
   }

   /* How many times should we say hello?
   ** This dumb program actually understands one parameter!
   */
   if( argc>1 )
   {
      if( 0==sscanf( argv[1], "%i", &iters ) )
         iters = 1;
      if( iters<=0 )
         iters = 1;
   }
   else
      iters = 1;

   /* Set the non-zero fields for the HELLO.
   ** The command asks for a hello from our companion
   ** on the coprocessor, supplying a buffer for the response.
   */
   memcpy (&rb_OEM.AgentID, &agent, sizeof (sccAgentID_t));
   rb_OEM.UserDefined = OEM_CMD_HELLO;
   rb_OEM.InBufferLength[OEM_TOHOST_BUF] = OEM_BUFLEN;
   rb_OEM.pInBuffer[OEM_TOHOST_BUF] = buf;

   for( ; iters>0; --iters )
   {
      rc = sccRequest( adpt, &rb_OEM );
      if( rc )
      {
         printf( "sccRequest returned %#X\n", rc );
         break;
      }
      if( rb_OEM.Status )
      {
         printf( "sccRequest.Status %#X\n", rb_OEM.Status );
         break;
      }
      printf( "Iter %d, message received: '%s'\n", iters, buf );
   }/*for*/

   /* Now set up the GOODBY command.
```

```
   ** We echo back the last hello; we have iterated at least once.
   ** It uses the same input buffer (to host) as the HELLO,
   ** and supplies an additional outbound (to-card) buffer.
   */
   rb_OEM.UserDefined = OEM_CMD_DONE;
   rb_OEM.OutBufferLength[OEM_TOCARD_BUF] = OEM_BUFLEN;
   rb_OEM.pOutBuffer[OEM_TOCARD_BUF] = buf;

   rc = sccRequest( adpt, &rb_OEM );
   if( rc )
   {
      printf( "last sccRequest returned %#X\n", rc );
   }
   else if( rb_OEM.Status )
   {
      /* We have a good rc, so the request went to the card
      ** and now we need to check the card's answer.
      */
      printf( "last sccRequest.Status %#X\n", rb_OEM.Status );
   }
   else
   {
      /* We have a good rc (transport) and good status (OEM_card).
      ** Therefore we have a last message to print.
      */
      printf( "last message received: '%s'\n", buf );
   }

   /* Indicate we no longer need to talk to the coprocessor.
   ** Always good to be polite.
   */
   sccCloseAdapter( adpt );

   return( rc );

}/*main()*/

/*end-of-c-file*/
```

How to Compile and Link the Sample Programs

OEM_host.c should be compiled and linked just like any other application on the host. Link with libscc.a and libodm.a on AIX, with crypto.lib on OS/2, and with cryptont.lib on Windows NT.

Refer to the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer's Toolkit Guide* for information on how to compile and link OEM_card.c (the coprocessor application) and how to load the executable into the coprocessor.

The host application and the coprocessor application must agree on the packing conventions for structures used in the interface between them (defined in *OEM_hdr.h* and the various SCC .h files). You may need to add pragmas to these files to ensure this is the case.

Chapter 2. Host-Side API

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

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

This chapter describes each of the functions supplied by the host API. Each description includes the function prototype (in C), the inputs to the function, the outputs returned by the function, and the most common return codes generated by the function.

General Information

Host-Side API Functions

The host API includes the following functions:

sccAdapterCount Determine the number of cryptographic coprocessors installed in the host. See page 2-3.

sccGetAdapterID Obtain coprocessor identification data. See page 2-4.

sccOpenAdapter Establish a communications channel to a specific coprocessor. See page 2-6.

sccRequest Send a request across an open communications channel to a specific application and receive the reply. See page 2-8.

sccCloseAdapter Close a communications channel that was previously opened via a call to sccOpenAdapter. See page 2-11.

All host API calls are synchronous (that is, the calls do not return until the corresponding function is complete).

Header Files

The prototypes for these functions are contained in *scc_host.h*. Other header files used to create host applications are *scctypes.h* and *scc_err.h*. The code that implements the host API functions is in *libscc.a* on AIX, in *crypto.lib* on OS/2, and in *cyptont.lib* on Windows NT. The NT library is included in the IBM 4758 Application Program Development Toolkit. Refer to the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer's Toolkit Guide* for details. On AIX, the cryptographic coprocessor uses the object data manager (ODM) and both *libscc.a* and the ODM library must be linked with an application.

The prototypes in *scc_host.h* include keywords, preprocessor directives, or both that ensure the functions are called using the appropriate linkage convention regardless of the default linkage convention in effect during compilation. For clarity, the prototypes that appear in this chapter do not include this syntax.

Sample Code

Examples of the use of many of the host API functions can be found in the following files shipped with the IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer's Toolkit:

- *scctk\src\samples\oem_samp\OEM_host.c*
- *scctk\src\samples\rte\hre.c*
- *scctk\src\samples\skeleton\host\skelhost.c*

Error Codes

Appendix A, "Error Code Formatting" on page A-1 describes the format of a return code. Note that although the host API calls return a 32-bit return code, in some cases the low order bits of the value contain additional information rather than a constant value:

- If the cryptographic coprocessor's power-on self test (POST) fails, a host API call may return POST_ERR in the high order 16 bits of the return code and a value that identifies the specific POST checkpoint that failed in the low order 16 bits. POST checkpoint identifiers are subject to change and are not made publicly available.
- If the cryptographic coprocessor microcode detects an attempt to tamper with the physical security of the card, a host API call may return HDDSecurityTamper in the high order 24 bits of the return code and the state of the hardware tamper bits (defined in *scctypes.h*) in the low order 8 bits.
- If a host API call invokes the host operating system for service and the invocation fails, the host API call may return HOST_OS_ERR in the high order 16 bits of the return code and the error code returned by the system call (or a portion of it) in the low order 16 bits.

sccAdapterCount - Count Installed Coprocessors

sddAdapterCount determines the number of cryptographic coprocessors installed in the host computer.

Function Prototype

```
long  sccAdapterCount(sccAdapterNumber_t *pAdapterCount);
```

Input

On entry to this routine:

`pAdapterCount` must contain the address of a variable in which an item of type `sccAdapterNumber_t` can be stored.

Output

On successful exit from this routine:

`*pAdapterCount` contains the number of coprocessors installed in the host.

Notes

Coprocessors Counted During Boot

The number of coprocessors installed on the host is determined by the device driver for the cryptographic coprocessor when the host is booted and is not updated to reflect any physical changes to the system (for example, removal of a coprocessor while the host is suspended or in hibernation) until a subsequent reboot.

sccAdapterNumber_t is Arithmetic

An item of type `sccAdapterNumber_t` can be used in an arithmetic context (for example, as an array index or for-loop terminal value).

Return Codes

Common return codes generated by this routine are:

HDDGood (i.e., 0) The operation was successful.

HDDError The operation was unsuccessful.

HOST_OS_ERR An error occurred on a call to the host operating system (the low-order 16 bits contain the return code from the call that failed).

Refer to *scc_err.h* for a comprehensive list of return codes.

sccGetAdapterID - Get Coprocessor Identification

sccGetAdapterID obtains information about the coprocessor's hardware level from the AMCC S5933 PCI Controller's configuration registers.

Function Prototype

```
long sccGetAdapterID(sccAdapterNumber_t  AdapterNumber,
                     sccAdapterID_t     *pAdapterID);
```

Input

On entry to this routine:

`AdapterNumber` uniquely identifies one of the cryptographic coprocessors installed in the host. `AdapterNumber` must contain an integer greater than or equal to zero and less than the value returned in the `*pAdapterCount` output from a call to sccAdapterCount.

`pAdapterID` must contain the address of a variable in which an item of type `sccAdapterID_t` can be stored.

Output

On successful exit from this routine:

`*pAdapterID` contains information about the coprocessor's hardware level. The fields of the `sccAdapterID_t` structure are set as follows:

- `VendorID` is the contents of the AMCC S5933's Vendor Identification Register (VID). This value matches the 16-bit word at offset zero in the `AMCC_EEPROM` field returned by sccGetConfig.
- `DeviceID` is the contents of the AMCC S5933's Device Identification Register (DID). This value matches the 16-bit word at offset 2 in the `AMCC_EEPROM` field returned by sccGetConfig.
- `RevisionID` is the contents of the AMCC S5933's Revision Identification Register (RID). This value matches the byte at offset 8 in the `AMCC_EEPROM` field returned by sccGetConfig.
- `ReservedByte` is zero.
- `ReservedShort` is zero.
- `ReservedLong` is zero.

Notes

sccOpenAdapter Not Required

A host application may call sccGetAdapterID before a communications channel to the coprocessor is opened via sccOpenAdapter.

Return Codes

Common return codes generated by this routine are:

HDDGood (i.e., 0) The operation was successful.

HDDError The operation was unsuccessful.

HOST_OS_ERR An error occurred on a call to the host operating system (the low-order 16 bits contain the return code from the call that failed).

Refer to *scc_err.h* for a comprehensive list of return codes.

sccOpenAdapter - Open Channel to Coprocessor

sccOpenAdapter establishes a communications channel between a host application and a specific coprocessor. The host application may interact with any application running on the coprocessor via the channel and may only interact with applications on coprocessors with which communications channels have been established.

Function Prototype

```
long sccOpenAdapter(sccAdapterNumber_t  AdapterNumber,
                    sccAdapterHandle_t *pAdapterHandle);
```

Input

On entry to this routine:

`pAdapterHandle` must contain the address of a variable in which an item of type `sccAdapterHandle_t` can be stored.

`AdapterNumber` uniquely identifies one of the cryptographic coprocessors installed in the host. `AdapterNumber` must contain an integer greater than or equal to zero and less than the value returned in the `*pAdapterCount` output from a call to sccAdapterCount.

Output

On successful exit from this routine:

`*pAdapterHandle` contains a handle that can be used in subsequent host API calls to identify the cryptographic coprocessor to which the call refers.

Notes

Assignment of Numbers to Coprocessors

The number assigned to a particular cryptographic coprocessor depends on the order in which information about devices in the system is presented to the device driver by the host operating system. At the present time there is no way to tell *a priori* which coprocessor will be assigned a given number.

Multiple Communications Channels

A host application may establish communications channels to more than one coprocessor by calling sccOpenAdapter multiple times with different `AdapterNumber` arguments. A host application may also establish more than one communications channel to a single coprocessor by calling sccOpenAdapter multiple times with the same `AdapterNumber` argument. In either case, each call to sccOpenAdapter returns a new handle in `*pAdapterHandle`.

Return Codes

Common return codes generated by this routine are:

HDDGood (i.e., 0)	The operation was successful.
HDDInvalidParm	One or more inputs were not valid.
HDDTooManyOpens	The device driver or host operating system cannot create a new communications channel due to lack of resources.
HDDAccessDenied	The device driver cannot open a communications channel to interact with an application on the adapter because another process on the host already has a channel open in order to interact with the adapter's system software.
HDDDeviceBusy	The cryptographic coprocessor is still booting up; try again later.
HDDError	The operation was unsuccessful.
HOST_OS_ERR	An error occurred on a call to the host operating system (the low-order 16 bits contain the return code from the call that failed).

Refer to *scc_err.h* for a comprehensive list of return codes.

sccRequest - Send Request to Coprocessor Application

sccRequest sends a request across a communications channel to a specific application running on the target coprocessor and waits for and receives the application's reply.

Function Prototype

```
long sccRequest(sccAdapterHandle_t  AdapterHandle,
                sccRB_t            *pRequestBlock);
```

Input

On entry to this routine:

`AdapterHandle` uniquely identifies a communications channel to one of the cryptographic coprocessors installed in the host. `AdapterHandle` must contain the handle returned in the `*pAdapterHandle` output from a call to sccOpenAdapter.

`pRequestBlock` must contain the address of a request block whose fields are initialized as follows:

- `AgentID` identifies the coprocessor application to which the request is to be sent. See "sccSignOn - Register to Receive Requests" on page 3-7 for details.
- `reserved` must be zero.
- `UserDefined` contains an arbitrary value. The coprocessor application receives this value as soon as a call it makes to sccGetNextHeader returns with information about the request.

  The host application will typically set this field to a value that identifies the action the coprocessor application is to perform on behalf of the host application, although it need not be used in this manner.
- `pOutBuffer` and `OutBufferLength` define as many as four buffers from which the coprocessor application may read data. `pOutBuffer[i]` is the address of the first byte of a buffer and `OutBufferLength[i]` is the length in bytes of the buffer (which must be a multiple of 4). The coprocessor application calls sccGetBufferData to read the contents of a buffer and consequently has complete control over which buffers are read and when a buffer is read, although whenever the coprocessor application reads a buffer it reads the entire buffer at once.

  These buffers should be aligned on a 4-byte boundary for the best performance.[1]

  The device driver and CP/Q++ operating system ensure that a coprocessor application cannot read past the end or prior to the start of a buffer. The host application should set any unused `pOutBuffer` entries to NULL and the corresponding `OutBufferLength` entries to zero to avoid inadvertently allowing a coprocessor application to read data it is not authorized to see.
- `pInBuffer` and `InBufferLength` define as many as four buffers into which the coprocessor application may write data. `pInBuffer[i]` is the address of the first byte of a buffer and `InBufferLength[i]` is the length in bytes of the buffer (which must be a multiple of 4). The coprocessor application calls

[1] If `pOutBuffer[i]` is not aligned on a 4-byte boundary, the device driver must copy it to an aligned buffer before it can be read.

sccPutBufferData or sccEndRequest to write data to a buffer and consequently has complete control over which buffers are written and when a buffer is written, although whenever the coprocessor application writes data to a buffer the first byte of data is written to the first byte of the buffer.

These buffers should be aligned on a 4-byte boundary for best performance.[2]

The device driver and CP/Q++ operating system ensure that a coprocessor application cannot write past the end or prior to the start of a buffer. The host application should set any unused `pInBuffer` entries to NULL and the corresponding `InBufferLength` entries to zero to avoid inadvertently allowing a coprocessor application to write to areas it is not authorized to modify.

Output

On successful exit from this routine:

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

- The buffers defined by `pInBuffer` and `InBufferLength` on entry to the routine may contain information written by the coprocessor application via a call to sccPutBufferData or sccEndRequest. `InBufferLength[i]` is updated to reflect the actual number of bytes written by the coprocessor application to the corresponding buffer (and must be a multiple of 4).
- The coprocessor application calls sccEndRequest to indicate that it has finished processing a request, at which point sccRequest returns to the host application. `Status` is set to the value of the last argument passed by the coprocessor application to sccEndRequest.

Notes

Length of Reply Unknown

The host application may not know *a priori* how much data the coprocessor application may write to a buffer. In this case, the host application must ensure the buffer is large enough to hold the largest possible result. The actual length of the result will appear in the appropriate `InBufferLength` entry after the call to sccRequest returns.

Overlapped Input and Output Buffers

The behavior of a coprocessor application is undefined if a buffer from which the application reads data and a buffer to which the application writes data overlap. In particular, if the coprocessor application writes data to a buffer and then reads from the same buffer, it may see the original contents of the buffer, the new contents of the buffer, a mix of old and new contents, or random bytes.

Changes to Buffers while Request Outstanding

Because the coprocessor application may access any of the buffers defined by the `pOutBuffer` and `pInBuffer` fields at any time, the host application must not modify, reuse, or deallocate any part of the buffers before the call to sccRequest returns.

---

[2] If `pInBuffer[i]` is not aligned on a 4-byte boundary, the device driver must place data written by the coprocessor to `pInBuffer[i]` in an aligned buffer and then copy that buffer to `pInBuffer[i]`.

The effect of changes to the buffers while a request is outstanding is undefined. In particular,

- If the coprocessor application reads a buffer, the host application changes the contents of the buffer, and the coprocessor application reads the buffer again, the value the coprocessor application sees may be the original contents of the buffer, the new contents of the buffer, a mix of old and new contents, or random bytes.
- If the coprocessor application writes a buffer and subsequently writes the buffer a second time, the value written the first time may never be visible to the host application.

Return Codes

Common return codes generated by this routine are:

HDDGood (i.e., 0) No error occurred on a call to the host operating system.

HDDError The operation was unsuccessful.

HOST_OS_ERR An error occurred on a call to the host operating system (the low-order 16 bits contain the return code from the call that failed).

Refer to *scc_err.h* for a comprehensive list of return codes.

A return code of zero does **not** imply that the request was successfully delivered to the coprocessor application, or that the coprocessor application successfully processed the request. A host application should always check both the return code and the `Status` field of the request block to determine whether or not the request was successfully completed.

Common nonzero values placed in the `Status` field by the device driver or the coprocessor operating system are:

HDDInvalidLength The length of a buffer associated with the request is not a multiple of 4.

HDDInvalidParm One or more inputs were not valid.

HDDDeviceBusy Due to the lack of resources, a new request cannot be initiated until a pending request has completed. Try again later.

HDDRequestAborted The request was aborted (for example, because an application on the coprocessor faulted).

HDDBufferTooSmall A buffer from which the coprocessor application attempted to read or to which the coprocessor application attempted to write is not valid or too short.

HDDSecurityTamper The coprocessor's tamper detection mechanisms have been triggered.

HDDError The operation was unsuccessful.

CM_UNDELIVERABLE The identifier in `pRequestBlock->AgentID` does not match the identifier of any registered agent on the coprocessor.

The coprocessor application may also place a nonzero value in the `Status` field; see "sccEndRequest - Return Result of Request to Host" on page 3-15 for details.

sccCloseAdapter - Close Channel to Coprocessor

sccCloseAdapter closes a communications channel that was previously opened via a call to sccOpenAdapter.

Function Prototype

```
long sccCloseAdapter(sccAdapterHandle_t AdapterHandle);
```

Input

On entry to this routine:

`AdapterHandle` uniquely identifies a communications channel to one of the cryptographic coprocessors installed in the host. `AdapterHandle` must contain the handle returned in the `*pAdapterHandle` output from a call to sccOpenAdapter.

Output

On successful exit from this routine:

The communications channel identified by `AdapterHandle` has been closed. The handle should not be subsequently passed as an argument to any host API function.

Return Codes

Common return codes generated by this routine are:

HDDGood (i.e., 0)	The operation was successful.
HDDInvalidParm	One or more inputs were not valid.
HDDError	The operation was unsuccessful.
HOST_OS_ERR	An error occurred on a call to the host operating system (the low-order 16 bits contain the return code from the call that failed).

Refer to *scc_err.h* for a comprehensive list of return codes.

Chapter 3. Coprocessor-Side API

The coprocessor-side portion of the SCC API (coprocessor API) allows an application running on a cryptographic coprocessor to request services from the various device managers running on the coprocessor and to exchange information with an application running on the host in which the cryptographic coprocessor is installed.

Coprocessor API calls can be used to perform various cryptographic operations (including DES and public key encryption and decryption, hashing, general large integer modular functions, and random number generation), manage sensitive data stored in the coprocessor's secure memory, and receive requests from and return results to applications running on the host. A coprocessor application can also make calls directly to the CP/Q base operating system and to the C run-time library. These APIs are described in the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System C Runtime Library Reference*.

This chapter describes each of the functions supplied by the coprocessor API. Each description includes the function prototype (in C), the inputs to the function, the outputs returned by the function, and the most common return codes generated by the function.

General Information

Coprocessor-Side API Functions

The coprocessor API includes functions in the following categories:

Communications Functions

These functions allow a coprocessor application to interact with a host application and obtain permission to request services from the coprocessor device managers:

sccSignOn Register a coprocessor application so that a host application can direct requests to it and so it can request cryptographic and other sensitive services from the coprocessor device managers. See page 3-7.

sccGetNextHeader Get the next request sent from the host to the coprocessor application. See page 3-9.

sccGetBufferData Read data from the host application. See page 3-11.

sccPutBufferData Write data to the host application. See page 3-13.

sccEndRequest Return the result of a request to the host application. See page 3-15.

Hash Functions

These functions allow a coprocessor application to compute a condensed representation of a block of data using various standard hash algorithms:

sccSHA1 Compute the hash of a block of data using the Secure Hash Algorithm (SHA-1) as defined in FIPS Publication 180-1. See page 3-18.

DES Functions

These functions allow a coprocessor application to request services from the Data Encryption Standard (DES) Manager, which uses the coprocessor's DES chip to support DES operations with key lengths of 40, 56, 112, or 168 bits and the Commercial Data Masking Facility (CDMF) algorithm:[1]

sccDES8bytes Encipher or decipher eight bytes of data using the DES algorithim. See page 3-22.

sccDES Encipher or decipher an arbitrary amount of data or generate a message authentication code using the DES algorithm. See page 3-24.

sccDES3Key Triple-encipher (wrap) or triple-decipher (unwrap) a cryptographic key using the DES algorithm. See page 3-28.

sccTDES Encipher or decipher an arbitrary amount of data or generate a message authentication code using the triple-DES algorithm. See page 3-30.

sccEDE3_3DES Perform an EDE3 mode triple-DES operation. See page 3-34.

sccTransformCDMFKey Transform a DES key into a key for use with the CDMF algorithm. See page 3-36.

Public Key Algorithm Functions

These functions allow a coprocessor application to request services from the Public Key Algorithm (PKA) Manager, which uses the coprocessor's large-integer modular math hardware to support public key cryptographic algorithms:

sccRSAKeyGenerate Generate an RSA key pair. See page 3-43.

sccRSA Encipher or decipher a block of data using the RSA algorithm or wrap or unwrap an X9.31 encapsulated hash. See page 3-46.

sccComputeBlindingValues Compute blinding values used to defeat timing-based attacks against an RSA key. See page 3-50.

sccDSAKeyGenerate Generate a DSA key pair. See page 3-54.

sccDSA Sign or verify the signature for an arbitrary amount of data using the DSA algorithm. See page 3-57.

---

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

Large Integer Modular Math Functions

These functions allow a coprocessor application to direct the PKA Manager to perform specific operations on large integers:

sccModMath Perform a modular multiplication (C = A x B mod N), modular exponentiation (C = A^B mod N), or modular reduction (C = A mod N). See page 3-62.

Random Number Generator Functions

These functions allow a coprocessor application to request services from the Random Number Generator (RNG) Manager, which uses a hardware noise source and a pseudo-random number generator to deliver random bits that meet the standards described in FIPS Publication 140-1, section 4.11:

sccGetRandomNumber Generate a 64-bit random number. See page 3-65.

sccTestRandomNumber Ensure the random number generator meets the applicable FIPS standards. See page 3-68.

Nonvolatile Memory Functions

These functions allow a coprocessor application to request services from the Program Proprietary Data (PPD) Manager, which controls the coprocessor's nonvolatile memory areas (flash memory and battery-backed RAM [BBRAM]):

sccQueryPPDSpace Determine the amount of free space in nonvolatile memory. See page 3-72.

sccCreate4UpdatePPD Create space in nonvolatile memory to hold data. See page 3-73.

sccSavePPD Store data in nonvolatile memory. See page 3-75.

sccUpdatePPD Update data in nonvolatile memory. See page 3-78.

sccGetPPDDir Determine the number of items in nonvolatile memory that belong to the caller. See page 3-80.

sccGetPPDLen Determine the length of an item stored in nonvolatile memory. See page 3-82.

sccGetPPD Retrieve data from nonvolatile memory. See page 3-83.

sccDeletePPD Delete data from nonvolatile memory. See page 3-85.

sccDeleteAllPPD Delete all items that belong to the caller from nonvolatile memory. See page 3-87.

Data saved in the coprocessor's nonvolatile memory persists when the coprocessor loses power or is rebooted. Data saved in flash memory persists even when the coprocessor detects an attempt to tamper with the hardware, so sensitive information should be encrypted before it is saved in flash memory. BBRAM is automatically cleared when the coprocessor detects an attempt to tamper with the hardware, so data saved in BBRAM need not be encrypted first.

Coprocessor Configuration Functions

These functions configure certain processor features or return information about the coprocessor:

sccGetConfig Get information about the coprocessor. See page 3-89.

sccSetClock Set the coprocessor time-of-day (TOD) clock. See page 3-92.

sccClearILatch Clear the coprocessor intrusion latch. See page 3-93.

sccClearLowBatt Clear the coprocessor low battery warning latch. See page 3-94.

Outbound Authentication Functions (4758 Model 002/023 only)

Note

These functions are not available on the 4758 Model 001/013

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

sccOAGetDir Count or list all certificates. See page 3-117.

sccOAGetCert Retrieve a certificate. See page 3-119.

sccOAGenerate Generate a keypair and a certificate that contains the public half of the keypair. See page 3-121.

sccOADelete Delete a certificate and the corresponding keypair. See page 3-124.

sccOAPrivOp Perform an operation using one of the keys from a keypair. See page 3-126.

sccOAVerify Verify the signature in one certificate using the public key from another certificate. See page 3-129.

sccOAStatus Obtain information about the contents of the coprocessor. See page 3-131.

Many coprocessor API functions have both a synchronous form (that is, the call does not return until the corresponding function is complete) and an asynchronous form (that is, the call enqueues a message for the appropriate CP/Q++ device manager and returns without waiting for the manager to perform the requested action). See “Synchronous and Asynchronous Calls” on page 1-3 for an overview of asynchronous calls under CP/Q.

Header Files

The prototypes for all coprocessor API functions are contained in *scc_int.h*. Many other header files are used to create coprocessor applications, including *scctypes.h* and *scc_err.h*. These files are included in the IBM 4758 Application Program Development Toolkit. Refer to the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide* for details.

Sample Code

Examples of the use of many of the coprocessor API functions can be found in the following files shipped with the IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer's Toolkit:

Communications Functions

- *scctk\src\samples\oem_samp\OEM_card.c*
- *scctk\src\samples\rte\rte.c*
- *scctk\src\samples\skeleton\scc\skelscc.c*

Hash Functions

- *scctk\src\samples\skeleton\hshserv\hshserv.c*

DES Functions

- *scctk\src\samples\skeleton\desserv\desserv.c*

Public Key Algorithm Functions

- *scctk\src\samples\skeleton\pkaserv\pkaserv.c*

Large Integer Modular Math Functions

- *scctk\src\samples\skeleton\limserv\limserv.c*

Random Number Generator Functions

- *scctk\src\samples\skeleton\rngserv\rngserv.c*

Nonvolatile Memory Functions

- *scctk\src\samples\skeleton\ppdserv\ppdserv.c*

Coprocessor Configuration Functions

- *scctk\src\samples\skeleton\ralserv\ralserv.c*

Serialization of Requests

In general, a single CP/Q++ device manager provides the services for all the functions in a particular category. And each device manager typically serializes all the requests it receives, since there is only a single instance of the hardware the device manager controls. This means that a request for service must wait until all previously received requests have completed. For example, if two applications (or two tasks within the same application) call sccDES, the request that is delivered to the DES Manager last cannot begin until the request that was delivered first has completed. Coprocessor applications should be designed with potential bottlenecks of this nature in mind, although functions that can take several seconds to complete (for example, sccRSAKeyGenerate) do attempt to yield to less demanding requests before starting certain time-consuming operations.

Furthermore, each device manager has a priority. A manager at a less favored (numerically higher) priority will not run if a manager at a more favored (numerically lower) priority is runnable:

Table 3-1. CP/Q++ Device Manager Priority

Device Manager	Priority (the smaller number is the more favored priority)
OA Manager	Priority 3 (if present)
Session Manager	Priority 4* (Priority 3 if OA Manager is not present)
Session Manager	Priority 3*
RNG Manager	Priority 6
Serial port driver	Priority 8*
Communications Manager	Priority 10
DES Manager	Priority 11
SCC Manager	Priorities 12, 16, and 17**
PKA Manager	Priority 13
PPD Manager	Priority 14
Debug probe	Priority 18*

* Present only in the kernel that supports the debugging of applications.
** The SCC Manager process incorporates three tasks, each running a different executable, and each with a different priority.

Note: The listed priorities apply to those parts of the system that are statically built. For example, when the RNG Manager runs it creates a second task which is at priority 5.

Applications created using the Developer's Toolkit are launched at priority 22. Note therefore that an application cannot run if any device manager has work to do.[2] Furthermore, because requests from the host must be processed by an application, a request that uses the services of a more favored manager cannot be serviced if a less favored manager is running. For example, if an application begins a DES operation, a host request that invokes the services of the RNG Manager cannot be serviced by the application until the DES Manager yields the CPU, even though the RNG Manager is more favored than the DES Manager.

Error Codes

Appendix A, "Error Code Formatting" on page A-1 describes the format of a return code. For synchronous functions, the return code generally indicates whether or not the requested operation was successfully completed. For asynchronous functions, the return code simply indicates whether or not the request was successfully sent to the proper CP/Q++ device manager.

Any coprocessor API function may return an error code generated by a CP/Q system call. Refer to the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for more information.

[2] An application may run if the device manager is waiting for the hardware the manager controls to complete an operation.

Communications Functions

The functions described in this section allow a coprocessor application to interact with a host application and obtain permission to request services from the coprocessor device managers.

sccSignOn - Register to Receive Requests

sccSignOn registers a coprocessor application with the SCC Manager so that it can receive requests from the host. Registration is also required to request cryptographic and other sensitive services from the CP/Q++ device managers.

Function Prototype

```
long sccSignOn(sccAgentID_t     *pAgentID,
               unsigned long    *pMsgQID);
```

Input

On entry to this routine:

`pAgentID` must contain the address of an agent identifier structure whose fields are initialized as follows:[3]

- `DeveloperID` uniquely identifies the developer that developed the application. Developer identifiers are assigned by IBM.[4]
- `ProgramID` contains an arbitrary string that identifies the application. The string is not null terminated and should be padded if necessary to occupy the entire field. The developer identified by `DeveloperID` may load several different applications into the coprocessor.[5] Each application must have a distinct `ProgramID`.
- Several versions (or releases) of an application may be loaded into the coprocessor.[5] Different versions are distinguished by the values in the `Version` field of their respective agent identifiers.
- CP/Q is a multitasking operating system and consequently several identical copies of an application may be loaded into the coprocessor.[5] Different instantiations are distinguished by the values in the `Instance` field of their respective agent identifiers.
- An application may define several logical message queues on which it will receive requests.[6] Different logical message queues are distinguished by the values in the `Queue` field of their respective agent identifiers.

  Note that each logical message queue may map to a distinct CP/Q message queue, or the traffic for several logical message queues may be multiplexed onto a single CP/Q message queue. See the following description of the `pMsgQID` argument.

---

3 See “Interpretation of Agent Identifier” on page 3-8.

4 A developer identifier is typically the same as the owner identifier for segment 3. Refer to the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer's Toolkit Guide* for more information.

5 Although it is possible to download several applications to the coprocessor, only one is actually run at present.

6 For example, the application might define one queue for urgent requests and another for normal requests and always service requests on the former first.

pMsgQID determines the CP/Q message queue on which requests from the host addressed to the coprocessor application are placed:[7]

- If *pMsgQID is NULL, requests from the host will be placed on the default message queue for the task that invoked sccSignOn.
- If pMsgQID is not NULL,
  - If *pMsgQID is nonzero, it is the identifier of the message queue[8] on which requests from the host should be placed.
  - If *pMsgQID is zero, sccSignOn creates a new message queue on which requests from the host will be placed.

See "Synchronous and Asynchronous Calls" on page 1-3 for a brief description of messaging and message queues in CP/Q.

Output

On successful exit from this routine:

If pMsgQID is not NULL, *pMsgQID identifies the CP/Q message queue on which requests from the host addressed to the coprocessor application will be placed. This message queue identifier must be passed to sccGetNextHeader when the coprocessor application wants to receive a request from the host.

If pMsgQID is NULL, requests from the host will be placed on the default CP/Q message queue for the task that invoked sccSignOn. A message queue identifier of zero must be passed to sccGetNextHeader when the coprocessor application wants to receive a request from the host.

Notes

Interpretation of Agent Identifier

Although the fields of a sccAgentID_t structure are intended to be used in the manner described in "Input" above, the SCC Manager does not enforce this particular interpretation. It simply establishes a mapping from agent identifier to CP/Q message queue. When a request is received from the host, the SCC Manager compares the agent identifier in the request with each registered agent identifier, byte-for-byte, and forwards the request to the appropriate CP/Q message queue when an exact match is found.

Return Codes

Common return codes generated by this routine are:

SCCGood (i.e., 0) The operation was successful.

SCCBadParm Another coprocessor application has already registered using the same agent identifier structure (*pAgentID).

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

---

7 A single CP/Q message queue may be associated with several different agent identifiers.

8 For example, a message queue created by the coprocessor application via a call to CPCreateMsgQ. Refer to the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for details.

sccGetNextHeader - Get Next Request from Host

sccGetNextHeader retrieves the next request header from the host from a CP/Q message queue.

Function Prototype

```
long sccGetNextHeader(sccRequestHeader_t   *pRequestHeader,
                      unsigned long         msgQID
                      unsigned long         timeout);
```

Input

On entry to this routine:

`pRequestHeader` must contain the address of a variable in which an item of type `sccRequestHeader_t` can be stored.

`msgQID` must contain the identifier of the CP/Q message queue on which the SCC manager was instructed to place requests addressed to the calling application when the application invoked sccSignOn. sccGetNextHeader retrieves a request from the message queue identified by `msgQID`.

If `msgQID` is zero, sccGetNextHeader retrieves a request from the default message queue for the calling task.

`timeout` specifies the number of microseconds to wait for a request if none is available immediately. If `timeout` is zero, sccGetNextHeader returns immediately even if no request is available. If `timeout` is SVCWAITFOREVER, sccGetNextHeader does not return until a request is received.

Output

On successful exit from this routine:

`*pRequestHeader` contains a request removed from the head of the CP/Q message queue specified by msgQID on entry to the routine. The fields of `*pRequestHeader` are set as follows:

- `AgentID` contains the agent identifier of the coprocessor application to which the request was sent as specified by the host application in the call to sccRequest that caused the request to be sent.[9] See “sccSignOn - Register to Receive Requests” on page 3-7 for details.
- `RequestID` contains a handle generated by CP/Q++ that uniquely identifies the request. This handle must be passed to sccGetBufferData when the coprocessor application wants to read data from the host, to sccPutBufferData when the coprocessor application wants to write data to the host, and to sccEndRequest when the coprocessor application wants to return the result of a request to the host.
- `rsvd` is undefined.

[9] The agent identifier in the request logically must match the agent identifier registered with the SCC Manager that is associated with the queue identified by `msgQID` (the request would not have been delivered otherwise). However, the task that retrieves a request with sccGetNextHeader need not be the task that registered the agent identifier with sccSignOn. The task that retrieves the request need only have authority to read the CP/Q message queue identified by `msgQID`. Refer to the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for details.

- `UserDefined`, `OutBufferLength`, and `InBufferLength` contain the values of the corresponding fields in the request block passed by the host application in the call to sccRequest that caused the request to be sent. In general, `UserDefined` identifies the action the coprocessor application is to perform on behalf of the host application, `OutBufferLength` contains the lengths of the host application buffers from which the coprocessor application may read data, and `InBufferLength` contains the lengths of the host application buffers to which the coprocessor application may write data.

  A coprocessor application that provides sensitive services or guards sensitive data should not assume a request was issued by a legitimate host application.

Return Codes

Common return codes generated by this routine are:

SCCGood (i.e., 0)	The operation was successful.
QSVCtimedout	The timeout expired before a request was received from the host.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccGetBufferData - Read Data from Host

sccGetBufferData reads data from a buffer supplied by the host application as part of a request.

Function Prototype

```
long sccGetBufferDataAsync(sccRequestID_t  requestID,
                           sccBufferID_t   bufIdx,
                           void           *pBuffer,
                           unsigned long   buflen,
                           unsigned long  *pMsgID);

#define sccGetBufferData(r,bi,pb,bl)  sccGetBufferDataAsync(r,bi,pb,bl,NULL)
```

Input

On entry to this routine:

`requestID` is the handle of the request with which the buffer to be read is associated. `requestID` must contain the handle returned in the `pRequestHeader->RequestID` output from a call to sccGetNextHeader.

A host application may associate as many as four readable buffers with a single request. `bufIdx` is the index of the buffer the coprocessor application wants to read and must be greater than or equal to zero and less than four.

`pBuffer` must contain the address of a buffer to which the contents of the buffer supplied by the host application may be copied. The buffer referenced by `pBuffer` must be aligned on a 4-byte boundary. The first byte of the buffer supplied by the host application is always the first byte copied, and it is always placed in the first byte of the buffer referenced by `pBuffer`.

`bufLen` is the length of the buffer referenced by `pBuffer`. `bufLen` must be a multiple of four equal to the value returned in the `pRequestHeader->OutBufferLength[bufIdx]` output from a call to sccGetNextHeader.

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until the contents of the buffer supplied by the host application have been completely copied into the buffer referenced by `pBuffer`.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the Communications Manager instructing it to transfer data from the host.

Output

On successful exit from this routine:

If `pMsgID` is NULL, the buffer referenced by `pBuffer` contains a copy of the contents of the appropriate host application buffer.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the Communications Manager to initiate the transfer of data from the host. When the transfer is complete, the Communications Manager will send the coprocessor

application a message whose type field (`MSG.h.msg_type`) contains this identifier and whose first (and only) data item (`MSG.msg_data[0]`) contains the return code generated by the routine.[10] The message is placed on the default CP/Q message queue for the task that called sccGetBufferDataAsync. Refer to the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for details.

Notes

Use of Host Application Buffers

The coprocessor application may use the buffers provided by the host application in any manner agreed to by the two applications. The SCC API does not interpret the contents of the buffers nor does it require that the buffers be read in a specific order or limit the number of times the contents of a buffer may be read.

Changes to Buffers while Request Outstanding

The effect of changes to the buffers while a request is outstanding is undefined. In particular, if the coprocessor application reads a buffer, the host application changes the contents of the buffer, and the coprocessor application reads the buffer again, the value the coprocessor application sees may be the original contents of the buffer, the new contents of the buffer, a mix of old and new contents, or random bytes.

Return Codes

Common return codes generated by this routine are:

SCCGood (i.e., 0)	The operation was successful.
CM_INVALID_REQUEST_ID	The `requestID` argument is not valid or host application ended.
CM_INVALID_BUFFER_ID	The buffer referenced by `pBuffer` is not aligned on a 4-byte boundary.
CM_INVALID_ADDRESS	The buffer referenced by `pBuffer` is not writeable.
CM_INVALID_LENGTH	The `bufLen` argument is not a multiple of four or is not equal to the length of the buffer provided by the host application.
CM_REQUEST_ABORTED	The request was aborted because the host application ended.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

[10] The return code from the call to sccGetBufferDataAsync indicates whether or not the initial message to the Communications Manager was successfully enqueued.

sccPutBufferData - Write Data to Host

sccPutBufferData writes data to a buffer supplied by the host application as part of a request.

Function Prototype

```
long sccPutBufferDataAsync(sccRequestID_t  requestID,
                           sccBufferID_t   bufIdx,
                           void           *pBuffer,
                           unsigned long   buflen,
                           unsigned long  *pMsgID);

#define sccPutBufferData(r,bi,pb,bl) sccPutBufferDataAsync(r,bi,pb,bl,NULL)
```

Input

On entry to this routine:

`requestID` is the handle of the request with which the buffer to be written is associated. `requestID` must contain the handle returned in the `pRequestHeader->RequestID` output from a call to sccGetNextHeader.

A host application may associate as many as four writeable buffers with a single request. `bufIdx` is the index of the buffer the coprocessor application wants to write and must be greater than or equal to zero and less than four.

`pBuffer` must contain the address of a buffer whose contents will be copied to the buffer supplied by the host application. The buffer referenced by `pBuffer` must be aligned on a 4-byte boundary. The first byte of the buffer referenced by `pBuffer` is always the first byte copied and it is always placed in the first byte of the buffer supplied by the host application.

`bufLen` is the length of the buffer referenced by `pBuffer`. `bufLen` must be a multiple of four less than or equal to the value returned in the `pRequestHeader->InBufferLength[bufIdx]` output from a call to sccGetNextHeader.

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until the contents of the buffer referenced by `pBuffer` have been completely copied into the buffer supplied by the host application.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the Communications Manager instructing it to transfer data to the host. In this case, the coprocessor application must not modify, deallocate, or reuse any portion of the buffer referenced by `pBuffer` before the transfer is complete.

Output

On successful exit from this routine:

If `pMsgID` is NULL, the appropriate host application buffer contains a copy of the contents of the buffer referenced by `pBuffer`.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the Communications Manager to initiate the transfer of data to the host. When the transfer is complete, the Communications Manager will send the coprocessor application a message whose type field (`MSG.h.msg_type`) contains this identifier and whose first (and only) data item (`MSG.msg_data[0]`) contains the return code generated by the routine.[11] The message is placed on the default CP/Q message queue for the task that called sccPutBufferDataAsync.

Notes

Use of Host Application Buffers

The coprocessor application may use the buffers provided by the host application in any manner agreed to by the two applications. The SCC API does not interpret the contents of the buffers nor does it require that the buffers be written in a specific order or limit the number of times the contents of a buffer may be written.

Changes to Buffers while Request Outstanding

The effect of changes to the buffers while a request is outstanding is undefined. In particular, if the coprocessor application writes a buffer and subsequently writes the buffer a second time, the value written the first time may never be visible to the host application.

Return Codes

Common return codes generated by this routine are:

SCCGood (i.e., 0)	The operation was successful.
CM_INVALID_REQUEST_ID	The `requestID` argument is not valid or host application ended.
CM_INVALID_BUFFER_ID	The buffer referenced by `pBuffer` is not aligned on a 4-byte boundary.
CM_INVALID_ADDRESS	The buffer referenced by `pBuffer` is not readable.
CM_INVALID_LENGTH	The `bufLen` argument is not a multiple of four or is greater than the length of the buffer provided by the host application.
CM_REQUEST_ABORTED	The request was aborted because the host application ended.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

[11] The return code from the call to sccPutBufferDataAsync indicates whether or not the initial message to the Communications Manager was successfully enqueued.

sccEndRequest - Return Result of Request to Host

sccEndRequest ends a request and returns a status code to the host application indicating whether or not the request was successful. sccEndRequest can also write data to a buffer supplied by the host application as part of the request.

Function Prototype

```
long sccEndRequest(sccRequestID_t  requestID,
                   sccBufferID_t   bufIdx,
                   void           *pBuffer,
                   unsigned long   buflen,
                   long            status);
```

Input

On entry to this routine:

`requestID` is the handle of the request that has been completed. `requestID` must contain the handle returned in the `pRequestHeader->RequestID` output from a call to sccGetNextHeader.

sccEndRequest can optionally write data to a buffer supplied by the host application in the same manner as sccPutBufferData. If it does, `bufIdx` is the index of the buffer the coprocessor application wants to write and must be greater than or equal to zero and less than four.

`pBuffer` may contain the address of a buffer whose contents will be copied to the buffer supplied by the host application. If the buffer referenced by `pBuffer` is used (see `bufLen` below), it must be aligned on a 4-byte boundary. The first byte of the buffer referenced by `pBuffer` is always the first byte copied and it is always placed in the first byte of the buffer supplied by the host application.

`bufLen` is the length of the buffer referenced by `pBuffer`. `bufLen` must be a multiple of four less than or equal to the value returned in the `pRequestHeader->InBufferLength[bufIdx]` output from a call to sccGetNextHeader. If `bufLen` is zero, sccEndRequest does not write data to a host application buffer and the `bufIdx` and `pBuffer` arguments are not used.

`status` contains an arbitrary value that is returned to the host application in the `pRequestBlock->Status` output from a call to sccRequest. This is typically used as a return code from the request indicating whether or not it was successful. See “sccRequest - Send Request to Coprocessor Application” on page 2-8 for details.

Output

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

The value in the `status` argument and any data written to the host (if `bufLen` is nonzero) have been sent to the host application.

The handle in `requestID` is no longer valid.

Return Codes

Common return codes generated by this routine are:

SCCGood (i.e., 0)	The operation was successful.
CM_INVALID_REQUEST_ID	The `requestID` argument is not valid or host application ended.
CM_INVALID_BUFFER_ID	The buffer referenced by `pBuffer` is not aligned on a 4-byte boundary.
CM_INVALID_ADDRESS	The buffer referenced by `pBuffer` is not readable.
CM_INVALID_LENGTH	The `bufLen` argument is not a multiple of four or is greater than the length of the buffer provided by the host application.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference*, CPQR-2A01 for a comprehensive list of return codes.

Hash Functions

The functions described in this section allow a coprocessor application to compute a condensed representation of a block of data using various standard hash algorithms.

Internal and External Buffers

The buffer from which hash functions read the block of data to hash may be located on the cryptographic coprocessor in a coprocessor application's address space. Such buffers are called *internal*. In addition, on the 4758 model 002 certain hash operations (for example, SHA-1) can read their input directly from a buffer located on the host in a host application's address space. Such buffers are called *external*.[12]

An internal buffer is described by a structure of type `sccInternalBuffer_t`. The fields of this structure are:

- `count`, which specifies the length in bytes of the block of data to hash. `count` must be less than 32M. Additional constraints may be imposed by particular hash algorithms.
- `buffer`, which is the address of the first byte in the buffer. Alignment constraints may be imposed by particular hash algorithms.

An external buffer is described by a structure of type `sccExternalBuffer_t`. The fields of this structure are:

- `count`, which specifies the length in bytes of the block of data to hash. `count` must be a multiple of 4 less than 32M. Additional constraints may be imposed by particular hash algorithms.
- `request_id`, which is the handle of the host request with which the buffer is associated. `request_id` must contain the handle returned in the `pRequestHeader->RequestID` output from a call to sccGetNextHeader.
- `buffer_id`. A host application may associate with a single request as many as four readable buffers. `buffer_id` is the index of the buffer to be read and must be greater than or equal to zero and less than four.

The hash functions read host buffers in the same way as sccGetBufferData.

[12] On the 4758 model 002, DES operations can also read input from or write results to an external buffer. The SHA-1 and DES hardware share a single path to the host, so if a coprocessor application requests a SHA-1 operation that uses an external buffer while a DES operation that also uses an external buffer is in progress, the SHA-1 operation waits until the DES operation is complete.

sccSHA1 - SHA-1 Hash

sccSHA1 computes the hash of a block of data using the Secure Hash Algorithm (SHA-1).

Function Prototype

```
long sccSHA1Async(sccSHA_RB_t     *pSHARB,
                  unsigned long   *pMsgID);

#define sccSHA1(p)  sccSHA1Async(p,NULL)
```

Input

On entry to this routine:

`pSHARB` must contain the address of a SHA-1 operation request block whose fields are initialized as follows:

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

  Operating Mode

  `options` must include one of the following constants:

SHA_MSGPART_ONLY	The input data constitutes the entire block of data to be hashed. The hash value is computed and returned.
SHA_MSGPART_FIRST	The input data constitutes the first portion of a block of data to be hashed. See “Chained Operations” on page 3-20 for details.
SHA_MSGPART_MIDDLE	The input data constitutes an additional portion of a block of data to be hashed. See “Chained Operations” on page 3-20 for details.
SHA_MSGPART_FINAL	The input data constitutes the final portion of a block of data to be hashed. See “Chained Operations” on page 3-20 for details.

  Source of Input

  `options` must include one of the following constants:

SHA_INTERNAL_INPUT	Read input data from an internal buffer.
SHA_EXTERNAL_INPUT	Read input data from an external buffer.

  See “Internal and External Buffers” on page 3-17 for details.

  `options` must specify SHA_INTERNAL_INPUT on the 4758 model 001.

- `source` describes the location of the buffer containing the input data. If `options` specifies SHA_INTERNAL_INPUT, `source.internal` defines an internal buffer that contains the input. If `options` specifies SHA_EXTERNAL_INPUT, `source.external` defines an external buffer that contains the input. See “Internal and External Buffers” on page 3-17 for details.

  If `options` specifies SHA_MSGPART_FIRST or SHA_MSGPART_MIDDLE, `source.internal.count` or `source.external.count`, as appropriate, must be a multiple of 64.

- final_data may hold the last few bytes of the input data. If options specifies SHA_MSGPART_ONLY or SHA_MSGPART_FINAL and the length of the input data (that is, the count field in the buffer descriptor contained in source) is not a multiple of 4, the last count mod 4 bytes of input are not read from the buffer described by source but are instead taken from final_data (starting with final_data[0]).

  final_data is not used if options does not specify SHA_MSGPART_ONLY or SHA_MSGPART_FINAL or if the length of the input data is a multiple of 4.

- hash_value contains the hash value computed for that portion of the block of data to hash that has been processed by prior calls to sccSHA1.[13] hash_value is used only if options specifies SHA_MSGPART_MIDDLE or SHA_MSGPART_FINAL.

- running_length contains the number of bytes of the block of data to hash that have been processed by prior calls to sccSHA1. running_length must be 0 if options specifies SHA_MSGPART_FIRST or SHA_MSGPART_ONLY.

pMsgID determines whether the function is performed synchronously or asynchronously:

- If pMsgID is NULL, the function is performed synchronously. The call does not return until the requested hash operation is complete.
- If pMsgID is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the DES Manager[14] instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the request block referenced by pSHARB or the buffer described by pSHARB->source before the operation is complete.

The asynchronous form of this call is not available on the 4758 model 001.

Output

On successful exit from this routine:

If pMsgID is NULL, pSHARB->hash_value contains the hash value of the input data. If options specifies SHA_MSGPART_MIDDLE or SHA_MSGPART_FINAL, this incorporates the value of pSHARB->hash_value on entry to the routine.

pSHARB->running_length reflects the number of bytes of input that were hashed. If options specifies SHA_MSGPART_ONLY or SHA_MSGPART_FIRST, this is the number of bytes of input that were hashed. If options specifies SHA_MSGPART_MIDDLE or SHA_MSGPART_FINAL, this is the value of pSHARB->running_length on entry to the routine increased by the number of bytes of input that were hashed.

If pMsgID is not NULL, *pMsgID uniquely identifies the message that was sent to the DES Manager to initiate the desired operation. When the operation is complete, the DES Manager sends the coprocessor application a message whose type field (Msg.h.msg_type) contains this identifier and whose first (and only) data item (Msg.msg_data[0]) contains the return code generated by the routine. If the

[13] That is, hash_value specifies the initial values of the H_i used in the SHA-1 algorithm.

[14] The DES Manager handles SHA-1 hashing.

operation was successful, pSHARB->hash_value and pSHARB->running_length contain the results as previously described. The message is placed on the default CP/Q message queue for the task that called sccSHA1Async.

Notes

Chained Operations

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

A chained operation is initiated by calling sccSHA1 with SHA_MSGPART_FIRST specified in pSHARB->options and the first piece of the block of data to hash identified in pSHARB->source. On return, pSHARB->hash_value contains the hash for the first piece of data and pSHARB->running_length contains the number of bytes of data processed. These values must be preserved and passed to sccSHA1 when the next piece of the block of data to hash is processed.

Subsequent pieces of the block are processed by calling sccSHA1 with SHA_MSGPART_MIDDLE specified in pSHARB->options (SHA_MSGPART_FINAL must be specified if the piece in question is the last) and the location of the piece identified in pSHARB->source. pSHARB->hash_value and pSHARB->running_length must contain the values returned in those fields by the call to sccSHA1 that processed the previous piece of the block. The function hashes the piece and updates pSHARB->hash_value and pSHARB->running_length appropriately.

Return Codes

Common return codes generated by this routine are:

SHA1Good (i.e., 0)	The operation was successful.
SHA1_DATA64_ERROR	The options argument specified SHA_MSGPART_FIRST or SHA_MSGPART_MIDDLE but the length of the data to process is not a multiple of 64.
SHA1_DATA32MB_ERROR	The length of the data to process is 32M or greater.
SHA1_FINAL_ERROR	An error occurred while attempting to pad the input data as dicated by the SHA-1 algorithm or while attempted to hash the pad bytes.
SHA1_EXTERNAL_NOT_SUPPORTED	The options argument specified SHA_EXTERNAL_INPUT but the coprocessor hardware does not support external buffers.

DES Functions

The functions described in this section allow a coprocessor application to request services from the DES Manager, which performs various encryption and decryption operations using the coprocessor's DES hardware.

Keys

The interface to the DES Manager defines the `sccDES_Key_t` type to hold DES and CDMF keys. Both DES and CDMF keys are 56 bits long (although CDMF keys have an effective strength of 40 bits). An item of type `sccDES_Key_t` is eight bytes long. The high-order seven bits of each byte are key bits. The low-order bit of each byte is a parity bit (which the DES Manager ignores[15]).

Internal and External Buffers

DES operations that process large blocks of data read the data to encrypt or decrypt from one buffer and write the resulting ciphertext or plaintext to another buffer. Either buffer (or both) may be located on the cryptographic coprocessor in a coprocessor application's address space. Such buffers are called *internal*. In addition, the DES Manager can be directed to read data directly from or write data directly to a buffer located on the host in a host application's address space. Such buffers are called *external*.[16]

An internal buffer is described by a structure of type `sccInternalBuffer_t`. The fields of this structure are:

- `count`, which specifies the length in bytes of the data to encrypt or of the buffer that is to hold the encrypted result. `count` must be a multiple of 8 less than 32M.
- `buffer`, which is the address of the first byte of the buffer. The buffer must be aligned on a 4-byte boundary.

An external buffer is described by a structure of type `sccExternalBuffer_t`. The fields of this structure are:

- `count`, which specifies the length in bytes of the data to encrypt or of the buffer that is to hold the encrypted result. `count` must be a multiple of 8 less than 32M. For an output (result) buffer, `count` must match the length of the buffer provided by the host application.
- `request_id`, which is the handle of the host request with which the buffer is associated. `request_id` must contain the handle returned in the `pRequestHeader->RequestID` output from a call to sccGetNextHeader.
- `buffer_id`. A host application may associate with a single request as many as four readable buffers and four writeable buffers. `buffer_id` is the index of the buffer to be read or written and must be greater than or equal to zero and less than four.

The DES Manager reads host buffers in the same way as sccGetBufferData and writes host buffers in the same way as sccPutBufferData.

---

[15] That is, the DES Manager does not check the parity of keys.

[16] On the 4758 model 002, SHA-1 operations can also read their input from an external buffer. The DES and SHA-1 hardware share a single path to the host, so if a coprocessor application requests a DES operation that uses one or two external buffers while a SHA-1 operation that also uses an external buffer is in progress, the DES operation waits until the SHA-1 operation is complete.

sccDES8bytes - Encipher/Decipher Eight Bytes of Data

sccDES8bytes enciphers and deciphers eight bytes of data using a DES or CDMF key. The operation is performed in CBC mode using zero for the initial vector.

Function Prototype

```
long sccDES8bytesAsync(sccDES8bytes_RB_t     *pDES8RB,
                       unsigned long         *pMsgID);

#define sccDES8bytes(p)  sccDES8bytesAsync(p,NULL)
```

Input

On entry to this routine:

`pDES8RB` must contain the address of a DES 8-byte operation request block structure whose fields are initialized as follows:

- `options` determines whether an encryption or a decryption is performed. `options` must be either DES_ENCRYPT (encryption) or DES_DECRYPT (decryption).
- `key` is the encryption/decryption key. `key` may be a regular DES key or a CDMF key (for example, the value returned in the `key_out` output from a call to sccTransformCDMFKey).
- `input_data` contains eight bytes of data to encrypt or to decrypt.

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until the encryption or decryption is complete.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the DES Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the request block referenced by `pDES8RB` before the operation is complete.

Output

On successful exit from this routine:

If `pMsgID` is NULL, `pDES8RB->output_data` contains

- the contents of `pDES8RB->input_data` encrypted with `pDES8RB->key` if `options` is DES_ENCRYPT and
- the contents of `pDES8RB->input_data` decrypted with `pDES8RB->key` if `options` is DES_DECRYPT.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the DES Manager to initiate the desired operation. When the operation is complete, the DES Manager will send the coprocessor application a message whose type field (`MSG.h.msg_type`) contains this identifier and whose first (and only) data item (`MSG.msg_data[0]`) contains the return code generated by the routine.[17] If the

[17] The return code from the call to sccDES8bytesAsync indicates whether or not the initial message to the DES Manager was successfully enqueued.

operation was successful, `pDES8RB->output_data` contains the result. The message is placed on the default CP/Q message queue for the task that called sccDES8bytesAsync.

Return Codes

Common return codes generated by this routine are:

DMGood (i.e., 0) The operation was successful.

DMNotAuth The coprocessor application is not authorized to perform DES operations (for example, because it has not called sccSignOn).

DMBadFlags The `options` argument is not valid.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccDES - Encipher/Decipher Data or Generate MAC

sccDES enciphers or deciphers an arbitrary amount of data using a DES or CDMF key. Both CBC and ECB modes are supported. sccDES can also generate a message authentication code (MAC).

Function Prototype

```
long sccDESAsync(sccDES_RB_t     *pDESRB,
                 unsigned long   *pMsgID);

#define sccDES(p)  sccDESAsync(p,NULL)
```

Input

On entry to this routine:

`pDESRB` must contain the address of a DES operation request block structure whose fields are initialized as follows:

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

  Requested Function

  `options` must include one of the following constants:

DES_ENCRYPT	Encrypt the input.
DES_DECRYPT	Decrypt the input.
DES_MAC	Generate a message authentication code for the input.

  If DES_MAC is specified, DES_USE_KEY must also specified and DES_CBC_MODE is forced.

  Key Transformation

  `options` must include one of the following constants:

  DES_TRANSFORM_KEY
  : Perform the requested operation using a CDMF key derived from the DES key passed in the `key` field. (The `key` field is not changed.)

  DES_USE_KEY
  : Perform the requested operation using the DES or CDMF key passed in the `key` field.

  If `options` specifies DES_MAC, DES_USE_KEY must also be specified. If DES_TRANSFORM_KEY is specified, the `key` field should not already contain a CDMF key.

  DES Mode

  `options` must include one of the following constants:

DES_CBC_MODE	Use CBC mode.
DES_ECB_MODE	Use ECB mode.

  DES_CBC_MODE is forced if `options` specifies DES_MAC. In this case, DES_ECB_MODE must not be specified. DES_CBC_MODE may be specified but need not be.

Source of Input

`options` must include one of the following constants:

DES_INTERNAL_INPUT
: Read input data from an internal buffer.

DES_EXTERNAL_INPUT
: Read input data from an external buffer.

See “Internal and External Buffers” on page 3-21 for details.

Destination of Output

`options` must include one of the following constants:

DES_INTERNAL_OUTPUT
: Write output data to an internal buffer.

DES_EXTERNAL_OUTPUT
: Write output data to an external buffer.

See “Internal and External Buffers” on page 3-21 for details.

If `options` specifies DES_MAC, the “Destination of Output” options are ignored.

Padding Options

`options` may include the following constants:

DES_PREPAD	Prepad the input with eight bytes of data.
DES_PAD_WITH_8	Pad the input with eight bytes of data.
DES_PAD_WITH_16	Pad the input with sixteen bytes of data.

DES_PREPAD is ignored unless `options` specifies DES_MAC. Either DES_PAD_WITH_8 or DES_PAD_WITH_16 may be specified, but not both.

- `key` is the encryption/decryption key. If `key` is a DES key, `options` should specify DES_USE_KEY to encrypt/decrypt using the DES algorithm and should specify DES_TRANSFORM_KEY to encrypt/decrypt using the CDMF algorithm. If `key` is a CDMF key (for example, the value returned in the `key_out` output from a call to sccTransformCDMFKey), `options` should specify DES_USE_KEY.
- `init_v` is the initialization vector for the operation. `init_v` is not used if `options` specifies DES_ECB_MODE.
- `source` describes the location of the buffer containing the input data. If `options` specifies DES_INTERNAL_INPUT, `source.internal` defines an internal buffer that contains the input. If `options` specifies DES_EXTERNAL_INPUT, `source.external` defines an external buffer that contains the input. See “Internal and External Buffers” on page 3-21 for details.
- `destination` describes the location of the buffer to which the output is to be written. If `options` specifies DES_INTERNAL_OUTPUT, `destination.internal` defines an internal buffer to which to write the output. If `options` specifies DES_EXTERNAL_OUTPUT, `destination.external` defines an external buffer to which to write the output. See “Internal and External Buffers” on page 3-21 for details.

  If `options` specifies DES_MAC, `destination` is not used.
- `prePadding` contains eight bytes of data to which the input (including any padding) is appended before the requested cryptographic operation is performed. `prePadding` is not used unless `options` specifies DES_MAC.

- postPadding contains data that is appended to the input (including any prepadding) before the requested cryptographic operation is performed. If options specifies DES_PAD_WITH_8, postPadding[0] through postPadding[7] are appended to the input. If options specifies DES_PAD_WITH_16, the entire postPadding field is appended to the input. If options specifies neither DES_PAD_WITH_8 nor DES_PAD_WITH_16, postPadding is not used.

pMsgID determines whether the function is performed synchronously or asynchronously:

- If pMsgID is NULL, the function is performed synchronously. The call does not return until the requested cryptographic operation is complete.
- If pMsgID is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the DES Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the request block referenced by pDESRB or the buffers described by pDESRB->source and pDESRB->destination before the operation is complete.

Output

On successful exit from this routine:

If pMsgID is NULL, pDESRB->term_v contains

- the message authentication code for the input if options specifies DES_MAC,
- the value to use for the initialization vector in the next call to sccDESAsync if options specifies DES_CBC_MODE, and
- garbage if options specifies DES_ECB_MODE.

The buffer described by pDESRB->destination contains

- the contents of the buffer described by pDESRB->source encrypted with pDESRB->key if options specifies DES_ENCRYPT and
- the contents of the buffer described by pDESRB->source decrypted with pDESRB->key if options specifies DES_DECRYPT.

If pMsgID is not NULL, *pMsgID uniquely identifies the message that was sent to the DES Manager to initiate the desired operation. When the operation is complete, the DES Manager will send the coprocessor application a message whose type field (MSG.h.msg_type) contains this identifier and whose first (and only) data item (MSG.msg_data[0]) contains the return code generated by the routine.[18] If the operation was successful, pDES8RB->term_v and pDESRB->destination contain the results as previously described. The message is placed on the default CP/Q message queue for the task that called sccDESAsync.

Notes

Notes on Source and Destination Buffers

The length of the input data (including the length of the input buffer described by pDESRB->source and any pad bytes) may be less than the length of the output

[18] The return code from the call to sccDESAsync indicates whether or not the initial message to the DES Manager was successfully enqueued.

buffer described by `pDESRB->destination`. In this case, any excess bytes at the end of the output buffer are not affected by sccDESAsync.

If `destination` is used, the buffers described by `source` and `destination` should either be the same buffer or not overlap at all.

Return Codes

Common return codes generated by this routine are:

DMGood (i.e., 0) The operation was successful.

DMNotAuth The coprocessor application is not authorized to perform DES operations (for example, because it has not called sccSignOn).

DMBadFlags The `options` argument is not valid.

DMBadParm The length of the input data or the output data is invalid (for example, not a multiple of 8) or the length of the input data exceeds the length of the output buffer, or an internal buffer is not aligned on a 4-byte boundary.

DMBadAddr The input buffer is an internal buffer and is not readable or the output buffer is an internal buffer and is not writeable.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccDES3Key - Wrap/Unwrap Cryptographic Key

sccDES3Key performs three successive encryptions or decryptions (or both encryptions and decryptions) on eight bytes of data using a distinct DES or CDMF key on each pass. Each operation is performed in ECB mode.

sccDES3Key is typically used to triple-encipher (wrap) or triple-decipher (unwrap) a cryptographic key as part of key management. The interface to sccDES3Key is completely flexible, however, and so permits encryption and decryption of eight bytes of data with an effective key length of up to 168 bits.

Function Prototype

```
long sccDES3KeyAsync(sccDES3Key_RB_t     *pDES3RB,
                     unsigned long       *pMsgID);

#define sccDES3Key(p)  sccDES3KeyAsync(p,NULL)
```

Input

On entry to this routine:

`pDES3RB` must contain the address of a DES triple operation request block structure whose fields are initialized as follows:

- `options` determines the nature of each of the three operations the function performs. `options` must be set to the logical OR of one of each of the following pairs of constants:
  - DES3_1_ENCRYPT or DES3_1_DECRYPT - Encrypt input on first pass or decrypt input on first pass, respectively.
  - DES3_2_ENCRYPT or DES3_2_DECRYPT - Encrypt input on second pass or decrypt input on second pass, respectively.
  - DES3_3_ENCRYPT or DES3_3_DECRYPT - Encrypt input on third pass or decrypt input on third pass, respectively.
- `key_in` contains eight bytes of input data.
- `key1`, `key2`, and `key3` are the encryption/decryption keys for pass 1, pass 2, and pass 3, respectively. Each key may be a regular DES key or a CDMF key (for example, the value returned in the `key_out` output from a call to sccTransformCDMFKey).

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until all three encryptions/decryptions are complete.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the DES Manager instructing it to perform the desired operations. In this case, the caller must not modify, deallocate, or reuse any portion of the request block referenced by `pDES3RB` before the operations are complete.

Output

On successful exit from this routine:

If `pMsgID` is NULL, `pDES3RB->key_out` contains the contents of `pDES3RB->key_in` encrypted or decrypted with `pDES3RB->key1`, `pDES3RB->key2`, and `pDES3RB->key3` as dictated by the value of `options`.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the DES Manager to initiate the desired operation. When the operation is complete, the DES Manager will send the coprocessor application a message whose type field (`MSG.h.msg_type`) contains this identifier and whose first (and only) data item (`MSG.msg_data[0]`) contains the return code generated by the routine.[19] If the operation was successful, `pDES3RB->key_out` contains the result. The message is placed on the default CP/Q message queue for the task that called sccDES3KeyAsync.

Return Codes

Common return codes generated by this routine are:

DMGood (i.e., 0)	The operation was successful.
DMNotAuth	The coprocessor application is not authorized to perform DES operations (for example, because it has not called sccSignOn).
DMBadFlags	The `options` argument is not valid.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

[19] The return code from the call to sccDES3KeyAsync indicates whether or not the initial message to the DES Manager was successfully enqueued.

sccTDES - Triple DES (4758 Model 002 only)

Note

This function is not available on the 4758 model 001.

sccTDES enciphers or deciphers an arbitrary amount of data using triple-DES with three DES keys. Both outer CBC and ECB modes are supported. sccTDES can also generate a message authentication code (MAC).

Function Prototype

```
long sccTDESAsync(sccTDES_RB_t     *pTDESRB,
                  unsigned long    *pMsgID);

#define sccTDES(p)  sccTDESAsync(p,NULL)
```

Input

On entry to this routine:

`pTDESRB` must contain the address of a triple-DES operation request block structure whose fields are initialized as follows:

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

  TDES Selection

  `options` must include DES_TRIPLE_DES.

  Requested Function

  `options` must include one of the following constants:

DES_ENCRYPT	Encrypt the input.
DES_DECRYPT	Decrypt the input.
DES_MAC	Generate a message authentication code for the input.

  If DES_MAC is specified, DES_CBC_MODE is forced.

  DES Mode

  `options` must include one of the following constants:

DES_CBC_MODE	Use outer CBC mode.
DES_ECB_MODE	Use ECB mode.

  DES_CBC_MODE is forced if `options` specifies DES_MAC. In this case, DES_ECB_MODE must not be specified. DES_CBC_MODE may be specified but need not be.

Source of Input

`options` must include one of the following constants:

DES_INTERNAL_INPUT
: Read input data from an internal buffer.

DES_EXTERNAL_INPUT
: Read input data from an external buffer.

See “Internal and External Buffers” on page 3-21 for details.

Destination of Output

`options` must include one of the following constants:

DES_INTERNAL_OUTPUT
: Write output data to an internal buffer.

DES_EXTERNAL_OUTPUT
: Write output data to an external buffer.

See “Internal and External Buffers” on page 3-21 for details.

If `options` specifies DES_MAC, the “Destination of Output” options are ignored.

Padding Options

`options` may include the following constants:

DES_PREPAD	Prepad the input with eight bytes of data.
DES_PAD_WITH_8	Pad the input with eight bytes of data.
DES_PAD_WITH_16	Pad the input with sixteen bytes of data.

DES_PREPAD is ignored unless `options` specifies DES_MAC. Either DES_PAD_WITH_8 or DES_PAD_WITH_16 may be specified, but not both.

- `key1`, `key2`, and `key3` are the keys to use in the operation. If `options` specifies DES_ENCRYPT or DES_MAC, the input is encrypted with `key1`, decrypted with `key2`, and encrypted with `key3`. If `options` specifies DES_DECRYPT, the input is decrypted with `key3`, encrypted with `key2`, and decrypted with `key1`.
- `init_v` is the initialization vector for the operation. `init_v` is not used if `options` specifies DES_ECB_MODE.
- `source` describes the location of the buffer containing the input data. If `options` specifies DES_INTERNAL_INPUT, `source.internal` defines an internal buffer that contains the input. If `options` specifies DES_EXTERNAL_INPUT, `source.external` defines an external buffer that contains the input. See “Internal and External Buffers” on page 3-21 for details.
- `destination` describes the location of the buffer to which the output is to be written. If `options` specifies DES_INTERNAL_OUTPUT, `destination.internal` defines an internal buffer to which to write the output. If `options` specifies DES_EXTERNAL_OUTPUT, `destination.external` defines an external buffer to which to write the output. See “Internal and External Buffers” on page 3-21 for details.

  If `options` specifies DES_MAC, `destination` is not used.
- `prePadding` contains eight bytes of data to which the input (including any padding) is appended before the requested cryptographic operation is performed. `prePadding` is not used unless `options` specifies DES_MAC.

- postPadding contains data that is appended to the input (including any prepadding) before the requested cryptographic operation is performed. If options specifies DES_PAD_WITH_8, postPadding[0] through postPadding[7] are appended to the input. If options specifies DES_PAD_WITH_16, the entire postPadding field is appended to the input. If options specifies neither DES_PAD_WITH_8 nor DES_PAD_WITH_16, postPadding is not used.

pMsgID determines whether the function is performed synchronously or asynchronously:

- If pMsgID is NULL, the function is performed synchronously. The call does not return until the requested cryptographic operation is complete.
- If pMsgID is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the DES Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the request block referenced by pTDESRB or the buffers described by pTDESRB->source and pTDESRB->destination before the operation is complete.

Output

On successful exit from this routine:

If pMsgID is NULL, pTDESRB->term_v contains

- the message authentication code for the input if options specifies DES_MAC,
- the value to use for the initialization vector in the next call to sccTDESAsync if options specifies DES_CBC_MODE, and
- garbage if options specifies DES_ECB_MODE.

The buffer described by pTDESRB->destination contains

- the contents of the buffer described by pTDESRB->source triple-DES encrypted with the keys in *pTDESRB if options specifies DES_ENCRYPT and
- the contents of the buffer described by pTDESRB->source triple-DES decrypted with the keys in *pTDESRB if options specifies DES_DECRYPT.

If pMsgID is not NULL, *pMsgID uniquely identifies the message that was sent to the DES Manager to initiate the desired operation. When the operation is complete, the DES Manager will send the coprocessor application a message whose type field (MSG.h.msg_type) contains this identifier and whose first (and only) data item (MSG.msg_data[0]) contains the return code generated by the routine.[20] If the operation was successful, pTDES8RB->term_v and pTDESRB->destination contain the results as previously described. The message is placed on the default CP/Q message queue for the task that called sccTDESAsync.

Notes

Notes on Source and Destination Buffers

The length of the input data (including the length of the input buffer described by pTDESRB->source and any pad bytes) may be less than the length of the output

[20] The return code from the call to sccTDESAsync indicates whether or not the initial message to the DES Manager was successfully enqueued.

buffer described by `pTDESRB->destination`. In this case, any excess bytes at the end of the output buffer are not affected by sccTDESAsync.

If `destination` is used, the buffers described by `source` and `destination` should either be the same buffer or not overlap at all.

Return Codes

Common return codes generated by this routine are:

DMGood (i.e., 0) The operation was successful.

DMNotAuth The coprocessor application is not authorized to perform DES operations (for example, because it has not called sccSignOn).

DMBadFlags The `options` argument is not valid.

DMBadParm The length of the input data or the output data is invalid (for example, not a multiple of 8) or the length of the input data exceeds the length of the output buffer, or an internal buffer is not aligned on a 4-byte boundary.

DMBadAddr The input buffer is an internal buffer and is not readable or the output buffer is an internal buffer and is not writeable.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccEDE3_3DES - Perform EDE3 Mode Triple-DES Operation

sccEDE3_3DES performs an EDE3 mode triple-DES operation. EDE3 mode is defined in Appendix C of the *IBM 4758 CCA Basic Services Reference and Guide*.

Note

EDE3 mode is an “inner-CBC” mode of triple DES and thus is not compliant with the ANSI X9.52 3DES standard.

Function Prototype

```
long sccEDE3_3DESAsync(sccEDE3DES_RB_t     *pEDE3DESRB,
                       unsigned long       *pMsgID);

#define sccEDE3_3DES(p)  sccEDE3_3DESAsync(p,NULL)
```

Input

On entry to this routine:

pEDE3DESRB must contain the address of an EDE3 operation request block structure whose fields are initialized as follows:

- `options` determines the nature of each of the operations the function performs and must be DES_ENCRYPT for Encipher-Decipher-Encipher or DES_DECRYPT for Decipher-Encipher-Decipher.
- `input` must contain the address of the data to be transformed by the operation.
- `output` must contain the address of a buffer to hold the transformed data.
- `count` must contain the length in bytes of the input data and the output buffer. `count` must be a multiple of 8.
- `key1`, `key2`, and `key3` are the encryption/decryption keys for pass 1, pass 2, and pass 3, respectively.

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until the operation is complete.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the DES Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the request block referenced by `pEDE3DESRB` or the buffers referenced by `pEDE3DESRB->input` and `pEDE3DESRB->output` before the operation is complete.

Output

On successful exit from this routine:

If `pMsgID` is NULL, `*(pEDE3DESRB->output)` contains the contents of `*(pEDE3DESRB->input)` as transformed by the EDE3 mode triple DES operation using `pEDE3DESRB->key1`, `pEDE3DESRB->key2`, and `pEDE3DESRB->key3` as dictated by the value of `options`.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the DES Manager to initiate the desired operation. When the operation is complete,

the DES Manager will send the coprocessor application a message whose type field (`MSG.h.msg_type`) contains this identifier and whose first (and only) data item (`MSG.msg_data[0]`) contains the return code generated by the routine.[21] If the operation was successful, `*(pEDE3DESRB->output)` contains the result. The message is placed on the default CP/Q message queue for the task that called sccEDE3_3DESAsync.

Notes

Additional Sources of Information

Refer to Appendix C of the *IBM 4758 CCA Basic Services Reference and Guide* for a detailed description of the operation of and restrictions imposed by this function.

Return Codes

Common return codes generated by this routine are:

DMGood (i.e., 0) The operation was successful.

DMNotAuth The coprocessor application is not authorized to perform DES operations (for example, because it has not called sccSignOn).

DMBadFlags The `options` argument is not valid.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

[21] The return code from the call to sccEDE3_3DESAsync indicates whether or not the initial message to the DES Manager was successfully enqueued.

sccTransformCDMFKey - Transform DES Key to CDMF Key

sccTransformCDMFKey transforms a 56-bit DES key into a 56-bit key suitable for use with the Commercial Data Masking Facility (CDMF) algorithm. The transformed key provides the same level of cryptographic security as a 40-bit DES key.

Function Prototype

```
long sccTransformCDMFKeyAsync(sccDES_key_t      keyIn
                              sccDES_key_t      keyOut
                              unsigned long    *pMsgID);

#define sccTransformCDMFKey(ki,ko) sccTransformCDMFKeyAsync(ki,ko,NULL)
```

Input

On entry to this routine:

`keyIn` must contain the 56-bit DES key to be transformed.

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until the operation is complete.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the DES Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of `keyOut` before the operation is complete.

Output

On successful exit from this routine:

If `pMsgID` is NULL, `keyOut` contains a 56-bit CDMF key derived from `keyIn`.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the DES Manager to initiate the desired operation. When the operation is complete, the DES Manager will send the coprocessor application a message whose type field (`MSG.h.msg_type`) contains this identifier and whose first (and only) data item (`MSG.msg_data[0]`) contains the return code generated by the routine.[22] If the operation was successful, `keyOut` contains the result. The message is placed on the default CP/Q message queue for the task that called sccTransformCDMFKeyAsync.

[22] The return code from the call to sccTransformCDMFKeyAsync indicates whether or not the initial message to the DES Manager was successfully enqueued.

Notes

Serial Transformation Discouraged

A CDMF key should not be passed as the `keyIn` input to sccTransformCDMFKey.

Return Codes

Common return codes generated by this routine are:

DMGood (i.e., 0) The operation was successful.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

Public Key Algorithm Functions

The functions described in this section allow a coprocessor application to request services from the Public Key Algorithm (PKA) Manager, which uses the coprocessor's large integer modular math hardware to support public key cryptographic algorithms. Currently, the following algorithms are supported:

- RSA (Rivest-Shamir-Adleman) encryption, decryption, and X.931 digital signature support
- DSA (Digital Signature Algorithm) digital signature

RSA Key Tokens

The interface to the PKA Manager defines the `sccRSAKeyToken_t` type to hold information about RSA public and private keys and defines the `sccPKCSKeyToken_t` type to hold information about RSA private keys stored in PKCS#1 CRT form. An item of either type has a variable length and consists of a descriptive header followed by a buffer containing the values of the various elements of the key. (For example, the key token for an RSA public key contains the values of the modulus *n* and the public exponent *e*.) The header indicates which elements are present and gives the length and offset within the token of the first byte of each element. Elements are stored in big-endian order: the byte at the lowest address contains the most significant byte of the element.[23]

The fields of the key token for an RSA public key are set as follows:

- `type` is RSA_PUBLIC_MODULUS_EXPONENT.
- `tokenLength` is the length in bytes of the key token. Note that this value is typically larger than `sizeof(sccRSAKeyToken_t)` because the `sccRSAKeyToken_t` structure maps only the first byte of the buffer that contains the elements of the key.
- `n_BitLength` is the length in bits of the modulus *n*.[24]
- `n_Length` is the length in bytes of *n*.
- `n_Offset` is the offset in bytes from the start of the key token to the first byte of *n*.[25]
- `e_Length` is the length in bytes of the public exponent *e*.[26]
- `e_Offset` is the offset in bytes from the start of the token to the first byte of *e*.[25]
- `tokenData` marks the beginning of the buffer that contains the values of *n* and *e*.

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

The PKA Manager supports six kinds of key token for an RSA private key. The PKA Manager uses a straightforward modular exponentiation approach to decrypt ciphertext or wrap an X9.31 encapsulated hash, as appropriate, using a `sccRSAKeyToken_t` key token whose fields are set as follows:

- `type` is RSA_PRIVATE_MODULUS_EXPONENT (decrypt ciphertext) or RSA_X931_PRIVATE_MODULUS_EXPONENT (wrap encapsulated hash).

---

[23] Fields in the header are normal arithmetic items and are stored in little-endian order.

[24] *n* is the product of two prime numbers, *p* and *q*.

[25] That is, if `t` is the key token and *x* is an element of the key, the first byte of *x* is `((char *) &t) [t.x_Offset]`.

[26] *e* is the inverse of the private exponent *d* modulo *(p-1)(q-1)*.

- `tokenLength` is the length in bytes of the key token. Note that this value is typically larger than `sizeof(sccRSAKeyToken_t)` because the `sccRSAKeyToken_t` structure maps only the first byte of the buffer that contains the elements of the key.
- `n_BitLength` is the length in bits of the modulus *n*. If `type` is RSA_X931_PRIVATE_MODULUS_EXPONENT, `n_BitLength` must be 1024, 1280, 1536, 1792, or 2048.
- `n_Length` is the length in bytes of *n*.
- `n_Offset` is the offset in bytes from the start of the key token to the first byte of *n*.[29]
- `e_Length` is the length in bytes of the public exponent *e*.
- `e_Offset` is the offset in bytes from the start of the key token to the first byte of *e*.[29]
- `x.d_Length` is the length in bytes of the private exponent *d*.[27]
- `y.d_Offset` is the offset in bytes from the start of the key token to the first byte of *d*.[29]
- `r_Length` is the length in bytes of the blinding value *r*.[28]
- `r_Offset` is the offset in bytes from the start of the key token to the first byte of *r*.[29]
- `r1Length` is the length in bytes of the inverse of the blinding value, r^{-1}.[28]
- `r1Offset` is the offset in bytes of the first byte of r^{-1} within the key token.[29]
- `tokenData` marks the beginning of the buffer that contains the values of *n*, *e*, *d*, *r*, and r^{-1}.

The remaining length and offset fields are not used and should be set to zero.

The PKA Manager uses an approach based on the Chinese Remainder Theorem to decrypt ciphertext or wrap an X9.31 encapsulated hash, as appropriate, using a `sccRSAKeyToken_t` key token whose fields are set as follows:

- `type` is RSA_PRIVATE_CHINESE_REMAINDER (decrypt ciphertext) or RSA_X931_PRIVATE_CHINESE_REMAINDER (wrap encapsulated hash).
- `tokenLength` is the length in bytes of the key token. Note that this value is typically larger than `sizeof(sccRSAKeyToken_t)` because the `sccRSAKeyToken_t` structure maps only the first byte of the buffer that contains the elements of the key.
- `n_BitLength` is the length in bits of the modulus *n*. If `type`, is RSA_X931_PRIVATE_CHINESE_REMAINDER, `n_BitLength` must be 1024, 1280, 1536, 1792, or 2048.
- `n_Length` is the length in bytes of *n*.
- `n_Offset` is the offset in bytes from the start of the key token to the first byte of *n*.[29]
- `e_Length` is the length in bytes of the public exponent *e*.
- `e_Offset` is the offset in bytes from the start of the key token to the first byte of *e*.[29]
- `x.p_Length` is the length in bytes of the prime number *p*.
- `y.p_Offset` is the offset in bytes from the start of the key token to the first byte of *p*.[29]
- `q_Length` is the length in bytes of the prime number *q*.

---

[27] *d* is the inverse of the public exponent *e* modulo *(p-1)(q-1)*.

[28] $r = R^e \bmod n$ and $r^{-1} = R^{-1} \bmod n$, where *R* is a random number less than the modulus *n*.

[29] That is, if `t` is the key token and *x* is an element of the key, the first byte of *x* is `((char *) &t) [t.x_Offset]`.

- q_Offset is the offset in bytes from the start of the key token to the first byte of *q*.[30]
- dp_Length is the length in bytes of *dp = d mod(p-1)*.
- dp_Offset is the offset in bytes from the start of the key token to the first byte of *dp*.[30]
- dq_Length is the length in bytes of *dq = d mod(q-1)*.
- dq_Offset is the offset in bytes from the start of the key token to the first byte of *dq*.[30]
- ap_Length is the length in bytes of $ap = q^{p-1} \bmod n$.
- ap_Offset is the offset in bytes from the start of the key token to the first byte of *ap*.[30]
- aq_Length is the length in bytes of *aq = n + 1 - ap*.
- aq_Offset is the offset in bytes from the start of the key token to the first byte of *aq*.[30]
- r_Length is the length in bytes of the blinding value *r*.
- r_Offset is the offset in bytes from the start of the key token to the first byte of *r*.[30]
- r1Length is the length in bytes of the inverse of the blinding value, r^{-1}.
- r1Offset is the offset in bytes of the first byte of r^{-1} within the key token.[30]
- tokenData marks the beginning of the buffer that contains the values of *n*, *e*, *p*, *q*, *dp*, *dq*, *ap*, *aq*, *r*, and r^{-1}.

The PKA Manager also uses an approach based on the Chinese Remainder Theorem to decrypt ciphertext or wrap an X9.31 encapsulated hash, as appropriate, using a sccPKCSKeyToken_t key token whose fields are set as follows:

- type is RSA_PKCS_PRIVATE_CHINESE_REMAINDER (decrypt ciphertext) or RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER (wrap encapsulated hash).
- tokenLength is the length in bytes of the key token. Note that this value is typically larger than sizeof(sccRSAKeyToken_t) because the sccRSAKeyToken_t structure maps only the first byte of the buffer that contains the elements of the key.
- n_BitLength is the length in bits of the modulus *n*. If type, is RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER, n_BitLength must be 1024, 1280, 1536, 1792, or 2048.
- n_Length is the length in bytes of *n*.
- n_Offset is the offset in bytes from the start of the key token to the first byte of *n*.[30]
- e_Length is the length in bytes of the public exponent *e*.
- e_Offset is the offset in bytes from the start of the key token to the first byte of *e*.[30]
- x.p_Length is the length in bytes of the prime number *p*.
- y.p_Offset is the offset in bytes from the start of the key token to the first byte of *p*.[30]
- q_Length is the length in bytes of the prime number *q*.
- q_Offset is the offset in bytes from the start of the key token to the first byte of *q*.[30]
- dp_Length is the length in bytes of *dp = d mod(p-1)*.
- dp_Offset is the offset in bytes from the start of the key token to the first byte of *dp*.[30]

[30] That is, if t is the key token and *x* is an element of the key, the first byte of *x* is ((char *) &t) [t.*x*_Offset].

- `dq_Length` is the length in bytes of $dq = d \bmod (q-1)$.
- `dq_Offset` is the offset in bytes from the start of the key token to the first byte of *dq*.[31]
- `qInvLength` is the length in bytes of $q^{-1} \bmod p$.
- `qInvOffset` is the offset in bytes of the first byte of $q^{-1} \bmod p$ within the key token.
- `notDefined` and `notDefined2` are reserved and should be set to zero.
- `r_Length` is the length in bytes of the blinding value *r*.
- `r_Offset` is the offset in bytes from the start of the key token to the first byte of *r*.[31]
- `r1Length` is the length in bytes of the inverse of the blinding value, r^{-1}.
- `r1Offset` is the offset in bytes of the first byte of r^{-1} within the key token.[31]
- `tokenData` marks the beginning of the buffer that contains the values of *n*, *e*, *p*, *q*, *dp*, *dq*, $q^{-1} \bmod p$, *r*, and r^{-1}.

Use of a private key of type RSA_PRIVATE_CHINESE_REMAINDER, RSA_X931_PRIVATE_CHINESE_REMAINDER, RSA_PKCS_PRIVATE_CHINESE_REMAINDER, or RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER can improve performance with no loss of security.

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

The `n_BitLength` field of an RSA key token cannot exceed 2048. If it is not a multiple of 8, any excess high-order bits in the modulus are treated as zeros (that is, *n* is essentially padded on the left with zeros, regardless of the actual bits that appear in the key token).

[31] That is, if `t` is the key token and *x* is an element of the key, the first byte of *x* is `((char *) &t) [t.x_Offset]`.

sccRSAKeyGenerate - Generate RSA Key Pair

sccRSAKeyGenerate generates a key token for an RSA private key. The token includes information that defines the corresponding RSA public key.

Function Prototype

```
long sccRSAKeyGenerateAsync(sccRSAKeyGen_RB_t  *pRSAKGRB,
                            unsigned long      *pMsgID);

#define sccRSAKeyGenerate(p) sccRSAKeyGenerateAsync(p, NULL)
```

Input

On entry to this routine:

pRSAKGRB must contain the address of an RSA key generate request block whose fields are initialized as follows:

- key_type specifies which kind of private key token is generated and must be one of the following constants:
  - RSA_PRIVATE_MODULUS_EXPONENT
  - RSA_PRIVATE_CHINESE_REMAINDER
  - RSA_X931_PRIVATE_MODULUS_EXPONENT
  - RSA_X931_PRIVATE_CHINESE_REMAINDER
  - RSA_PKCS_PRIVATE_CHINESE_REMAINDER
  - RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER
- mod_size specifies the desired length in bits of the modulus *n*. mod_size must be less than or equal to 2048. If key_type specifies RSA_X931_PRIVATE_MODULUS_EXPONENT, RSA_X931_PRIVATE_CHINESE_REMAINDER, or RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER mod_size must be 1024, 1280, 1536, 1792, or 2048.
- public_exp determines how the value of the public exponent *e* is chosen and must be one of the following constants:

RSA_EXPONENT_RANDOM	Choose a pseudo-random number containing mod_size bits that meets the standards described in the ANSI X9.31 specification.
RSA_EXPONENT_FIXED	Use the value of *e* in the RSA key token referenced by key_token.
RSA_EXPONENT_2	Use 2.
RSA_EXPONENT_3	Use 3.
RSA_EXPONENT_65537	Use 65537.

  RSA_EXPONENT_3 and RSA_EXPONENT_65537 provide support for certain standards that require specific public exponents (for example, SET).

  If RSA_EXPONENT_FIXED is specified, the public exponent must be odd unless key_type is RSA_X931_PRIVATE_MODULUS_EXPONENT, RSA_X931_PRIVATE_CHINESE_REMAINDER, or RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER.

  If RSA_EXPONENT_2 is specified, key_type must be RSA_X931_PRIVATE_MODULUS_EXPONENT, RSA_X931_PRIVATE_CHINESE_REMAINDER, or RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER.
- key_token must contain the address of a buffer in which an item of type sccRSAKeyToken_t can be stored. If public_exp is RSA_EXPONENT_FIXED,

key_token.tokenlength, key_token->e_Length, key_token->e_Offset, and key_token->tokenData must describe the public exponent *e*.

- key_size must contain the address of a variable in which an item of type unsigned long can be stored. *key_size must be the length in bytes of the buffer referenced by key_token.
- regen_data points to a string of bits used to seed the PKA Manager's pseudo-random number generator, which is used to generate the prime numbers *p* and *q* (and the public exponent *e* if public_exp is RSA_EXPONENT_RANDOM). Use of regen_data ensures reproducible results and thus assists testing and benchmarking. regen_data should be NULL when generating keys in the course of normal operations. In that case, the PKA Manager obtains its random numbers from the RNG Manager. If regen_data is not NULL, the string it references should contain at least 160 bits of entropy to ensure the keys generated from the seed are cryptographically sound.[32]
- regen_size must contain the length in bytes of the string referenced by regen_data. If regen_data is NULL, regen_size must be zero.

pMsgID determines whether the function is performed synchronously or asynchronously:

- If pMsgID is NULL, the function is performed synchronously. The call does not return until the key token has been generated.
- If pMsgID is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the PKA manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the request block referenced by pRSAKGRB or the areas of memory referenced by pRSAKGRB->key_token, pRSAKGRB->key_size, and pRSAKGRB->regen_data before the operation is complete.

Output

On successful exit from this routine:

If pMsgID is NULL, *(pRSAKGRB->key_token) contains a key token for an RSA private key. pRSAKGRB->key_token->type is equal to pRSAKGRB->key_type and the remaining fields of the key token are set appropriately. *(pRSAKGRB->key_size) contains the length in bytes of the key token.

If pMsgID is not NULL, *pMsgID uniquely identifies the message that was sent to the PKA Manager to initiate the desired operation. When the operation is complete, the PKA Manager will send the coprocessor application a message whose type field (MSG.h.msg_type) contains this identifier and whose first (and only) data item (MSG.msg_data[0]) contains the return code generated by the routine.[33] If the operation was successful, *(pRSAKGRB->key_token) and *(pRSAKGRB->key_size) contain the result. The message is placed on the default CP/Q message queue for the task that called sccRSAKeyGenerateAsync.

---

[32] Entropy is a measure of the uncertainty, unpredictability, and randomness in data output from a source. For a clearer explanation of entropy and suggestions on how to obtain random seeds with sufficient entropy see *http://www.rsa.com/rsa/developers/random.htm*.

[33] The return code from the call to sccRSAKeyGenerateAsync indicates whether or not the initial message to the PKA Manager was successfully enqueued.

Notes

Generating Public Keys

A key token for an RSA public key can be generated from the key token for the corresponding RSA private key by copying *n*, *e*, and `n_BitLenth` from the private key token and setting the public key token's `type` field to RSA_PUBLIC_MODULUS_EXPONENT.

Return Codes

Common return codes generated by this routine are:

PKAGood (i.e., 0) The operation was successful.

PKABadParm An argument is not valid.

PKANoSpace The operation failed due to lack of space (for example, the buffer referenced by `pRSAKGRB->key_token` is not large enough to hold the token generated by the call or there is no free memory available to the PKA manager).

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccRSA - Encipher/Decipher Data or Wrap/Unwrap X9.31 Encapsulated Hash

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

Function Prototype

```
long sccRSAAsync(sccRSA_RB_t    *pRSARB,
                 unsigned long  *pMsgID);

#define sccRSA(p)  sccRSAAsync(p, NULL)
```

Input

On entry to this routine:

`pRSARB` must contain the address of an RSA operation request block whose fields are initialized as follows:

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

  Public or Private Key

  `options` must include one of the following constants:

RSA_PUBLIC	Perform the operation using the public key from the key token (that is, output = input^{e} mod n).
RSA_PRIVATE	Perform the operation using the private key from the key token (for example, output = input^{d} mod n).

  If RSA_PRIVATE is specified, `key_token->type` must not be RSA_PUBLIC_MODULUS_EXPONENT.

  If RSA_PRIVATE is specified, RSA_DECRYPT must also be specified. If RSA_PUBLIC is specified, RSA_ENCRYPT must also be specified.

  RSA_PRIVATE must be specified to wrap an X9.31 encapsulated hash. RSA_PUBLIC must be specified to unwrap an X9.31 encapsulated hash.

  Blinding Operation

  sccRSA's implementation of the RSA algorithm may be vulnerable to a timing attack.[34] That is, an adversary can use differences in the amount of time it takes to process various messages with a particular private key to defeat the cryptographic security provided by the key. The blinding values in a key token, r and r^{-1}, are used to defeat timing attacks. `options` may include one of the following constants:

RSA_DONT_BLIND	Perform the operation without using the blinding values.
RSA_BLIND_NO_UPDATE	Perform the operation using the blinding values and replace the blinding values in the key token.[35]

[34] IBM believes certain features of the IBM 4758 Model 002 eliminate this vulnerability.

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

RSA_BLIND_UPDATE	Perform the operation using the blinding values but do not replace the blinding values in the key token.

RSA_DONT_BLIND yields the best performance, but makes the operation vulnerable to timing attacks. RSA_BLIND_NO_UPDATE requires the most time. RSA_BLIND_UPDATE uses a fast blinding scheme but is secure only if the coprocessor application replaces the blinding values in the key token (for example, by calling sccComputeBlindingValues) before calling sccRSA again.

RSA_BLIND_NO_UPDATE is the default.

ANSI X9.31 Operation

`options` must include RSA_X931_OPERATION if `key_token_->key_type` is RSA_X931_PRIVATE_MODULUS_EXPONENT, RSA_X931_PRIVATE_CHINESE_REMAINDER, or RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER.

- `*key_token` is the RSA key token for the key to be used in the operation.
- `key_size` is the length in bytes of the RSA key token referenced by `key_token` (that is, `key_token->tokenLength`).
- `data_in` points to a buffer that contains the input data. If `options` specifies RSA_X931_OPERATION, the buffer referenced by `data_in` is assumed to contain a valid X9.31 encapsulated hash. The encapsulated hash should be wrapped if `options` specifies RSA_PUBLIC and should not be wrapped if `options` specifies RSA_PRIVATE.
- `data_out` points to a buffer that is to contain the result of the operation.
- `data_size` is the length in **bits** of the buffers referenced by `data_in` and `data_out`. `data_size` should be equal to `key_token->n_BitLength`.

Note: The `data_size` field for sccRSA is the length in **bits** of the input and output buffers, whereas the `data_size` field for sccDSA is the length in **bytes** of the input buffer.

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until the requested operation has been performed.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the PKA manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the request block referenced by `pRSARB` or the areas of memory referenced by `pRSARB->key_token`, `pRSARB->data_in`, and `pRSARB->data_out` before the operation is complete.

Output

On successful exit from this routine:

If `pMsgID` is NULL, `*(pRSARB->data_out)` contains

- `*(pRSARB->data_in)` transformed using the public key (*n* and *e*) from `*(pRSARB->key_token)` if `pRSARB->options` specifies RSA_PUBLIC and
- `*(pRSARB->data_in)` transformed using the private key from `*(pRSARB->key_token)` if `pRSARB->options` specifies RSA_PRIVATE.

The blinding values *r* and r^{-1} in `*(pRSARB->key_token)` are replaced if `pRSARB->options` specifies RSA_BLIND_NO_UPDATE.

If pMsgID is not NULL, *pMsgID uniquely identifies the message that was sent to the PKA Manager to initiate the desired operation. When the operation is complete, the PKA Manager will send the coprocessor application a message whose type field (MSG.h.msg_type) contains this identifier and whose first (and only) data item (MSG.msg_data[0]) contains the return code generated by the routine.[36] If the operation was successful, *(pRSARB->data_out) and the blinding values in *(pRSARB->key_token) contain the result. The message is placed on the default CP/Q message queue for the task that called sccRSAAsync.

Notes

Buffer Overlap

The buffers referenced by pRSARB->data_in and pRSARB->data_out should either be the same buffer or not overlap at all.

Buffer Length Not Equal to Modulus Length

If the length of the input and output buffers is less than the length of the modulus *n* (that is, if pRSARB->data_size < pRSARB->key_token->n_BitLength), sccRSAAsync returns PKABadParm.

If the length of the input and output buffers is greater than the length of the modulus *n* (that is, if pRSARB->data_size > pRSARB->key_token->n_BitLength), sccRSAAsync processes the rightmost bytes of the input buffer and places its result in the rightmost bytes of the output buffer. For example,

```
char              inbuffer[256];
char              outbuffer[256];
sccRSA_RB_t       RSARB;
sccRSAKeyToken_t *pToken;

...
pToken->n_BitLength = 1024;
sccRSAKeyGenerate(...);  /* Generate 1024-bit RSA keypair */
...
RSARB.data_in   = inbuffer;
RSARB.data_out  = outbuffer;
RSARB.data_size = 256*8; /* Input data and output buffer are 2048 bits */
sccRSA(&RSARB);

/*
* sccRSA processes inbuffer[128] through inbuffer[255]
* and places the result in outbuffer[128] through outbuffer[255]
*/
```

X9.31 Support

The X9.31 signature generation process incorporates three steps:

1. The message is hashed.
2. The hash is encapsulated.
3. The encapsulated hash is wrapped to generate the signature.

[36] The return code from the call to sccRSAAsync indicates whether or not the initial message to the PKA Manager was successfully enqueued.

sccRSAAsync performs the third step as dictated by the X9.31 specification if

- `options` includes RSA_PRIVATE and RSA_X931_OPERATION,
- `*key_token->key_type` is RSA_X931_PRIVATE_MODULUS_EXPONENT, RSA_X931_PRIVATE_CHINESE_REMAINDER, or RSA_PKCS_X931_PRIVATE_CHINESE_REMAINDER (and had that value when the key was generated), and
- the buffer referenced by `data_in` contains a valid X9.31 encapsulated hash.

The first two steps are the application's responsibility.

Similarly, the signature verification process incorporates four steps:

1. The signature is opened (or unwrapped) to produce an encapsulated hash.
2. The format of the encapsulated hash is verified.
3. The hash value is extracted from the encapsulated hash.
4. The message is hashed and the value is compared to the extracted hash.

sccRSAAsync performs the first step as dictated by the X9.31 specification if

- `options` includes RSA_PUBLIC and RSA_X931_OPERATION,
- `*key_token->key_type` is RSA_PUBLIC (and the key itself corresponds to the private key used to generate the signature), and
- the buffer referenced by `data_in` contains a valid X9.31 signature.

The last three steps are the application's responsibility.

Return Codes

Common return codes generated by this routine are:

PKAGood (i.e., 0)	The operation was successful.
PKABadAddr	The offset and length for an element in a key token are invalid (that is, the element described does not fit within the key token buffer).
PKABadParm	An argument is not valid or `pRSARB->data_size` is less than `pRSARB->key_token->n_BitLength`.
PKANoSpace	The operation failed due to lack of space (for example, the buffer referenced by `pRSARB->key_token` is not large enough to hold the blinding values generated by the call).
PKARangeOverflow	The last `pRSARB->key_token->n_Length` bytes of `*(pRSARB->data_in)`, when interpreted as a big-endian integer, exceed the value of the modulus *n*.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccComputeBlindingValues - Compute Blinding Values for RSA Key

sccRSA's implementation of the RSA algorithm may be vulnerable to a timing attack.[37] That is, an adversary can use differences in the amount of time it takes to process various messages with a particular private key to defeat the cryptographic security provided by the key. sccComputeBlindingValues generates two large integers that can be used to defeat timing attacks.

Function Prototype

```
long sccComputeBlindingValuesAsync(sccCBV_RB_t     *pCBVRB,
                                   unsigned long   *pMsgID) ;

#define sccComputeBlindingValues(p)  sccComputeBlindingValuesAsync(p,NULL)
```

Input

On entry to this routine:

`pCBVRB` must contain the address of an RSA blinding value request block whose fields are initialized as follows:

- `n` must contain the address of the most significant byte of the modulus *n*.
- `nsize` is the length in **bits** of *n*.
- `e` must contain the address of the most significant byte of the public exponent *e*.
- `esize` is the length in **bytes** of *e*, which contains the same number of **bits** as *n*. If the length of *e* in **bits** is not a multiple of eight, any excess high-order bits in the public exponent are treated as zeros (that is, *e* is essentially padded on the left with zeros, regardless of the actual bits that appear in the buffer).
- `r_e` must contain the address of a buffer large enough to contain the blinding value *r* generated by the call. This value has the same number of bits as the modulus *n*.
- `rinv` must contain the address of a buffer large enough to contain the inverse of the blinding value r^{-1} generated by the call. This value has the same number of bits as the modulus *n*.

The modulus, public exponent, and blinding values are stored in big-endian order: the byte at the lowest address contains the most significant byte of the value.

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until the blinding values have been generated.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the RSA manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the request block referenced by `pCBVRB` or the areas of memory referenced by `pCBVRB->n`, `pCBVRB->e`, `pCBVRB->r_e`, and `pCBVRB->rinv` before the operation is complete.

[37] IBM believes certain features of the IBM 4758 Model 002 eliminate this vulnerability.

Output

On successful exit from this routine:

If `pMsgID` is NULL, `*(pCBVRB->r_e)` and `*(pCBVRB->rinv)` contain the blinding values generated by the call.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the PKA Manager to initiate the desired operation. When the operation is complete, the PKA Manager will send the coprocessor application a message whose type field (`MSG.h.msg_type`) contains this identifier and whose first (and only) data item (`MSG.msg_data[0]`) contains the return code generated by the routine.[38] If the operation was successful, the blinding values referenced by `pCBVRB->r_e` and `pCBVRB->r_inv` contain the result. The message is placed on the default CP/Q message queue for the task that called sccComputeBlindingValuesAsync.

Return Codes

Common return codes generated by this routine are:

PKAGood (i.e., 0) The operation was successful.

PKABadParm An argument is not valid.

PKANoSpace The buffer referenced by `pCBVRB->r_e` or `pCBVRB->rinv` is not large enough to hold the updated blinding values generated by the call.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

[38] The return code from the call to sccComputeBlindingValuesAsync indicates whether or not the initial message to the PKA Manager was successfully enqueued.

DSA Key Tokens

The interface to the PKA Manager defines the `sccDSAKeyToken_t` type to hold information about DSA public and private keys. An item of type `sccDSAKeyToken_t` has a variable length and consists of a descriptive header followed by a buffer containing the values of the various elements of the key. (For example, the key token for a DSA public key contains the values of the large prime *p*, the small prime *q*, the generator *g*, and the public exponent *y*.) The header indicates which elements are present and gives the length and offset within the token of the first byte of each element. Elements are stored in big-endian order: the byte at the lowest address contains the most significant byte of the element.[39]

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

- `key_type` is DSA_PUBLIC_KEY_TYPE.
- `key_token_length` is the length in bytes of the key token. Note that this value is typically larger than `sizeof(sccDSAKeyToken_t)` because the `sccDSAKeyToken_t` structure maps only the first byte of the buffer that contains the elements of the key.
- `prime_p_bit_length` is the length in bits of the large prime *p*. `prime_p_bit_length` must be a multiple of 64 between 512 and 1024, inclusive.
- `p_length` is the length in bytes of *p*.
- `p_offset` is the offset in bytes from the start of the key token to the first byte of *p*.[40]
- `q_length` is the length in bytes of the small prime *q*. `q_length` must be 20.
- `q_offset` is the offset in bytes from the start of the key token to the first byte of *q*.[40]
- `g_length` is the length in bytes of the generator *g*.[41]
- `g_offset` is the offset in bytes from the start of the key token to the first byte of *g*.[40]
- `y_length` is the length of the public exponent *y*.[42]
- `y_offset` is the offset in bytes from the start of the key token to the first byte of *y*.[40]
- `keydata_start` marks the beginning of the buffer that contains the values of *p*, *q*, *g*, and *y*.

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

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

- `key_type` is DSA_PRIVATE_KEY_TYPE.
- `key_token_length` is the length in bytes of the key token. Note that this value is typically larger than `sizeof(sccDSAKeyToken_t)` because the `sccDSAKeyToken_t` structure maps only the first byte of the buffer that contains the elements of the key.
- `prime_p_bit_length` is the length in bits of the large prime *p*. `prime_p_bit_length` must be a multiple of 64 between 512 and 1024, inclusive.
- `p_length` is the length in bytes of *p*.

---

[39] Fields in the header are normal arithmetic items and are stored in little-endian order.

[40] That is, if `t` is the key token and *x* is an element of the key, the first byte of *x* is `((char *) &t) [t.`*x*`_offset]`.

[41] *g* is $h^{(p-1)/q}$ modulo *p* where *h* is a number less than *p-1* chosen at random so that *g* is greater than 1.

[42] *y* is g^{x} modulo *p*, where *x* is the private exponent corresponding to the DSA public key defined by *p*, *q*, *g*, and *y*.

- `p_offset` is the offset in bytes from the start of the key token to the first byte of *p*.[43]
- `q_length` is the length in bytes of the small prime *q*. `q_length` must be 20.
- `q_offset` is the offset in bytes from the start of the key token to the first byte of *q*.[43]
- `g_length` is the length in bytes of the generator *g*.
- `g_offset` is the offset in bytes from the start of the key token to the first byte of *g*.[43]
- `y_length` is the length of the public exponent *y*.
- `y_offset` is the offset in bytes from the start of the key token to the first byte of *y*.[43]
- `x_length` is the length of the private exponent *x*.[44]
- `x_offset` is the offset in bytes from the start of the key token to the first byte of *x*.[43]
- `keydata_start` marks the beginning of the buffer that contains the values of *p*, *q*, *g*, *y*, and *x*.

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

DSA Signature Tokens

The interface to the PKA Manager defines the `sccDSASignatureToken_t` type to hold a digital signature generated by the DSA algorithm. An item of type `sccDSASignatureToken_t` consists of a descriptive header followed by a fixed-length buffer containing the various elements of the signature. The header gives the length and offset within the token of the first byte of each element. Elements are stored in big-endian order: the byte at the lowest address contains the most significant byte of the element.[45]

The fields of a DSA signature token are set as follows:

- `signature_token_length` is the length in bytes of the signature token. Note that this value is typically larger than `sizeof(sccDSASignatureToken_t)` because the `sccDSASignatureToken_t` structure maps only the first byte of the buffer that contains the elements of the signature.
- `r_length` is the length in bytes of the first half of the signature *r*.[46] `r_length` must be less than or equal to 20.
- `r_offset` is the offset in bytes from the start of the signature token to the first byte of *r*.[43]
- `s_length` is the length in bytes of the second half of the signature *s*.[47] `s_length` must be less than or equal to 20.
- `s_offset` is the offset in bytes from the start of the signature token to the first byte of *s*.[43]
- `signature_data_start` marks the start of the buffer that contains the values of *r* and *s*.

---

[43] That is, if `t` is the key token and *x* is an element of the key, the first byte of *x* is `((char *) &t) [t.`*x*`_offset]`.

[44] *x* is a number less than *q* chosen at random.

[45] Fields in the header are normal arithmetic items and are stored in little-endian order.

[46] $r = (g^k \bmod p) \bmod q$ where *k* is a number less than *q* chosen at random.

[47] $s = (H(m) + xr)/k \bmod q$ where *H(m)* is the value generated by the Secure Hash Algorithm when applied to the data to be signed.

sccDSAKeyGenerate - Generate DSA Key Pair

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

Function Prototype

```
long sccDSAKeyGenerateAsync(sccDSAKeyGen_RB_t *pDSAKGRB,
                            unsigned long     *pMsgID);

#define sccDSAKeyGenerate(p) sccDSAKeyGenerateAsync(p,NULL)
```

Input

On entry to this routine:

`pDSAKGRB` must contain the address of a DSA key generate request block whose fields are initialized as follows:

- `prime_p_size` specifies the desired length in bits of the large prime *p*. `prime_p_size` must be a multiple of 64 between 512 and 1024, inclusive.
- `key_token` must contain the address of a buffer in which an item of type `sccDSAKeyToken_t` can be stored.

  The values of the large prime *p*, the small prime *q*, and the generator *g* used to generate the DSA private key may be specified in `*key_token`. If `key_token->p_length`, `key_token->q_length`, and `key_token->g_length` are all nonzero, sccDSAKeyGenerateAsync does not generate random values for *p*, *q*, and *g* but instead uses the values in the key token.

  In this case, `key_token->p_length` and `key_token->g_length` must be equal to `prime_p_size/8` and `key_token->q_length` must be 20. `key_token->p_offset`, `key_token->q_offset`, and `key_token->g_offset` specify the locations of the first bytes of *p*, *q*, and *g*, respectively. The remaining fields of `*key_token` need not be initialized.
- `key_token_size` must be the length in bytes of the buffer referenced by `key_token`.
- `random_seed` points to a string of bits used to seed the PKA Manager's pseudo-random number generator, which is used to generate the prime numbers *p* and *q* and the generator *g*.[48] `random_seed` should be NULL when generating keys in the course of normal operations. In that case, the PKA Manager obtains its random numbers from the RNG Manager. If `random_seed` is not NULL, the string it references should contain at least 160 bits of entropy to ensure the keys generated from the seed are cryptographically sound.[49]

  `random_seed` is not used if the values of the large prime *p*, the small prime *q*, and the generator *g* are specified in `*key_token`.
- `random_seed_size` must contain the length in bytes of the string referenced by `random_seed`. If `random_seed` is NULL, `random_seed_size` must be zero.

[48] Although sccDSAKeyGenerate does use the value provided in `random_seed` to generate *p*, *q*, and *g*, it does *not* use that value to generate *x*. Instead, it always chooses *x* at random. Thus, use of `random_seed` does not ensure reproducible results.

[49] Entropy is a measure of the uncertainty, unpredictability, and randomness in data output from a source. For a clearer explanation of entropy and suggestions on how to obtain random seeds with sufficient entropy see *http://www.rsa.com/rsa/developers/random.htm*.

`random_seed_size` is not used if the values of the large prime *p*, the small prime *q*, and the generator *g* are specified in `*key_token`.

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until the key token has been generated.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the PKA Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the request block referenced by `pDSAKGRB` or the areas of memory referenced by `pDSAKGRB->key_token` and `pDSAKGRB->random_seed` before the operation is complete.

Output

On successful exit from this routine:

If `pMsgID` is NULL, `*(pDSAKGRB->key_token)` contains a key token for a DSA private key.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the PKA Manager to initiate the desired operation. When the operation is complete, the PKA Manager will send the coprocessor application a message whose type field (`MSG.h.msg_type`) contains this identifier and whose first (and only) data item (`MSG.msg_data[0]`) contains the return code generated by the routine.[50] If the operation was successful, `*(pDSAKGRB->key_token)` contains the result. The message is placed on the default CP/Q message queue for the task that called sccDSAKeyGenerateAsync.

If the caller specified values for *p*, *q*, and *g*, those values remain in `*key_token`, but they may have moved from their original positions within the token (that is, `key_token->p_offset`, `key_token->q_offset`, and `key_token->g_offset` may have changed).

Notes

Generating Public Keys

A key token for a DSA public key can be generated from the key token for the corresponding DSA private key by copying everything but *x* from the private key token and setting the public key token's `type` field to DSA_PUBLIC_KEY_TYPE.

Return Codes

Common return codes generated by this routine are:

PKAGood (i.e., 0)	The operation was successful.
PKADSAKeyGenFailed	`pDSAKGRB->prime_p_size` is not a multiple of 64 between 512 and 1024, inclusive.
PKABadParm	An argument is not valid.

[50] The return code from the call to sccDSAKeyGenerateAsync indicates whether or not the initial message to the PKA Manager was successfully enqueued.

PKANoSpace The buffer referenced by `pDSAKGRB->key_token` is not large enough to hold the token generated by the call.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccDSA - Sign Data or Verify Signature for Data

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

Function Prototype

```
long sccDSAAsync(sccDSA_RB_t    *pDSARB,
                 unsigned long  *pMsgID);

#define sccDSA(p) sccDSAAsync(p,NULL)
```

Input

On entry to this routine:

pDSARB must contain the address of a DSA operation request block whose fields are initialized as follows:

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

  Sign or Verify

  options must include one of the following constants:

DSA_SIGNATURE_SIGN	Compute the DSA signature for the input data.
DSA_SIGNATURE_VERIFY	Verify that the signature for the input data is correct.

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

  Pre-Digested Data

  options may include DSA_PRE_DIGESTED_DATA. If this option is specified, the PKA Manager assumes the input data has already been hashed using the SHA-1 algorithm and does not hash the data a second time. In this case, data_size must be 20.
- *key_token is the DSA key token for the key to be used in the operation.
- key_token_size is the length in bytes of the DSA key token referenced by key_token (that is, key_token->key_token_length).
- sig_token must contain the address of a buffer in which an item of type sccDSASignatureToken_t can be stored. If options specifies DSA_SIGNATURE_SIGN, the buffer must be at least 60 bytes long. If options specifies DSA_SIGNATURE_VERIFY, *sig_token must contain the signature that is to be checked against the block of data.
- sig_token_size is the length in bytes of the DSA signature token referenced by sig_token.
- data points to a buffer that contains the input data (that is, a SHA-1 hash if options specifies DSA_PRE_DIGESTED_DATA or an arbitrary block of data otherwise).
- data_size is the length in **bytes** of the buffer referenced by data.

  Note: The data_size field for sccDSA is the length in **bytes** of the input buffer, whereas the data_size field for sccRSA is the length in **bits** of the input and output buffers.

pMsgID determines whether the function is performed synchronously or asynchronously:

- If pMsgID is NULL, the function is performed synchronously. The call does not return until the signature has been generated or verified, as appropriate.
- If pMsgID is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the PKA Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the request block referenced by pDSARB or the areas of memory referenced by pDSARB->key_token, pDSARB->sig_token, and pRSARB->data before the operation is complete.

Output

On successful exit from this routine:

If pMsgID is NULL and pDSARB->options specifies DSA_SIGNATURE_SIGN, *(pDSARB->sig_token) contains the digital signature produced by signing *(pDSARB->data) with the private key (*g* and *x*) from *(pDSARB->key_token).

If pMsgID is NULL and pDSARB->options specifies DSA_SIGNATURE_VERIFY, a return code of zero implies that the signature in *(pDSARB->sig_token) was produced by signing *(pDSARB->data) with the private key corresponding to the public key (*p*, *q*, *g*, and *y*) from *(pDSARB->key_token).

If pMsgID is not NULL, *pMsgID uniquely identifies the message that was sent to the PKA Manager to initiate the desired operation. When the operation is complete, the PKA Manager will send the coprocessor application a message whose type field (MSG.h.msg_type) contains this identifier and whose first (and only) data item (MSG.msg_data[0]) contains the return code generated by the routine.[51] If pDSARB->options specified DSA_SIGNATURE_SIGN and the operation was successful, *(pDSARB->sig_token) contains the result. The message is placed on the default CP/Q message queue for the task that called sccDSAAsync.

Notes

Order of Elements in Generated Signature

r precedes *s* in digital signatures generated by sccDSA when pDSARB->options specifies DSA_SIGNATURE_SIGN. That is, on successful exit pDSARB->sig_token->r_offset is less than pDSARB->sig_token->s_offset.

Return Codes

Common return codes generated by this routine are:

PKAGood (i.e., 0)	The operation was successful.
PKADSASigIncorrect	pDSARB->options specifies DSA_SIGNATURE_VERIFY but the signature in *(pDSARB->sig_token) was not produced by signing *(pDSARB->data) with the private key corresponding to the public key (*p*, *q*, *g*, and *y*) from *(pDSARB->key_token).

[51] The return code from the call to sccDSAAsync indicates whether or not the initial message to the PKA Manager was successfully enqueued.

PKABadParm	An argument is not valid.
PKANoSpace	The operation failed due to lack of space.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

Large Integer Modular Math Functions

The functions described in this section allow a coprocessor application to direct the PKA Manager to perform specific modular operations on large integers. Currently, the following operations are supported:

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

Large Integers

A large integer is described by a structure of type `sccModMath_Int_t`. The fields of this structure are:

- `bytesize`, which specifies the length in bytes of the buffer that contains the integer. `bytesize` must be less than or equal to MODM_MAXBYTES.
- `bitsize`, which specifies the number of bits in the integer. `bitsize` must be less than or equal to 8*`bytesize`.
- `buffer`, which is the address of the first byte in the buffer that contains the integer.

The integer occupies the first (`bitsize` + 7)/8 bytes of the buffer. Both big-endian (`buffer[0]` is the most significant byte of the integer), and little-endian (`buffer[0]` is the least significant byte of the integer) byte orders are supported. Large integers are always nonnegative (that is, there is no sign bit).

Large integers that are passed as input arguments to the large integer modular math functions may contain leading zero bits (that is, the most significant bit of the integer may be zero). Any bits in the most significant byte that are not part of the large integer are ignored.[52]

Large integers that are generated as outputs by the large integer modular math functions do not contain leading zero bits (that is, the most significant bit of the integer is one). Any bits in the most significant byte that are not part of the large integer are zero.[52]

[52] That is, if `bitsize` is not a multiple of 8, the high-order 8-(`bitsize` mod 8) bits in the most significant byte are ignored on input and zeroed on output.

sccModMath - Perform Modular Computations

sccModMath performs one of the following operations on large integers:

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

Function Prototype

```
long sccModMathAsync(unsigned long      options,
                     unsigned long      numInts,
                     sccModMath_Int_t   aInts[],
                     unsigned long     *pMsgID);

#define sccModMath(c,n,a)  sccModMathAsync(c,n,a,NULL)
```

Input

On entry to this routine:

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

Requested Function

`options` must include one of the following constants:

MODM_MULT	Compute C = (A x B) mod N
MODM_EXP	Compute C = A^B mod N
MODM_MOD	Compute C = A mod N

Large Integer Byte Order

`options` must include one of the following constants:

MODM_BIG	Large integers received as input and generated as output are big-endian (the byte at the lowest address is the most significant byte of the integer).
MODM_LITTLE	Large integers received as input and generated as output are little-endian (the byte at the lowest address is the least significant byte of the integer).

`numInts` is the number of elements in the `aInts` array. If `options` specifies MODM_MULT or MODM_EXP, `numInts` must be at least 4. If `options` specifies MODM_MOD, `numInts` must be at least 3.

`aInts` must contain the address of an array of large integer descriptors. Its elements are as follows:

- `aInts[MODM_C]` is the descriptor for C, the result. The buffer defined by `aInts[MODM_C].bytesize` and `aInts[MODM_C].buffer` must be large enough to hold the result of the operation. `aInts[MODM_C].bitsize` is not used.
- `aInts[MODM_N]` is the descriptor for N, the modulus.
- `aInts[MODM_A]` is the descriptor for A, the first (or only) operand. If `options` specifies MODM_MULT or MODM_EXP, A must be less than N.
- `aInts[MODM_B]` is the descriptor for B, the second operand. If `options` specifies MODM_MULT, B must be less than N. If `options` specifies MODM_MOD, `aInts[MODM_B]` is not used.

pMsgID determines whether the function is performed synchronously or asynchronously.

- If pMsgID is NULL, the function is performed synchronously. The call does not return until the requested operation is complete.
- If pMsgID is not NULL, the function is perform asynchronously. The call returns as soon as a message has been sent to the PKA Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the aInts array or the buffers referenced by the elements of that array before the operation is complete.

Output

On successful exit from this routine:

If pMsgID is NULL, aInts[MODM_C].bitsize contains the number of bits in the result and aInts[MODM_C].buffer contains the value of the result. If options specifies MODM_BIG, the value is in big-endian order. If options specifies MODM_LITTLE, the value is in little-endian order. See “Large Integers” on page 3-61 for a description of the format of the value.

If pMsgID is not NULL, *pMsgID uniquely identifies the message that was sent to the PKA Manager to initiate the desired operation. When the operation is complete, the PKA Manager sends the coprocessor application a message whose type field (Msg.h.msg_type) contains this identifier and whose first (and only) data item (Msg.msg_data[0]) contains the return code generated by the routine. If the operation was successful, aInts[MODM_C].bitsize and aInts[MODM_C].buffer contain the result as previously described. The message is placed on the default CP/Q message queue for the task that called sccModMathAsync.

Return Codes

Common return codes generated by this routine are:

PKAGood (i.e., 0)	The operation was successful.
PKABadParm	An argument is not valid. (for example, options does not specify MODM_MULT, MODM_EXP, or MODM_MOD or specifies both MODM_BIG and MODM_LITTLE) or an invalid operation was requested (that is, 0^0 mod N).
PKANoSpace	The operation failed due to lack of space.
PKABadAddr	One of the large integers supplied as inputs is invalid (for example, bitsize or bytesize exceeds the maximum or buffer is NULL).
PKARangeOverflow	The buffer provided to hold the result of the operation is not large enough.

Random Number Generator Functions

The functions described in this section allow a coprocessor application to request services from the RNG Manager, which obtains random bits from the coprocessor's hardware noise source or from a pseudo-random number generator.

sccGetRandomNumber - Generate Random Number

sccGetRandomNumber generates a 64-bit random number based on a hardware noise source. The bits meet the standards described in FIPS Publication 140-1, section 4.11.

Function Prototype

```
long sccGetRandomNumberAsync(unsigned char  *pRandom,
                             unsigned long   options,
                             unsigned long  *pMsgID);

#define sccGetRandomNumber(pr,opt)  sccGetRandomNumberAsync(pr,opt,NULL)
```

Input

On entry to this routine:

`pRandom` must contain the address of a variable in which an 8-byte string of bits can be stored.

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

Parity Bits

`options` must include one of the following constants:

RANDOM_RANDOM	Generate 64 bits of random data.
RANDOM_ODD_PARITY	Generate 64 bits of random data, then set or clear the least significant bit of each byte so that the number of bits set in each byte is odd.
RANDOM_EVEN_PARITY	Generate 64 bits of random data, then set or clear the least significant bit of each byte so that the number of bits set in each byte is even.

Source of Bits

`options` may include one or both of the following constants:

RANDOM_HW	Obtain random bits from the coprocessor's hardware noise source.
RANDOM_SW	Obtain random bits from a pseudo-random number generator (PRNG). The PRNG uses a FIPS-approved algorithm and is periodically reseeded with random bits from the hardware noise source. See "Reseeding the Pseudo-Random Number Generator" on page 3-67 for details.

If `options` specifies both RANDOM_HW and RANDOM_SW, the function returns a block of random bits from the coprocessor's hardware noise source if such a block is available immediately. Otherwise the function returns a block of random bits from the PRNG.

If options specifies neither RANDOM_HW nor RANDOM_SW, RANDOM_SW is assumed.

Filter DES Weak Keys

options may include RANDOM_NOT_WEAK. If this option is specified, random numbers that are weak, semi-weak, or possibly weak when used as DES keys will not be returned. The number is checked after any requested parity bits are generated. See Appendix B, “DES Weak, Semi-Weak, and Possibly Weak Keys” on page B-1 for a list of DES weak, semi-weak, or possibly weak keys.

pMsgID determines whether the function is performed synchronously or asynchronously:

- If pMsgID is NULL, the function is performed synchronously. The call does not return until a random number is available.
- If pMsgID is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the RNG Manager instructing it to obtain a random number. In this case, the caller must not modify, deallocate, or reuse any portion of the buffer referenced by pRandom before the operation is complete.

Output

On successful exit from this routine:

If pMsgID is NULL, *pRandom contains a 64-bit random number, with parity bits set as requested in options.

If pMsgID is not NULL, *pMsgID uniquely identifies the message that was sent to the RNG Manager to initiate the desired operation. When the operation is complete, the RNG Manager will send the coprocessor application a message whose type field (MSG.h.msg_type) contains this identifier and whose first (and only) data item (MSG.msg_data[0]) contains the return code generated by the routine.[53] If the operation was successful, *pRandom contains the result. The message is placed on the default CP/Q message queue for the task that called sccGetRandomNumberAsync.

Notes

Output Buffer Must Be Writeable

If the memory referenced by the pRandom argument is not writeable by the task that called sccGetRandomNumberAsync, the function causes that task to take an exception.

Outstanding Request Limits

A single task can only have one request for a random number outstanding at a time. This prevents a denial-of-service condition in which one requestor floods the RNG Manager with requests to the exclusion of all others. The RNG Manager also limits the number of simultaneous requests pending from all sources.

[53] The return code from the call to sccGetRandomNumberAsync indicates whether or not the initial message to the RNG Manager was successfully enqueued.

Reseeding the Pseudo-Random Number Generator

sccGetRandomNumberAsync reseeds the PRNG with random bits from the coprocessor's hardware noise source if

- `options` specifies both RANDOM_HW and RANDOM_SW *and* there are exactly three 8-byte blocks of random bits from the coprocessor's hardware noise source available. The random bits returned in this case are then taken from the PRNG.
- `options` specifies RANDOM_SW (but not RANDOM_HW) *and* the number of pseudo-random numbers generated since the PRNG was last reseeded is a multiple of eight *and* there are at least three 8-byte blocks of random bits from the coprocessor's hardware noise source available.

Return Codes

Common return codes generated by this routine are:

random_success (i.e., 0)	The operation was successful.
no_random_generator	The RNG Manager is not present in the system
random_invalid	The `options` argument is not valid.
random_already_waiting	The task that called sccGetRandomNumberAsync has already requested a random number and that request is still outstanding.
random_Qfull	The maximum number of simultaneous requests is pending.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccTestRandomNumber - Test Random Number Generator (4758 Model 002 only)

Note

This function is not available on the 4758 model 001.

sccTestRandomNumber verifies that the bits returned by sccGetRandomNumber meet the standards described in FIPS Publication 140-1, section 4.11.

Function Prototype

```
long sccTestRandomNumberAsync(sccRNG_test_RB_t  *pRNGTRB,
                              unsigned long     *pMsgID);

#define sccTestRandomNumber(p)  sccTestRandomNumberAsync(p,NULL)
```

Input

On entry to this routine:

`pRNGTRB` must contain the address of a RNG test request block structure whose fields are initialized as follows:

- `options` determines whether output from the coprocessor's hardware noise source or from the pseudo-random number generator (PRNG) is tested. `options` must be either RNG_TEST_HRNG (test hardware noise source) or RNG_TEST_PRNG (test PRNG).

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until the test is complete.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the RNG Manager instructing it to perform the test.

Output

On successful exit from this routine:

If `pMsgID` is NULL, the test is complete. See "Interpretation of Test Results" on page 3-69 for details.

If `pMsgID` not NULL, `*pMsgID` uniquely identifies the message that was sent to the RNG Manager to initiate the desired operation. When the operation is complete, the RNG Manager will send the coprocessor application a message whose type field (`MSG.h.msg_type`) contains this identifier and whose first (and only) data item (`MSG.msg_data[0]`) contains the return code generated by the routine.[54] The message is placed on the default CP/Q message queue for the task that called sccTestRandomNumberAsync.

[54] The return code from the call to sccTestRandomNumberAsync indicates whether or not the initial message to the RNG Manager was successfully enqueued.

Notes

Interpretation of Test Results

If the high-order bit of the return code from the function is zero, all tests were completed. If the return code is zero, all tests passed. Other possibilities are:

- `(rc & 1) != 0`: The ratio of one bits to zero bits in a large sample of random bits is not within allowable limits.
- `(rc & 2) != 0`: The sum of the squares of the number of times each hex digit appears in a large sample of random bits is not within allowable limits.
- `(rc & 4) != 0`: The number of times consecutive bits of the same value appear in sequence and the length of such sequences in a large sample of random bits is not within allowable limits.

Return Codes

Common return codes generated by this routine are:

random_success (i.e., 0)	The operation was successful.
no_random_generator	The RNG Manager is not present in the system
random_invalid	The `options` argument is not valid.
random_already_waiting	The task that called sccGetRandomNumberAsync has already requested a random number and that request is still outstanding.
random_Qfull	The maximum number of simultaneous requests is pending.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference*

Nonvolatile Memory Functions

The functions described in this section allow a coprocessor application to request services from the Program Proprietary Data (PPD) Manager, which controls access to the coprocessor's flash and battery-backed RAM (BBRAM) storage.

Information saved in nonvolatile memory is not lost when the coprocessor is rebooted or when it is removed from its host.

The coprocessor clears BBRAM to zeros if it detects an attempt to tamper with the coprocessor hardware. If is therefore unnecessary (although permissible) to encrypt data stored in BBRAM. The coprocessor does *not* clear flash memory if it detects a tamper event, however, so sensitive data stored in flash should always be encrypted. An application may encrypt the data itself or may direct the PPD Manager to perform the encryption.

Information saved in nonvolatile memory should be saved in flash whenever possible because the coprocessor has a very limited amount of BBRAM. However, items that are frequently updated should be placed in BBRAM since writing to BBRAM is faster than writing to flash and since there is a limit to the number of times a particular flash memory cell can be written.

Information may be written to nonvolatile memory as often as required, but for reasons of efficiency information should be read from nonvolatile memory only infrequently. An application that needs to read information in nonvolatile memory frequently should keep an updated copy of the information in (regular) RAM and refer to the copy.

Names and Namespaces

Associated with each block of data (or item) saved in nonvolatile memory is a name chosen by the application that owns the data. The name is assigned when the item is first saved or when space to hold the item is allocated and subsequent requests to read, write, update or delete the item refer to it by name. The interface to the PPD Manager defines the `ppd_name_t` type to hold a name. An item of type `ppd_name_t` is eight bytes long. Names stored in a variable of type `ppd_name_t` should be padded if necessary to occupy the entire variable.

The PPD Manager maintains a separate namespace for each coprocessor application that uses nonvolatile memory.[55] All the tasks within an application share the application's namespace. Thus, an item written by one application cannot overwrite an item owned by another application, even if both applications use the same name to refer to the items, but one task in an application can manipulate an item originally saved by another task in the same application.

[55] That is, each CP/Q process has a separate namespace.

sccQueryPPDSpace - Count Free Space in Nonvolatile Memory

sccQueryPPDSpace determines the amount of free space in the coprocessor's flash memory or BBRAM. The value this function returns does not include space occupied by deleted items that has not yet been reclaimed by the PPD Manager.

Function Prototype

```
long sccQueryPPDSpace(unsigned long  *pSpace,
                      unsigned long   options);
```

Input

On entry to this routine,

`pSpace` must contain the address of a variable in which an item of type `unsigned long` can be stored.

`options` determines whether the amount of free space in flash or the amount of free space in BBRAM is returned and must be either PPD_FLASH (flash) or PPD_BBRAM (BBRAM).

Output

On successful exit from this routine,

`*pSpace` contains the number of bytes of flash that are unused if `options` is PPD_FLASH and the number of bytes of BBRAM that are unused if `options` is PPD_BBRAM. Since other applications may use nonvolatile memory, this number is a snapshot and may not reflect the amount of space actually available.

Notes

Deleted Items and Free Space

The amount of free space returned by sccQueryPPDSpace does not include space occupied by deleted items that has not yet been reclaimed by the PPD Manager.

Items Saved in BBRAM Use Free Space in Flash

Any time a new item is stored in nonvolatile memory (flash or BBRAM) using sccSavePPD or sccCreate4UpdatePPD a certain amount of flash memory is allocated to hold a directory entry for the item.

Return Codes

Common return codes generated by this routine are:

PPDGood (i.e., 0)	The operation was successful.
PPD_NOT_AUTHORIZED	The coprocessor application is not authorized to request services from the PPD Manager (for example, because it has not called sccSignOn).

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference*

sccCreate4UpdatePPD - Allocate Space in Nonvolatile Memory

sccCreate4UpdatePPD allocates a block of BBRAM and optionally stores an item (a block of data) in the allocated space.

Function Prototype

```
long sccCreate4UpdatePPDAsync(ppd_name_t       name,
                              void            *pbuffer
                              unsigned long    len
                              unsigned long   *pMsgID);
#define sccCreate4UpdatePPD(n,p,l) sccCreate4UpdatePPDAsync(n,p,l,NULL)
```

Input

On entry to this routine,

`name` is the name assigned to items stored in the block of BBRAM the function allocates. See “Names and Namespaces” on page 3-71 for details. If the name is already in use, the PPD Manager deletes the item with which the name is associated and reassigns the name.

`pBuffer` may contain the address of a block of data to be stored in BBRAM or may be NULL. See the description of the `len` argument for a discussion of how `pBuffer` is used.

`len` is the length in bytes of the block of BBRAM to allocate. If `pBuffer` is not NULL, the PPD Manager copies `len` bytes of data from `pBuffer` to BBRAM. If `pBuffer` is NULL, the PPD Manager fills the block of BBRAM with binary zeros.

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until the block of BBRAM has been allocated and the item referenced by `pBuffer` (if `pBuffer` is not NULL) has been saved in BBRAM.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the PPD Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the block of data referenced by `pBuffer` (if `pBuffer` is not NULL) before the operation is complete.

Output

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

If `pMsgID` is NULL, `len` bytes of BBRAM have been allocated and the item referenced by `pBuffer` (if `pBuffer` is not NULL) has been saved in BBRAM.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the PPD Manager to initiate the desired operation. When the operation is complete, the PPD Manager sends the coprocessor application a message whose type field (`MSG.h.msg_type`) contains this identifier and whose first (and only) data item (`MSG.msg_data[0]`) contains the return code generated by the routine.[56] The

[56] The return code from the call to sccCreate4UpdatePPDAsync indicates whether or not the initial message to the PPD Manager was successfully enqueued.

message is placed on the default CP/Q message queue for the task that called sccCreate4UpdatePPDAsync.

Notes

Effects on Flash

This function always writes to flash (to create or delete (or both) a directory entry).

Garbage Collection

The PPD Manager may need to reclaim space in flash or BBRAM (or both) occupied by items that have been deleted in order to create the new item. This activity is transparent to the caller except for the additional time it requires.

Atomic Operation

This function is an atomic operation - it either succeeds in full or fails without saving any part of the data block referenced by `pBuffer` (if `pBuffer` is not NULL) or changing an existing item with the same name.

Faults Stop System

If the PPD Manager encounters a hardware problem when writing to flash memory or to BBRAM the PPD Manager traps to the CP/Q fault handler (which stops the system).

Return Codes

Common return codes generated by this routine are:

PPDGood (i.e., 0)	The operation was successful.
PPD_NOT_AUTHORIZED	The coprocessor application is not authorized to request services from the PPD Manager (for example, because it has not called sccSignOn).
PPD_NO_SPACE	There is not enough space in the requested region of nonvolatile memory to hold the data block to be stored.
PPD_NO_DIR_SPACE	There is not enough space in flash memory to hold the directory entry for the data block to be stored.
PPD_NO_DES	The caller asked that the item be encrypted but there is no DES Manager installed in the system.
PPD_ILLEGAL_MEM	The buffer defined by `pBuffer` and `len` is not readable.
PPD_NO_PRIVS	The buffer defined by `pBuffer` and `len` is not readable by the calling task.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccSavePPD - Store Item in Nonvolatile Memory

sccSavePPD stores an item (a block of data) in flash memory or BBRAM.

This function always writes to flash, even when the item is stored in BBRAM. See “sccUpdatePPD - Update Item in BBRAM” on page 3-78 for an alternative that does not.

Function Prototype

```
long sccSavePPDAsync(ppd_name_t      name,
                     void           *pBuffer,
                     unsigned long   len,
                     unsigned long   options,
                     unsigned long  *pMsgID);

#define sccSavePPD(n,b,l,o) sccSavePPDAsync(n,b,l,o,NULL)
```

Input

On entry to this routine:

`name` is the name assigned to the item. See “Names and Namespaces” on page 3-71 for details. If an item with this name does not exist, the PPD Manager allocates the required amount of space and creates the item. If an item with the same name does exist, the PPD Manager replaces it with the block of data referenced by `pBuffer`.

`pBuffer` must contain the address of the block of data that is to be saved.

`len` is the length in bytes of the block of data referenced by `pBuffer`.

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

Nonvolatile Memory Region

`options` must include one of the following constants:

PPD_FLASH	Store item in flash memory.
PPD_BBRAM	Store item in BBRAM.

Items saved in nonvolatile memory should be saved in flash whenever possible because the coprocessor has a very limited amount of BBRAM.

Encryption Options

The PPD Manager can be directed to encrypt the block of data before it is saved in nonvolatile memory.[57] `options` may include one of the following constants:

PPD_SINGLE	Encrypt using DES (CBC mode) and a single key.
PPD_TRIPLE	Encrypt using DES (CBC mode) and three distinct keys.
PPD_NONE	Do not encrypt.
PPD_USE_PREV	Encrypt using the method that was most recently used to encrypt the item.

[57] Sensitive data stored in flash should always be encrypted, either by the application or by the PPD Manager.

If PPD_USE_PREV is specified and `name` does not refer an item already saved in nonvolatile memory, no encryption is performed.

The PPD Manager uses its own keys to encrypt the item.

Items in BBRAM that have been encrypted by the PPD Manager cannot be modified by the sccUpdatePPD function.

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until the block of data has been saved in nonvolatile memory.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the PPD Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the block of data referenced by `pBuffer` before the operation is complete.

Output

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

If `pMsgID` is NULL, the block of data referenced by `pBuffer` has been saved in flash memory if `options` specifies PPD_FLASH and has been saved in BBRAM if `options` specifies PPD_BBRAM.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the PPD Manager to initiate the desired operation. When the operation is complete, the PPD Manager will send the coprocessor application a message whose type field (`MSG.h.msg_type`) contains this identifier and whose first (and only) data item (`MSG.msg_data[0]`) contains the return code generated by the routine.[58] The message is placed on the default CP/Q message queue for the task that called sccSavePPDAsync.

Notes

Effect on Flash

This function always writes to flash (to create or delete [or both] a directory entry).

Garbage Collection

The PPD Manager may need to reclaim space in flash or BBRAM (or both) occupied by items that have been deleted in order to create the new item. This activity is transparent to the caller except for the additional time it requires.

Atomic Operation

This function is an atomic operation - it either succeeds in full or fails without saving any part of the data block or changing an existing item with the same name.

[58] The return code from the call to sccSavePPDAsync indicates whether or not the initial message to the PPD Manager was successfully enqueued.

Faults Stop System

If the PPD Manager encounters a hardware problem when writing to flash memory or to BBRAM, the PPD Manager traps to the CP/Q fault handler (which stops the system).

Return Codes

Common return codes generated by this routine are:

PPDGood (i.e., 0)	The operation was successful.
PPD_NOT_AUTHORIZED	The coprocessor application is not authorized to request services from the PPD Manager (for example, because it has not called sccSignOn).
PPD_NO_SPACE	There is not enough space in the requested region of nonvolatile memory to hold the data block to be stored.
PPD_NO_DIR_SPACE	There is not enough space in flash memory to hold the directory entry for the data block to be stored.
PPD_NO_DES	The caller asked that the item be encrypted but there is no DES Manager installed in the system.
PPD_ILLEGAL_MEM	The buffer defined by `pBuffer` and `len` is not readable.
PPD_NO_PRIVS	The buffer defined by `pBuffer` and `len` is not readable by the calling task.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccUpdatePPD - Update Item in BBRAM

sccUpdatePPD updates an arbitrary portion of an item (a block of data) in BBRAM.

Function Prototype

```
long sccUpdatePPDAsync(ppd_name_t      name,
                       void           *pBuffer,
                       unsigned long   len,
                       unsigned long   offset,
                       unsigned long  *pMsgID);

#define sccUpdatePPD(n,b,l,o) sccUpdatePPD(n,b,l,o,NULL)
```

Input

On entry to this routine:

`name` is the name of the item to be updated. See "Names and Namespaces" on page 3-71 for details. The item must reside in BBRAM and must not have been encrypted by the PPD Manager.[59]

`pBuffer` contains the address of a buffer of data that is to be written over a portion of the item.

`len` is the length in bytes of the buffer referenced by `pBuffer`.

`offset` is the offset within the item at which the first byte of the buffer referenced by `pBuffer` is to be written.

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until the item has been updated by the contents of the bufffer referenced by `pBuffer`.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the PPD Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the block of data referenced by `pBuffer` before the operation is complete.

Output

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

If `pMsgID` is NULL, `len` bytes of the item named by `name` (beginning with the byte at `offset`) have been replaced by the contents of the buffer referenced by `pBuffer`.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the PPD Manager to initiate the desired operation. When the operation is complete, the PPD Manager sends the coprocessor application a message whose type field (`MSG.h.msg_type`) contains this identifier and whose first (and only) data item (`MSG.msg_data[0]`) contains the return code generated by the routine.[60] The

[59] That is, if the item was created by a call to sccSavePPD, the options on the call must have been PPD_BBRAM | PPD_NONE.

[60] The return code from the call to sccUpdatePPDAsync indicates whether or not the initial message to the PPD Manager was successfully enqueued.

message is placed on the default CP/Q message queue for the task that called sccUpdatePPDAsync.

Notes

Effect on Flash

This function does not write to flash.

Nonatomic Operation

This function is *not* an atomic operation. If the coprocessor is reset after the operation begins but before it completes, a partial update may occur.

Faults Stop System

If the PPD Manager encounters a hardware problem when writing to BBRAM, the PPD Manager traps to the CP/Q fault handler (which stops the system).

Return Codes

Common return codes generated by this routine are:

PPDGood (i.e., 0)	The operation was successful.
PPD_NOT_AUTHORIZED	The coprocessor application is not authorized to request services from the PPD Manager (for example, because it has not called sccSignOn).
PPD_NOT_FOUND	The coprocessor application does not own an item in nonvolatile memory named `name`.
PPD_BAD_PARM	The length of the buffer referenced by `pBuffer` and the offset specified by `offset` extend past the end of the item to be updated.
PPD_ILLEGAL_MEM	The buffer defined by `pBuffer` and `len` is not readable.
PPD_NO_PRIVS	The buffer defined by `pBuffer` and `len` is not readable by the calling task.
PPD_NOT_UPDATABLE	The item to be updated is in flash (rather than BBRAM) or was encrypted by the PPD Manager when it was last saved.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccGetPPDDir - Count Items in Nonvolatile Memory

sccGetPPDDir determines the total number of items stored in nonvolatile memory that belong to the coprocessor application of which the task that calls sccGetPPDDir is a part. The names of all such items can also be retrieved.

Function Prototype

```
long sccGetPPDDirAsync(unsigned long  *pCount,
                       void           *pBuffer,
                       unsigned long  *pLen,
                       unsigned long  *pMsgID);

#define sccGetPPDDir(pc,pb,pl) sccGetPPDDirAsync(pc,pb,pl,NULL)
```

Input

On entry to this routine:

`pCount` must contain the address of a variable in which an item of type `unsigned long` can be stored.

`pBuffer` must contain the address of a buffer in which the names of some or all of the items the calling task's application owns can be returned if this information is desired and must be NULL otherwise.

`pLen` must contain the address of a variable that contains the length in bytes of the buffer referenced by `pBuffer`. If `pBuffer` is NULL, `*pLen` must be zero.

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until the requested information has been returned.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the PPD Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the areas of memory referenced by `pCount`, `pBuffer`, or `pLen` before the operation is complete.

Output

On successful exit from this routine:

If `pMsgID` is NULL, `*pCount` is the number of items in nonvolatile memory owned by the coprocessor application of which the task that called sccSavePPDAsync is a part. If `pBuffer` is not NULL, the buffer it references contains an array of items of type `ppd_name_t`. Each entry in the array contains the name of an item owned by the coprocessor application. `*pLen` is the total length in bytes of the entries in the array (that is, `*pLen` is `sizeof(ppd_name_t)` times the number of entries in the array). `*pLen` will be less than or equal to the value of `*pLen` on entry to the routine.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the PPD Manager to initiate the desired operation. When the operation is complete, the PPD Manager will send the coprocessor application a message whose type field (`MSG.h.msg_type`) contains this identifier and whose first (and only) data item

(`MSG.msg_data[0]`) contains the return code generated by the routine.[61] If the operation was successful, `*pCount` (and `*pBuffer` and `*pLen`, if appropriate) contain the result. The message is placed on the default CP/Q message queue for the task that called sccGetPPDDirAsync.

Return Codes

Common return codes generated by this routine are:

PPDGood (i.e., 0)	The operation was successful.
PPD_NOT_AUTHORIZED	The coprocessor application is not authorized to request services from the PPD Manager (for example, because it has not called sccSignOn).
PPD_NOT_FOUND	The coprocessor application does not own any items in nonvolatile memory.
PPD_ILLEGAL_MEM	The buffer defined by `pBuffer` and `*pLen` is not readable or writeable.
PPD_NO_PRIVS	The buffer defined by `pBuffer` and `len` is not readable by the calling task.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

[61] The return code from the call to sccGetPPDDirAsync indicates whether or not the initial message to the PPD Manager was successfully enqueued.

sccGetPPDLen - Get Length of Item in Nonvolatile Memory

sccGetPPDLen obtains the length of an item stored in flash memory or BBRAM.

Function Prototype

```
long sccGetPPDLen(ppd_name_t      name,
                  unsigned long  *pLen);
```

Input

On entry to this routine:

`name` is the name of the item whose length is desired. See “Names and Namespaces” on page 3-71 for details.

`pLen` must contain the address of a variable in which an item of type `unsigned long` can be stored.

Output

On successful exit from this routine:

`*pLen` is the length in bytes of the item whose name is `name`.

Return Codes

Common return codes generated by this routine are:

PPDGood (i.e., 0)	The operation was successful.
PPD_NOT_AUTHORIZED	The coprocessor application is not authorized to request services from the PPD Manager (for example, because it has not called sccSignOn).
PPD_NOT_FOUND	The coprocessor application does not own an item in nonvolatile memory named `name`.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccGetPPD - Retrieve Item from Nonvolatile Memory

sccGetPPD reads an item (a block of data) from flash memory or BBRAM.

Function Prototype

```
long sccGetPPDAsync(ppd_name_t      name,
                    void           *pBuffer,
                    unsigned long   len,
                    unsigned long  *pMsgID);

#define sccGetPPD(n,b,l) sccGetPPDAsync(n,b,l,NULL)
```

Input

On entry to this routine:

`name` is the name of the item to retrieve. See “Names and Namespaces” on page 3-71 for details.

`pBuffer` must contain the address of a buffer to which the item can be copied.

`len` is the length in bytes of the buffer referenced by `pBuffer`.

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until the item has been retrieved from nonvolatile memory.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the PPD Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the buffer referenced by `pBuffer` before the operation is complete.

Output

On successful exit from this routine:

If `pMsgID` is NULL, the item named name has been retrieved from nonvolatile memory, decrypted (if it was encrypted by the PPD Manager when it was last saved), and copied to the buffer referenced by `pBuffer`.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the PPD Manager to initiate the desired operation. When the operation is complete, the PPD Manager will send the coprocessor application a message whose type field (`MSG.h.msg_type`) contains this identifier and whose first (and only) data item (`MSG.msg_data[0]`) contains the return code generated by the routine.[62] If the operation was successful, the item named `name` has been copied to the buffer referenced by `pBuffer`. The message is placed on the default CP/Q message queue for the task that called sccGetPPDAsync.

[62] The return code from the call to sccGetPPDAsync indicates whether or not the initial message to the PPD Manager was successfully enqueued.

Notes

sccGetPPD and sccUpdatePPD

There is an asymmetry between the sccUpdatePPD function, which can be directed to write only a portion of a block of data saved in BBRAM, and the sccGetPPD function, which can only read an entire block. This asymmetry is intentional. The latest copy of data that has been written to BBRAM will presumably be available in regular RAM and should be read from regular RAM - using sccGetPPD to do so would be a much more expensive operation.

In other words, the intent is that changes be written to PPD as often as desired, but that data be read from PPD only infrequently.

Return Codes

Common return codes generated by this routine are:

PPDGood (i.e., 0)	The operation was successful.
PPD_NOT_AUTHORIZED	The coprocessor application is not authorized to request services from the PPD Manager (for example, because it has not called sccSignOn).
PPD_NOT_FOUND	The coprocessor application does not own an item in nonvolatile memory named `name`.
PPD_SMALL_BUF	The buffer defined by `pBuffer` and `len` is not large enough to hold the desired item.
PPD_ILLEGAL_MEM	The buffer defined by `pBuffer` and `len` is not writeable.
PPD_NO_PRIVS	The buffer defined by `pBuffer` and `len` is not readable by the calling task.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccDeletePPD - Delete Item from Nonvolatile Memory

sccDeletePPD deletes an item (a block of data) from flash memory or BBRAM.

Function Prototype

```
long sccDeletePPDAsync(ppd_name_t      name,
                       unsigned long  *pMsgID);

#define sccDeletePPD(n)  sccDeletePPDAsync(n,NULL)
```

Input

On entry to this routine:

`name` is the name of the item to delete. See “Names and Namespaces” on page 3-71 for details.

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until the item has been deleted from nonvolatile memory.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the PPD Manager instructing it to perform the desired operation.

Output

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

If `pMsgID` is NULL, the item named `name` has been deleted from nonvolatile memory.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the PPD Manager to initiate the desired operation. When the operation is complete, the PPD Manager will send the coprocessor application a message whose type field (`MSG.h.msg_type`) contains this identifier and whose first (and only) data item (`MSG.msg_data[0]`) contains the return code generated by the routine.[63] The message is placed on the default CP/Q message queue for the task that called sccDeletePPDAsync.

Notes

Item Deleted Even if Caller Terminates or Coprocessor Resets

Once the PPD Manager has accepted a request to delete an item, the item will be deleted even if the task or application that sent the request terminates without waiting for a response and even if the operation is interrupted by a coprocessor reset. However, an application cannot easily determine the point at which the request has been accepted—the PPD Manager must dequeue and validate the request, find the item to be deleted, and mark its directory entry in flash. If the application that sent the request terminates before these steps are complete, the call will fail and items that should have been removed may be left in nonvolatile memory. An application that is about to end should therefore use the synchronous

[63] The return code from the call to sccDeletePPDAsync indicates whether or not the initial message to the PPD Manager was successfully enqueued.

form of this call to delete any items in nonvolatile memory that are no longer needed.

Return Codes

Common return codes generated by this routine are:

PPDGood (i.e., 0)	The operation was successful.
PPD_NOT_AUTHORIZED	The coprocessor application is not authorized to request services from the PPD Manager (for example, because it has not called sccSignOn).
PPD_NOT_FOUND	The coprocessor application does not own an item in nonvolatile memory named `name`.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccDeleteAllPPD - Delete All Items from Nonvolatile Memory

sccDeleteAllPPD deletes from flash memory and BBRAM all items (blocks of data) that belong to the coprocessor application of which the task that calls sccDeleteAllPPD is a part.

Function Prototype

```
long sccDeleteAllPPDAsync(unsigned long  *pMsgID);

#define sccDeleteAllPPD()  sccDeleteAllPPDAsync(NULL)
```

Input

On entry to this routine:

`pMsgID` determines whether the function is performed synchronously or asynchronously:

- If `pMsgID` is NULL, the function is performed synchronously. The call does not return until all items belonging to the calling application have been deleted from nonvolatile memory.
- If `pMsgID` is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the PPD Manager instructing it to perform the desired operation.

Output

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

If `pMsgID` is NULL, all items belonging to the calling application have been deleted from nonvolatile memory.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the PPD Manager to initiate the desired operation. When the operation is complete, the PPD Manager will send the coprocessor application a message whose type field (`MSG.h.msg_type`) contains this identifier and whose first (and only) data item (`MSG.msg_data[0]`) contains the return code generated by the routine.[64] The message is placed on the default CP/Q message queue for the task that called sccDeleteAllPPDAsync.

Notes

Items Deleted Even if Caller Terminates or Coprocessor Resets

Once the PPD Manager has accepted a request to delete all items belonging to an application, the items will be deleted even if the task or application that sent the request terminates without waiting for a response and even if the operation is interrupted by a coprocessor reset. However, an application cannot easily determine the point at which the request has been accepted—the PPD Manager must dequeue and validate the request, find the items to be deleted, and mark their directory entries in flash. If the application that sent the request terminates before these steps are complete, the call will fail and items that should have been removed may be left in nonvolatile memory. An application that is about to end

[64] The return code from the call to sccDeleteAllPPDAsync indicates whether or not the initial message to the PPD Manager was successfully enqueued.

should therefore use the synchronous form of this call to delete any items in nonvolatile memory that are no longer needed.

Return Codes

Common return codes generated by this routine are:

PPDGood (i.e., 0)	The operation was successful.
PPD_NOT_AUTHORIZED	The coprocessor application is not authorized to request services from the PPD Manager (for example, because it has not called sccSignOn).
PPD_NOT_FOUND	The coprocessor application does not own any items in nonvolatile memory.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

Configuration Functions

The functions described in this section allow a coprocessor application to interact with the SCC Manager and other CP/Q++ device managers to configure certain processor features or obtain information about the coprocessor.

Privileged Operations

Some of the functions described in this section can only be performed by the first application that is loaded and run when the coprocessor boots. That application is deemed to "own" the coprocessor. Refer to the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer's Toolkit Guide* for an explanation of how to control the order in which applications are loaded when the coprocessor boots.

sccGetConfig - Get Coprocessor Configuration

sccGetConfig obtains information about the coprocessor from the SCC Manager.

Function Prototype

```
long sccGetConfig(sccAdapterInfo_t   *pInfo
                  unsigned long      *pLength);
```

Input

On entry to this routine:

`pInfo` must contain the address of a buffer into which information about the coprocessor is to be stored.

`*pLength` must be the length in bytes of the buffer referenced by `pInfo`.

Output

On successful exit from this routine:

`*pInfo` contains as much information about the coprocessor as could be returned in the buffer provided. If the buffer was sufficiently large, this will be a full `sccAdapterInfo_t` structure whose fields are set as indicated below. If the buffer was too small, the structure will be truncated.

`*pLength` contains the length of the full `sccAdapterInfo_t` structure. If this is larger than the buffer originally provided, the application can acquire a suitably-sized buffer and repeat the call.

The fields of the `sccAdapterInfo_t` structure are set as follows (all constants are current as of the date of publication of this document):

- `sid` identifies the structure and contains its length in bytes. `sid.ID` is STRUCT_sccAdapterInfo.
- `AMCC_EEPROM` contains a copy of the values copied into the AMCC S5933 PCI Controller chip when power is first supplied to the coprocessor.
- `VPD` contains the coprocessor Vital Product Data. Its fields are set as follows:
  - `signature` contains the null-terminated string "VPD".
  - `vpd_length` is the length in bytes of the `VPD` field.
  - `crc` contains a CRC covering the fields in `VPD` from `pn_tag` to `ds`, inclusive.
  - `pn_tag` contains the characters "*PN".

  - pn_length is 6 (the number of bytes in the pn_tag, pn_length, and pn fields divided by 2).
  - pn contains the part number for the IBM 4758 PCI Cryptographic Coprocessor (for example, 69H6479) and is padded with blanks to its full length.
  - ec_tag contains the characters "*EC".
  - ec_length is 6 (the number of bytes in the ec_tag, ec_length, and ec fields divided by 2).
  - ec contains the engineering change level for the IBM 4758 PCI Cryptographic Coprocessor (for example, C75554C) and is padded with blanks to its full length.
  - sn_tag contains the characters "*SN".
  - sn_length is 6 (the number of bytes in the sn_tag, sn_length, and sn fields divided by 2).
  - sn contains the serial number for the cryptographic coprocessor (of the form 41-xxxxx) and is padded with blanks to its full length.
  - fn_tag contains the characters "*FN".
  - fn_length is 6 (the number of bytes in the fn_tag, fn_length, and fn fields divided by 2).
  - fn contains the FRU number for the IBM 4758 PCI Cryptographic Coprocessor (for example, 09J8193) and is padded with blanks to its full length.
  - mf_tag contains the characters "*MF".
  - mf_length is 6 (the number of bytes in the mf_tag, mf_length, and mf fields divided by 2).
  - mf identifies the location at which the cryptographic coprocessor was manufactured and is padded with blanks to its full length.
  - ds_tag contains the characters "*DS".
  - ds_length is 6 (the number of bytes in the ds_tag, ds_length, and ds fields divided by 2).
  - ds contains a description of the cryptographic coprocessor (IBM 4758 PCI Cryptographic Coprocessor) and is padded with blanks to its full length.
  - reserved contains garbage.
- EC_Level is not used.
- POST_Version indicates which version of the coprocessor power-on self test microcode is installed. This microcode operates in two phases (POST0 and POST1), so POST_Version contains two fields.
- MiniBoot_Version indicates which version of the coprocessor microcode that initializes the coprocessor operating system and controls updates to software in flash memory is installed. This microcode also operates in two phases (MiniBoot0 and MiniBoot1), so MiniBoot_Version contains two fields.
- OS_Name contains the characters "CP/Q++".
- OS_Version indicates which version of the operating system is installed.
- CPU_Speed is the speed in megahertz of the coprocessor CPU.
- HardwareOptions provides information about the coprocessor hardware:
  - HardwareOptions.DES_level indicates the speed of the coprocessor's DES chip. Possible values are:
    - CMOS_DES_ABSENT - No DES chip is present.
    - CMOS_DES_25MB - Peak encryption/decryption rate is 25MB/second.
      CMOS_DES_30MB - Peak encryption/decryption rate is 30MB/second.
  - HardwareOptions.RSA_level indicates the RSA keylength supported by the coprocessor's large-integer modular math hardware. Possible values are:
    - CMOS_RSA_ABSENT - No modular math hardware is present.
    - CMOS_RSA_1024 - 1024 bits.

  - CMOS_RSA_2048 - 2048 bits.
- `HardwareStatus` contains the current state (active high) of the hardware tamper bits (refer to *scc_types.h*).
- `AdapterID` is a unique serial number incorporated in the coprocessor chip that implements the real-time clock and the BBRAM. It can be used to distinguish the physical coprocessor card from all others but is unrelated to the serial number in VPD.sn.
- `flashSize` is the size of the coprocessor's flash memory. The unit of measurement is 64K, that is, `flashSize` == 1 implies 64K of EEPROM.
- `bbramSize` is the size of the coprocessor's BBRAM. The unit of measurement is 1K, that is, `bbramSize` == 16 implies 16K BBRAM.
- `dramSize` is the size of the coprocessor's regular (non-battery-backed) random access memory (RAM). The unit of measurement is 1K, that is, `dramSize` = 128 implies 128K RAM.
- `reserved` contains garbage.

Notes

Format of Return Information May Change

The format and type of information returned by this function may be altered or extended in the future. The returned information will always include an `sccAdapterInfo_t` structure (although the definition of the `sccAdapterInfo_t` data type may change over time). The `sccAdapterInfo_t` structure may be followed by additional structures, each of which will contain a structure of type `structId_t` as its first element. A `structID_t` structure contains a code that identifies its parent structure and the length of the parent structure.

Return Codes

Common return codes generated by this routine are:

SCCGood (i.e., 0)	The operation was successful.
QSVCsmallbuff	The buffer provided was not large enough to receive the entire structure returned by the SCC Manager. The returned structure has been truncated.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccSetClock - Set Coprocessor Time-Of-Day Clock

sccSetClock sets the coprocessor time-of-day (TOD) clock and updates the system time.

Function Prototype

```
long sccSetClock(unsigned long  day,
                 unsigned long  month,
                 unsigned long  year,
                 unsigned long  hour,
                 unsigned long  minute,
                 unsigned long  second);
```

Input

This function forwards its first three arguments to CPSetDate and its last three arguments to CPSetTime. Refer to the descriptions of these functions in the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for an explanation of what these arguments contain and how they are used.

Output

On successful exit from this routine, the coprocessor TOD clock and the system time maintained by CP/Q have been set to the requested time and date.

Notes

Privileged Operation

This function is privileged and can only be performed by the application that owns the coprocessor. See “Privileged Operations” on page 3-89 for details.

Return Codes

Common return codes generated by this routine are:

PPDGood (i.e., 0)	The operation was successful.
PPD_NOT_AUTHORIZED	The caller is not authorized to set the clock (for example, because it does not own the coprocessor or has not called sccSignOn).

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccClearILatch - Clear Coprocessor Intrusion Latch

The IBM 4758 PCI Cryptographic Coprocessor hardware provides an input pin (the "intrusion latch") to which an external device can be connected. For example, the user might connect a sensor that detects unauthorized attempts to open the case of the host in which the coprocessor is installed.

The application that owns the coprocessor can determine the state of the intrusion latch by calling sccGetConfig. The host device driver can also determine the state of the intrusion latch (see Chapter 4, "Coprocessor Interface for Host Device Drivers" on page 4-1 for details). Neither the coprocessor operating system nor the microcode that monitors attempts to compromise the coprocessor's secure environment take any action when the intrusion latch is triggered. sccClearILatch resets the coprocessor intrusion latch.

Function Prototype

```
long sccClearILatch(void);
```

Input

This function takes no input.

Output

On successful exit from this routine, the coprocessor intrusion latch (defined by HW_ILATCH in *scctypes.h*) is reset.

Notes

Privileged Operation

This function is privileged and can only be performed by the application that owns the coprocessor. See "Privileged Operations" on page 3-89 for details.

Return Codes

Common return codes generated by this routine are:

SCCGood (i.e., 0)	The operation was successful.
PPD_NOT_AUTHORIZED	The caller is not authorized to clear the coprocessor intrusion latch (for example, because it does not own the coprocessor or has not called sccSignOn).

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccClearLowBatt - Clear Coprocessor Low Battery Warning Latch

The IBM 4758 PCI Cryptographic Coprocessor hardware includes two batteries that allow the coprocessor to detect certain attempts to compromise its physical integrity. If the batteries are allowed to drain completely, the coprocessor clears its secrets and resets itself as if it had detected an attempt to tamper with the secure hardware. The coprocessor therefore monitors the battery voltage and triggers the low battery warning latch if it drops below a certain value[65].

The application that owns the coprocessor can determine the state of the low battery warning latch by calling sccGetConfig. The host device driver can also determine the state of the low battery warning latch (see Chapter 4, “Coprocessor Interface for Host Device Drivers” on page 4-1 for details). Neither the coprocessor operating system nor the microcode that monitors attempts to compromise the coprocessor’s secure environment takes any action when the low battery warning is triggered. sccClearLowBatt resets the coprocessor low battery warning latch.

Function Prototype

```
long sccClearLowBatt(void);
```

Input

This function takes no input.

Output

On successful exit from this routine, the coprocessor low battery warning latch (defined by HW_BATTERYLOW in *scctypes.h*) is reset.

Notes

Privileged Operation

This function is privileged and can only be performed by the application that owns the coprocessor. See “Privileged Operations” on page 3-89 for details.

Return Codes

Common return codes generated by this routine are:

SCCGood (i.e., 0)	The operation was successful.
PPD_NOT_AUTHORIZED	The caller is not authorized to clear the coprocessor low battery latch (for example, because it does not own the coprocessor or has not called sccSignOn).

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

[65] The precise value is chosen to provide a reasonable expectation that the low battery warning latch will be triggered at least one month before the batteries are exhausted.

Outbound Authentication Functions

The functions described in this section allow a coprocessor application to request services from the Outbound Authentication (OA) Manager, which supports cryptographic operations and data structures that allow the coprocessor application to authenticate itself to another agent[66] and to engage in a wide range of cryptographic protocols. In particular, a coprocessor application can use these functions to:

- Prove to another agent that the coprocessor on which the application is running has not been tampered with
- Provide another agent a list of all the software that has ever been loaded on the coprocessor that could have revealed the application's secrets or compromised the authentication scheme
- Report in a manner that cannot be forged (unless the authentication scheme has been compromised) the status of the coprocessor, including its serial number and the identity of the software it contains
- Perform general cryptographic operations (encryption, decryption, signing, and verification) and engage in cryptographic protocols (for example, key exchange) using keys whose validity is assured by the authentication scheme

The remainder of this introduction describes certain aspects of the coprocessor architecture that form the basis of the authentication scheme and provides an overview of the authentication scheme. For a thorough overview of the coprocessor's security goals and a description of the security architecture, refer to *Building a High-Performance, Programmable Secure Coprocessor*, Research Report RC21102 published by the IBM T.J. Watson Research Center in February, 1998. A revised version of this paper appeared in *Computer Networks* 31:831-860, April 1999.

Coprocessor Architecture

The nonvolatile memory on a coprocessor is partitioned into four "segments," each of which can contain program code and sensitive data:

- Segment 0 contains one portion of "Miniboot," the most privileged software in the coprocessor. Miniboot implements, among other things, the protocols that ensure nothing is loaded into the coprocessor without the proper authorization. The code in segment 0 is in ROM.
- Segment 1 contains another portion of Miniboot. The code in segment 1 is in flash. The division of Miniboot into a ROM portion and a flash portion preserves flexibility while guaranteeing a basic level of security.
- Segment 2 contains the coprocessor operating system (CP/Q++). This code is in flash.
- Segment 3 contains the coprocessor application. This code is in flash.

A segment's sensitive data is either saved in battery-backed RAM (BBRAM) or is encrypted[67] and saved in flash. The coprocessor incorporates special hardware (independent of the CPU and whose operation cannot be affected by software) that

[66] The other agent could be an entity on a host or another coprocessor anywhere in the world or could be a newer version of the same application subsequently loaded into the same coprocessor. Authenticating to a later version of oneself demonstrates the utility of Epoch Keypairs (see "Changes to Segments 2 and 3" on page 3-97).

[67] The encryption key is saved in BBRAM.

prevents the operating system and any application (that is, code in segments 2 and 3) from modifying sensitive information in flash or reading secrets in BBRAM.

One of the data items Miniboot saves in BBRAM in segment 0 is a 32-bit "boot counter." The boot counter is initialized to zero during manufacture; the Miniboot code in segment 0 increments the boot counter each time the coprocessor boots. The authentication scheme uses the boot counter as a timestamp in many contexts.[68]

Information that identifies the code loaded in a segment is also saved in the segment. This information includes:

- The identity of the owner of the segment, that is, the party responsible for the software that is loaded in the segment. Owner identifiers are two bytes long. IBM owns segment 1 and issues an owner identifier to any party that is developing code to be loaded into segment 2. An owner of segment 2 issues an owner identifier to any party that is developing code that is to be loaded into segment 3 under the segment 2 owner's authority (that is, while the segment 2 owner owns segment 2).
- The name (an arbitrary string no longer than 80 bytes), revision number (a two-byte integer), and SHA-1 hash of the software in the segment. The hash that covers a segment is computed by the software in segment 1.

Overview of the Authentication Scheme

Initialization

During manufacture, a coprocessor generates a random RSA keypair[69] (the "Device Keypair") and exports the public key. The factory incorporates the Device Public Key into a certificate and signs the certificate using the private half of a keypair owned and controlled by the factory (an "IBM Class Root Keypair"). The coprocessor imports and saves this certificate and a certificate containing the IBM Class Root Public Key. The latter certificate is signed using the private half of a keypair owned and controlled by IBM (an "IBM Root Keypair").[70]

Updates to Segment 1

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

1. Generates a new random RSA keypair (a "Transition Keypair").
2. Incorporates the new Transition Public Key and information that identifies the new segment 1 software into a certificate and signs the certificate using the private half of the active segment 1 keypair. If this is the first time the software in segment 1 has been updated, the active segment 1 keypair is the Device Keypair. Otherwise the active segment 1 keypair is the Transition Keypair created the last time segment 1 was updated.
3. Deletes the private half of the active segment 1 keypair and makes the new Transition Keypair the active segment 1 keypair.

---

[68] The coprocessor's real-time clock cannot be used to generate timestamps because it can be changed by the application and because there is no way to synchronize it with an external clock in a reliable manner that is guaranteed to work in all scenarios.

[69] The coprocessor generates an RSA keypair or a DSA keypair, as directed by Officer 1. The IBM Officer 1 currently specifies an RSA keypair.

[70] The value of the IBM Root Public Key appears in Appendix C, "The IBM Root Public Key" on page C-1.

The result is a chain of certificates that links the IBM Class Root Certificate and the most recently created Transition Certificate. If an adversary tampers with the coprocessor, the coprocessor clears the active segment 1 private key. Any subsequent attempt to assert the coprocessor has not been tampered with fails because the adversary does not possess any of the private keys used to create the certificate chain. The adversary also does not possess the IBM Root Private Key and so cannot forge an IBM Class Root Certificate. The adversary therefore cannot sign a nonce with an existing key or create a new key that is linked to the IBM Class Root Certificate to do so.

The certificate chain also identifies every piece of software that has ever been loaded into segment 1. Although a malicious or defective program loaded into segment 1 can reveal its own Transition Private Key (and so compromise any certificates that are subsequently generated), such a program cannot mask its presence because its identity is incorporated into a certificate using a private key whose value the program never knows. Once the program's behavior is recognized, a host can treat a certificate chain that includes the program with the suspicion it warrants.[71]

Changes to Segments 2 and 3

The software in segment 1 also manipulates the certificate chain when changes are made to segment 2 or to segment 3. The specific actions segment 1 performs depend on whether the changes affect the sensitive data saved in segment 3 BBRAM. Certain operations dictate that segment 1 clear segment 3 BBRAM; others do not.[72]

The authentication scheme defines an "epoch" to be the maximum possible lifetime of a piece of sensitive data in segment 3 BBRAM. An epoch begins when an event occurs that loads runnable code into segment 3 (or leaves any code that is already in segment 3 in a runnable state) and causes segment 1 to erase the contents of segment 3 BBRAM. An epoch ends the next time segment 3 BBRAM is cleared for any reason (for example, because the software in segment 3 or the software in segment 2 has been unloaded or has been reloaded in a manner that clears BBRAM).

The authentication scheme defines a "configuration" to be a period of time during which the software in a coprocessor does not change. A configuration begins when an event occurs that changes the software in any segment and that either loads runnable code into segment 3 or leaves any code that is already in segment 3 in a runnable state. A configuration ends the next time the software in any segment changes or when the code in segment 3 is no longer runnable. A configuration also ends if the epoch in which the configuration started ends.[73]

---

71 Note that although a malicious program can attempt to hide by adopting the name and revision number of a benign program, the SHA-1 hash that is saved in segment 1 is computed by the previous occupant of segement 1 and cannot be forged.

72 Refer to "Appendix F. Using Signer and Packager" in the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer's Toolkit Guide* for a discussion of which operations clear segment 3 BBRAM.

73 The notions of "epoch" and "configuration" are actually more general than these definitions indicate. For example, certain actions can cause the sensitive data in segment 3 BBRAM to be erased without affecting any sensitive data in segment 2 BBRAM. In that case, the current "segment 3" epoch ends while the current "segment 2" epoch continues. Similarly, a change to the software in segment 3 begins a new "segment 3" configuration but does not affect the current "segment 2" configuration. The only context in which these distinctions might be of interest to an application on the host is when interpreting the `epoch_start`, `config_start`, and `config_count` fields in a layer name. See "Layer Names and Layer Descriptors" on page 3-114 for details.

An application can ask the OA Manager to create one or more keypairs the application can use to perform general cryptographic operations. The application can specify that the private half of the keypair in question is to be used only during the current configuration (a “Configuration Keypair”) or is to be used for the duration of the current epoch (an “Epoch Keypair”). The OA Manager also creates a certificate for the keypair and signs it using the private half of an “Operating System” keypair.

Configuration Start

When a configuration begins, the software in segment 1 creates:

1. An operating system keypair
2. A certificate that contains the public half of the keypair

The certificate is signed using the private half of the active segment 1 keypair.

Configuration End

When a configuration ends, the software in segment 1 erases:

1. The private half of any configuration keypairs the application has caused to be created
2. The private half of the current operating system keypair

The certificates for such keypairs are retained (since there may still be sensitive data that was encrypted using the private half of one of the keypairs) but they are marked “inactive.”

Epoch End

When an epoch ends, the software in segment 1 erases:

1. Any configuration keypairs and epoch keypairs the application has caused to be created
2. Any operating system keypairs that have been created

The software in segment 2 subsequently erases the certificates that contain the public halves of the keypairs that the software in segment 1 erased.

Examples

Figure 3-1 shows the certificate chain after an application has been loaded into a coprocessor for the first time and has asked the OA Manager to create an Epoch Keypair. Figure 3-1 also indicates which certificates contain a public key whose corresponding private key is also stored on the coprocessor (the Device Private Key is deleted when the first Transition Keypair is created).

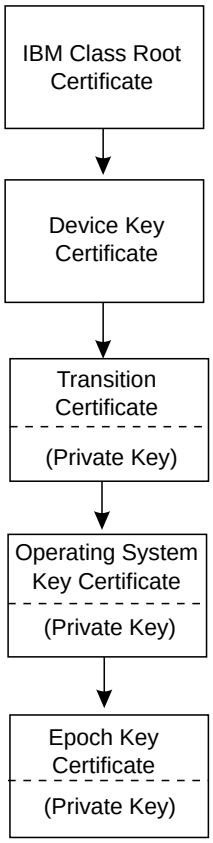

Figure 3-1. Initial Certificate Chain

The coprocessor application then asks the OA Manager to create a Configuration Keypair. The OA Manager adds a new Configuration Key Certificate to the chain, as shown in Figure 3-2.

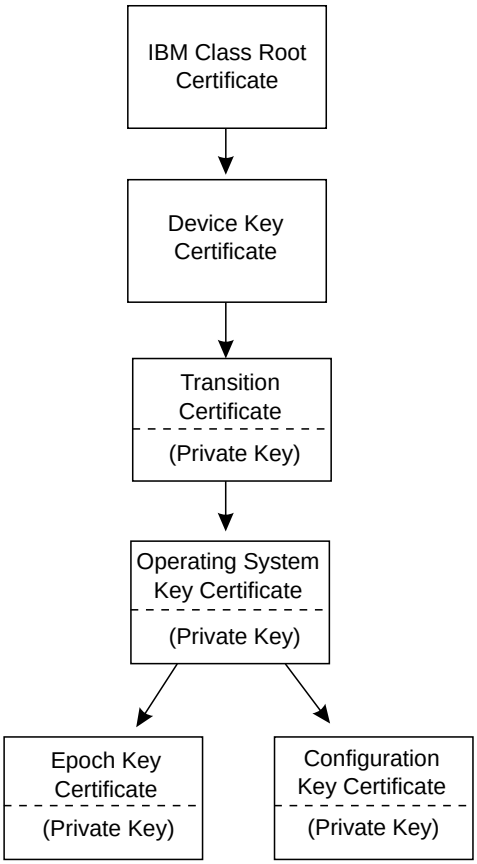

Figure 3-2. Application Generates Configuration Key

A new version of the operating system is loaded into the coprocessor in a manner that does not clear segment 3 BBRAM. This changes the configuration and so the private keys in the Operating System Keypair and the Configuration Keypair are deleted. This is appropriate since the configuration the Operating System Key Certificate names is no longer current and because Configuration Keypairs are by definition effective only during a single configuration. The private key in the Epoch Keypair is retained since the data in segment 3 BBRAM remains the same. The software in segment 1 creates a new Operating System Keypair and signs its certificate. The resulting certificate chain is shown in Figure 3-3.

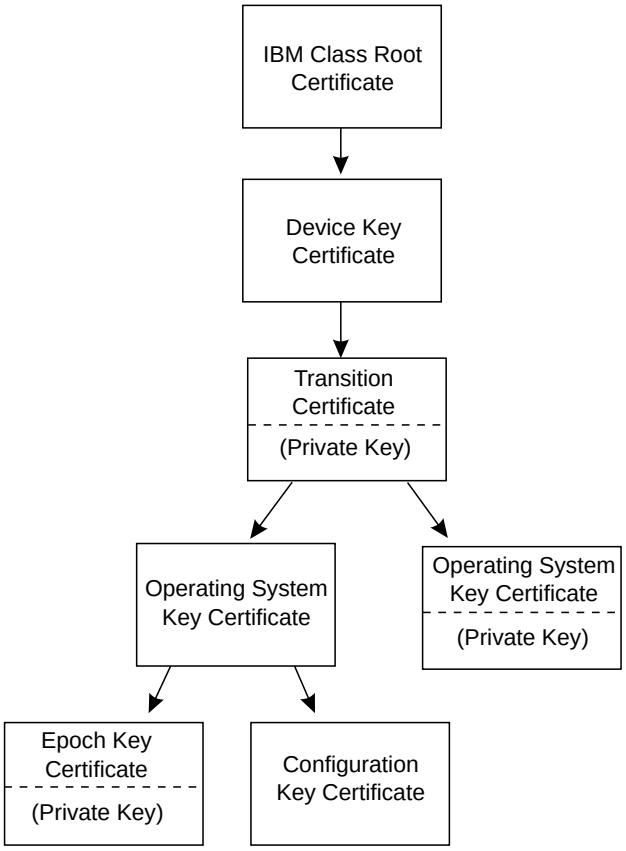

Figure 3-3. Operating System Updated

Since the existing Configuration Keypair no longer has a private key, the application asks the OA Manager to create a new Configuration Keypair. Figure 3-4 shows the new certificate chain. The application could generate another Epoch Keypair (whose certificate would be signed by the new Operating System Private Key), even though the epoch has not changed. One reason to do so (and to discontinue use of the original Epoch Private Key or delete the original Epoch Keypair entirely) is that it is easier to locate the current Operating System Key Certificate using the new Epoch Key Certificate rather than the old one, and the current Operating System Key Certificate is the one that identifies the new software in segment 2.

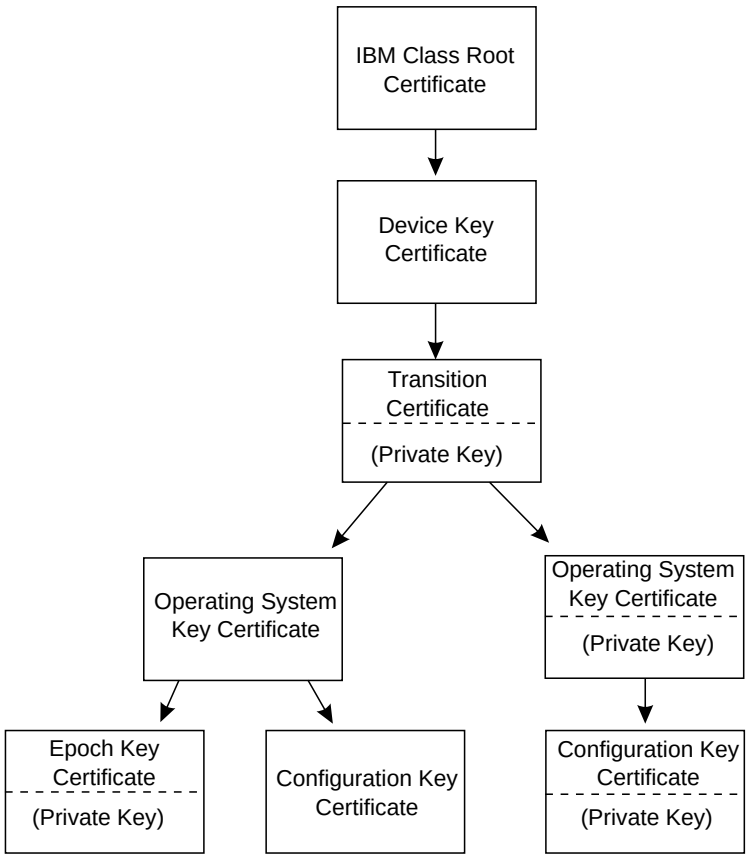

Figure 3-4. Application Generates New Configuration Key

The software in segment 1 is updated in a manner that does not clear segment 3 BBRAM. The existing Configuration Private Key and Operating System Private Key are deleted. A new Transition Certificate and Operating System Certificate are added to the certificate chain and new private keys are created, as shown in Figure 3-5.

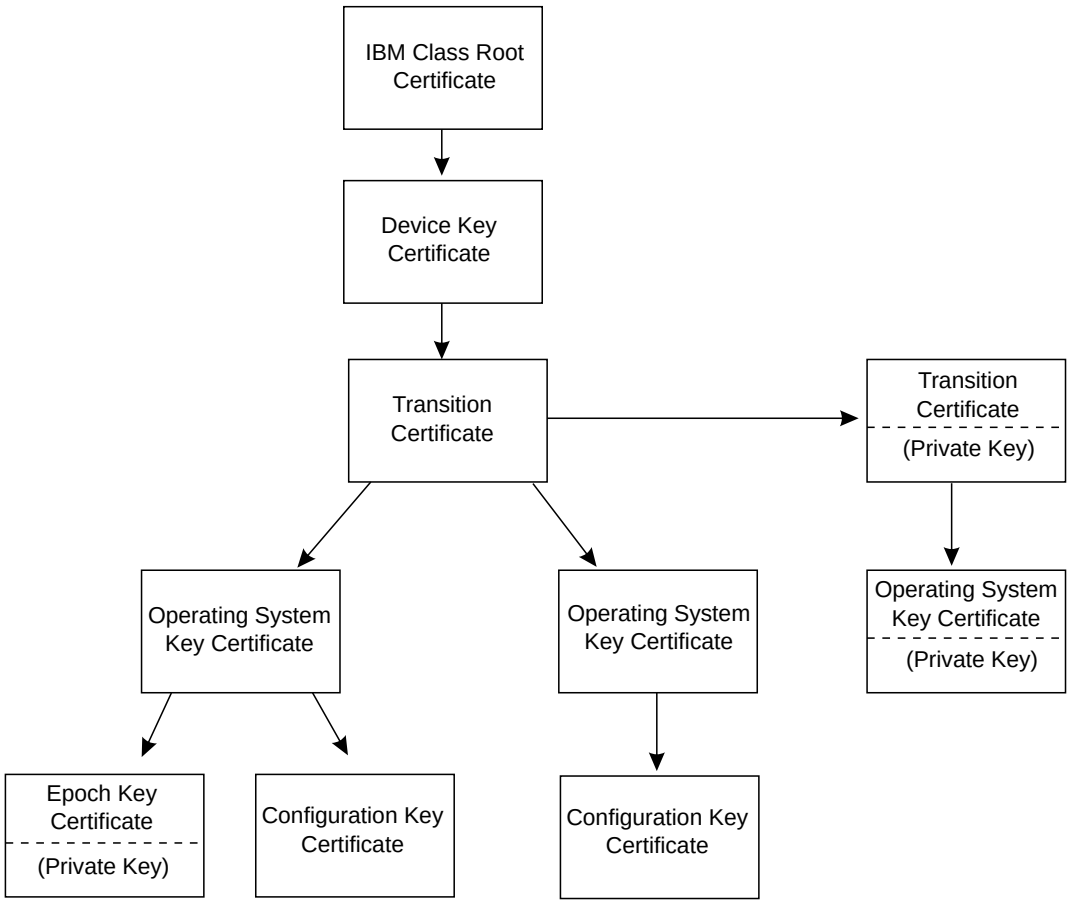

Figure 3-5. Miniboot Updated

The application asks the OA Manager to create a new Epoch Keypair and a new Configuration Keypair. The application then asks the OA Manager to delete the original Epoch Keypair and the certificate that contains the public half of the keypair. Figure 3-6 shows the certificate chain after these requests are processed.

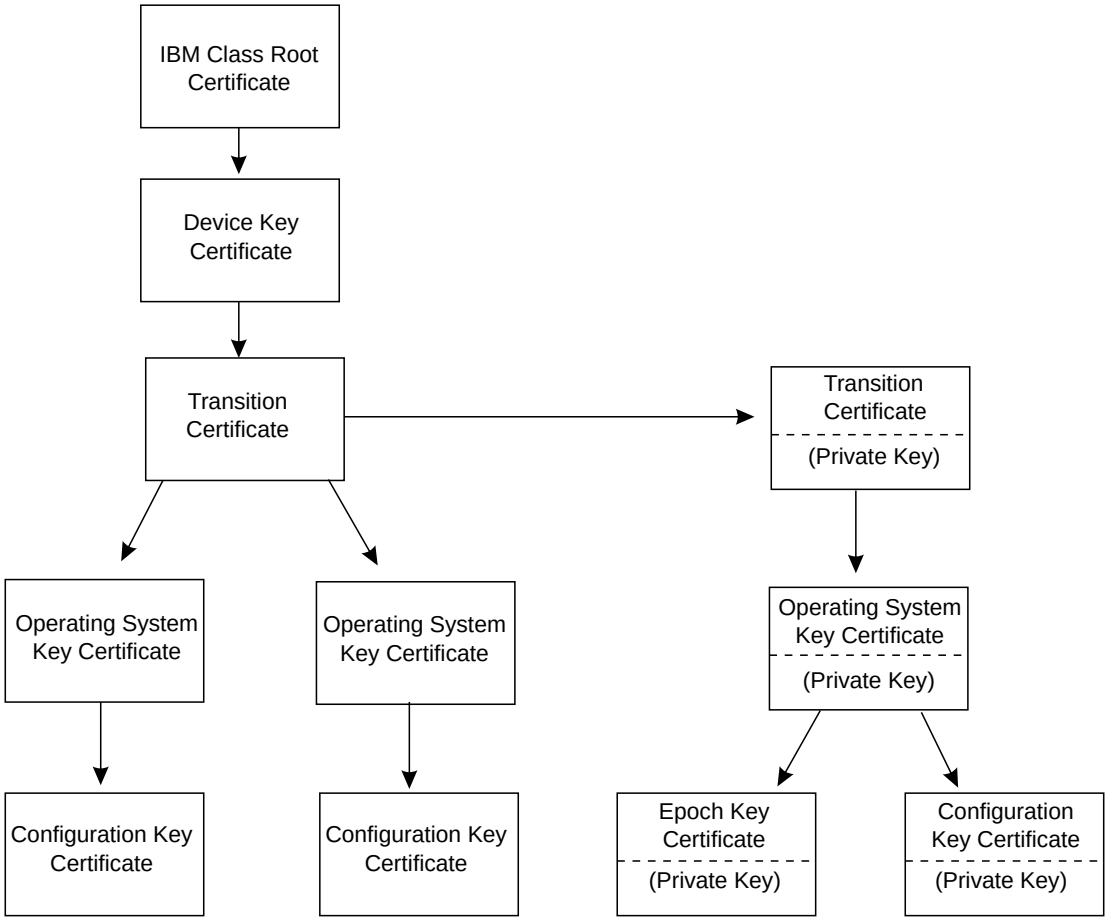

Figure 3-6. Configuration Keypair and Epoch Keypair Created

An application from another vendor is loaded into segment 3. This operation perforce clears segment 3 BBRAM. This ends the current epoch, so all existing Operating System Certificates and any certificates created on behalf of the old application are deleted, as are any private keys that correspond to the public keys in those certificates. The start of a new epoch also marks the beginning of a new configuration, so the software in segment 1 creates a new Operating System Keypair and the corresponding certificate. The resulting certificate chain is shown in Figure 3-7. Note that the current Operating System Certificate and private key are not the same as the current Operating System Certificate and private key in Figure 3-6 on page 3-104 even though the two certificates have the same parent. The items shown in Figure 3-6 on page 3-104 are deleted at the end of the epoch.

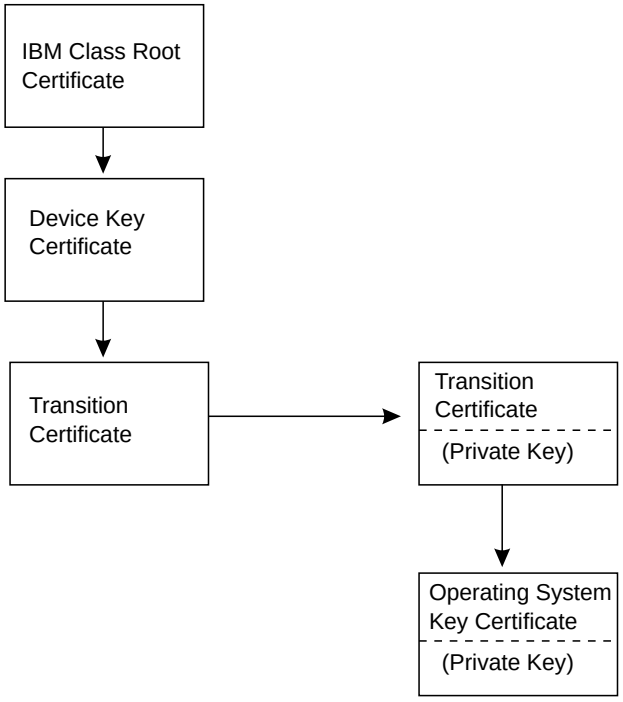

Figure 3-7. Foreign Application Loaded

OA Certificates

The interface to the Outbound Authentication (OA) Manager defines the `sccOA_CKO_Head_t` and `sccOA_CKO_Body_t` types to hold information about an OA certificate. An OA certificate has a variable length and consists of two descriptive headers followed by a buffer containing the various elements of the certificate. Figure 3-8 shows the general structure of an OA certificate.

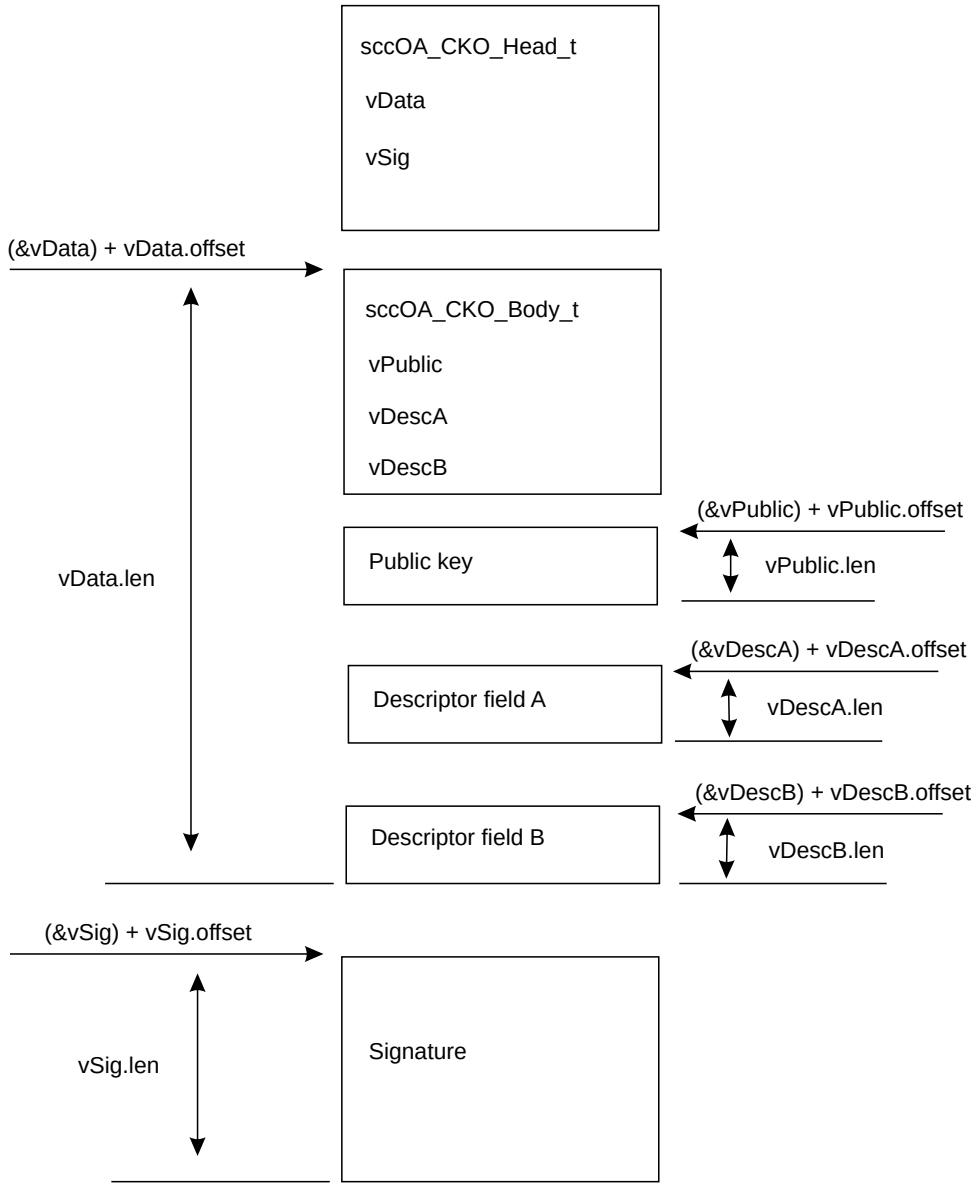

Figure 3-8. Structure of an OA Certificate

For convenience, the following fields appear both in the sccOA_CKO_Head_t header and in the sccOA_CKO_Body_t header. The fields in the first header are easier to locate, but only the fields in the second header are part of the body of the certificate and hence covered by the cryptographic signature for the certificate. The following discussion describes the fields only once, with the understanding that they should have the same values in both headers.

- cko_name
- cko_type
- parent_name

Fields Common to All Certificates

The first descriptive header is an item of type sccOA_CKO_Head_t. Certain fields in this header are either constant or interpreted in the same manner regardless of the type of certificate the header defines:

- struct_id.name is SCCOA_CKO_HEAD_T.
- struct_id.version is the value to which SCCOA_CKO_HEAD_VER is defined in the header file that defines the version of sccOA_CKO_Head_t that maps the header.[74]
- padbytes is two bytes of zeros.
- tData is OA_NEW_CERT.
- vData specifies the offset and length of the body of the certificate:
  - vData.offset is the offset in bytes from the start of the vData field to the first byte of the body of the certificate, which begins with the second descriptive header (an item of type sccOA_CKO_Body_t).[75]
  - vData.len is the length in bytes of the body of the certificate.[76]
- vSig specifies the offset and length of the cryptographic signature that covers the body of the certificate. The format of the signature depends on the value of the tsig field (see below).
  - vSig.offset is the offset in bytes from the start of the vSig field to the first byte of the signature.[75]
  - vSig.len is the length in bytes of the signature.[76]
- tSig specifies how the cryptographic signature that covers the body of the certificate is generated. The name of the keypair whose private key is used to generate the signature and whose public key is used to verify the signature appears in the parent_name field.
  - If tSig is SHA1_ISO_RSA, an RSA private key is used to generate the signature. The body of the certificate is hashed using the SHA-1 algorithm. The hash is extended according to the ISO 9796 standard to the full length of the modulus of the key. The extended hash is then decrypted using the RSA private key to give the signature.
  - If tSig is SHA1_ISO_COMPLEMENT_RSA, the cryptographic signature is computed in the same manner as in the SHA1_ISO_RSA case. If the

---

[74] For example, if the struct_id.version field in a structure of type SCCOATime_t is not equal to SCCOATIME_VER, the definition of SCCOATime_t used to build the code that performs the comparison does not match the definition used to build the code that created the structure, and the code that performs the comparison must not attempt to parse the structure unless it has another way to know how the structure is mapped.

[75] If *v* is an item of type var_t, the address of the item *v* describes is ((char *)&(v)) + v.offset. By convention, if v.offset is zero, the item *v* describes is empty or missing. Also by convention if *x* and *y* are var_t structures and *y* is a part of the item *x* describes, the item *y* describes is also a part of the item *x* describes (that is, "nested" var_t structures describe nested items).

[76] If *v* is an item of type var_t, the careful programmer will check that the region defined by v.offset and v.len is completely contained within the buffer or object that allegedly contains it.

signature is congruent to 6 modulo 16, it is then complemented with respect to the modulus of the key. Otherwise the signature is used as-is.
  - If `tSig` is DSS_COMPLIANT, a DSA private key is used to generate the signature. The body of the certificate is processed as dictated by the Digital Signature Standard.

  A signature generated using an RSA private key is stored as a simple (but potentially very large) binary integer. The block of data whose offset and length are specified in `vSig` contains the signature, which is stored in big-endian order: the byte at the lowest address is the most significant byte of the signature.

  A signature generated using a DSA private key is stored in a DSA signature token. The block of data whose offset and length are specified in `vSig` begins with a structure of type `sccDSASignatureToken_t`. This structure defines the elements of the signature (as described in "DSA Signature Tokens" on page 3-53), which appear following the `sccDSASignatureToken_t` structure.
- `cko_status` is OA_CKO_ACTIVE if the coprocessor knows the value of the private key corresponding to the public key contained in the certificate and is OA_CKO_INACTIVE otherwise.

The contents of the `cko_type`, `cko_name`, and `parent_name` fields should be copies of the corresponding fields in the `sccOA_CKO_Body_t` header.

The second descriptive header (which appears at the beginning of the body of the certificate) is an item of type `sccOA_CKO_Body_t`. Certain fields in this header are either constant or interpreted in the same manner regardless of the type of certificate the header defines:

- `struct_id.name` is SCCOA_CKO_BODY_T.
- `struct_id.version` is the value to which SCCOA_CKO_BODY_VER is defined in the header file that defines the version of `sccOA_CKO_Body_t` that maps the header.[77]
- `padbytes` is two bytes of zeros.
- `tPublic` specifies which type of public key the certificate contains:
  - If `tPublic` is OA_RSA, the public key is an RSA public key. The block of data whose offset and length are specified in `vPublic` begins with a structure of type `sccRSAKeyToken_t`. This structure defines the elements of the public key (as described in "RSA Key Tokens" on page 3-39), which appear following the `sccRSAKeyToken_t` structure.
  - If `tPublic` is OA_DSS, the public key is a DSA public key. The block of data whose offset and length are specified in `vPublic` begins with a structure of type `sccDSAKeyToken_t`. This structure defines the elements of the public key (as described in "DSA Key Tokens" on page 3-52), which appear following the `sccDSAKeyToken_t` structure.

[77] For example, if the `struct_id.version` field in a structure of type `SCCOATime_t` is not equal to `SCCOATIME_VER`, the definition of `SCCOATime_t` used to build the code that performs the comparison does not match the definition used to build the code that created the structure, and the code that performs the comparison must not attempt to parse the structure unless it has another way to know how the structure is mapped.

- vPublic specifies the offset and length of the public key the certificate contains:
  - vPublic.offset is the offset in bytes from the start of the vPublic field to the first byte of the public key.[78]
  - vPublic.len is the length in bytes of the public key.[79]
- cko_name identifies the keypair whose public key is contained in the certificate. This name is unique if the keypair is an IBM Root Keypair or an IBM Class Root Keypair. Keypairs of other types are generated by a coprocessor. Two keypairs generated by different coprocessors may have the same name but can be distinguished by using the device_name field. See "Keypair Names" on page 3-112 for details.
- parent_name identifies the keypair whose private key was used to create the cryptographic signature that covers the body of the certificate. This name is unique if the keypair is an IBM Root Keypair or an IBM Class Root Keypair. Keypairs of other types are generated by a coprocessor. Two keypairs generated by different coprocessors may have the same name but can be distinguished by using the device_name field. See "Keypair Names" on page 3-112 for details.

The contents of the remaining fields in the sccOA_CKO_Body_t header depend on which type of certificate the header defines.

IBM Class Root Certificates

The type-dependent fields in the sccOA_CKO_Body_t header for an IBM Class Root Certificate are set as follows:

- cko_type is OA_CKO_IBM_ROOT.
- cko_name names an IBM Class Root Keypair. See "Keypair Names" on page 3-112 for details.
- parent_name names an IBM Root Keypair. See "Keypair Names" on page 3-112 for details.
- device_name is undefined.
- vDescA specifies the offset and length of a timestamp that indicates when the IBM Class Root Keypair whose public key is contained in the certificate was created. See "Timestamps" on page 3-115 for details.
  - vDescA.offset is the offset in bytes from the start of the vDescA field to the first byte of the timestamp.[78]
  - vDescA.len is the length in bytes of the timestamp.[79]
- vDescB specifies the offset and length of a structure that describes the IBM Class Root Keypair whose public key is contained in the certificate. See "Class Root Descriptions" on page 3-116 for details.
  - vDescB.offset is the offset in bytes from the start of the vDescB field to the first byte of the description.[78]
  - vDescB.len is the length in bytes of the description.[79]

---

[78] If *v* is an item of type var_t, the address of the item *v* describes is ((char *)&(v)) + v.offset. By convention, if v.offset is zero, the item *v* describes is empty or missing. Also by convention if *x* and *y* are var_t structures and *y* is a part of the item *x* describes, the item *y* describes is also a part of the item *x* describes (that is, "nested" var_t structures describe nested items).

[79] If *v* is an item of type var_t, the careful programmer will check that the region defined by v.offset and v.len is completely contained within the buffer or object that allegedly contains it.

Device Key Certificates

The type-dependent fields in the sccOA_CKO_Body_t header for a Device Key Certificate are set as follows:

- cko_type is OA_CKO_MB.
- cko_name names a coprocessor-generated keypair. See “Keypair Names” on page 3-112 for details.
- parent_name names an IBM Class Root Keypair. See “Keypair Names” on page 3-112 for details.
- device_name uniquely identifies the coprocessor that generated the keypair whose public key is contained in the certificate. See “Device Names and Device Descriptors” on page 3-113 for details.
- vDescA specifies the offset and length of a description of the coprocessor that generated the keypair whose public key is contained in the certificate that was created. See “Device Names and Device Descriptors” on page 3-113 for details.
  - vDescA.offset is the offset in bytes from the start of the vDescA field to the first byte of the device description.[80]
  - vDescA.len is the length in bytes of the device description.[81]
- vDescB specifies the offset and length of a layer descriptor that describes the miniboot software (that is, the software in segment 1) that was present in the coprocessor identified by device_name when that coprocessor created the keypair whose public key is contained in the certificate. See “Layer Names and Layer Descriptors” on page 3-114 for details.
  - vDescB.offset is the offset in bytes from the start of the vDescB field to the first byte of the layer descriptor.[80]
  - vDescB.len is the length in bytes of the layer descriptor.[81]

Transition Certificates

The type-dependent fields in the sccOA_CKO_Body_t header for a Transition Certificate are set as follows:

- cko_type is OA_CKO_MB.
- cko_name names a coprocessor-generated keypair. See “Keypair Names” on page 3-112 for details.
- parent_name names a keypair whose public key is contained in a Device Key Certificate or in a Transition Certificate. See “Keypair Names” on page 3-112 for details.
- device_name uniquely identifies the coprocessor that generated the keypair whose public key is contained in the certificate. See “Device Names and Device Descriptors” on page 3-113 for details.
- vDescA specifies the offset and length of a description of the coprocessor that generated the keypair whose public key is contained in the certificate. See “Device Names and Device Descriptors” on page 3-113 for details.
  - vDescA.offset is the offset in bytes from the start of the vDescA field to the first byte of the device description.[80]
  - vDescA.len is the length in bytes of the device description.[81]

---

[80] If *v* is an item of type var_t, the address of the item *v* describes is ((char *)&(v)) + v.offset. By convention, if v.offset is zero, the item *v* describes is empty or missing. Also by convention if *x* and *y* are var_t structures and *y* is a part of the item *x* describes, the item *y* describes is also a part of the item *x* describes (that is, “nested” var_t structures describe nested items).

[81] If *v* is an item of type var_t, the careful programmer will check that the region defined by v.offset and v.len is completely contained within the buffer or object that allegedly contains it.

- vDescB specifies the offset and length of a layer descriptor that describes the miniboot software (that is, the software in segment 1) that was present in the coprocessor identified by device_name when that coprocessor created the keypair whose public key is contained in the certificate. See “Layer Names and Layer Descriptors” on page 3-114 for details.
  - vDescB.offset is the offset in bytes from the start of the vDescB field to the first byte of the layer descriptor.[82]
  - vDescB.len is the length in bytes of the layer descriptor.[83]

Operating System Key Certificates

The type-dependent fields in the sccOA_CKO_Body_t header for an Operating System Key Certificate are set as follows:

- cko_type is OA_CKO_SEG2_SEG3.
- cko_name names a coprocessor-generated keypair. See “Keypair Names” on page 3-112 for details.
- parent_name names a keypair whose public key is contained in a Device Key Certificate or in a Transition Certificate. See “Keypair Names” on page 3-112 for details.
- device_name uniquely identifies the coprocessor that generated the keypair whose public key is contained in the certificate. See “Device Names and Device Descriptors” on page 3-113 for details.
- vDescA specifies the offset and length of a layer descriptor that describes the operating system (that is, the software in segment 2) that was present in the coprocessor identified by device_name when that coprocessor created the keypair whose public key is contained in the certificate. See “Layer Names and Layer Descriptors” on page 3-114 for details.
  - vDescA.offset is the offset in bytes from the start of the vDescA field to the first byte of the layer descriptor.[82]
  - vDescA.len is the length in bytes of the layer descriptor.[83]
- vDescB specifies the offset and length of a layer descriptor that describes the application (that is, the software in segment 3) that was present in the coprocessor identified by device_name when that coprocessor created the keypair whose public key is contained in the certificate. See “Layer Names and Layer Descriptors” on page 3-114 for details.
  - vDescB.offset is the offset in bytes from the start of the vDescB field to the first byte of the layer descriptor.[82]
  - vDescB.len is the length in bytes of the layer descriptor.[83]

Application Key Certificates

The type-dependent fields in the sccOA_CKO_Body_t header for an Application Key Certificate are set as follows:

- cko_type is OA_CKO_SEG3_CONFIG if the public key the certificate contains is part of a Configuration Keypair and OA_CKO_SEG3_EPOCH if the public key is part of an Epoch Keypair.

---

[82] If *v* is an item of type var_t, the address of the item *v* describes is ((char *)&(v)) + v.offset. By convention, if v.offset is zero, the item *v* describes is empty or missing. Also by convention if *x* and *y* are var_t structures and *y* is a part of the item *x* describes, the item *y* describes is also a part of the item *x* describes (that is, “nested” var_t structures describe nested items).

[83] If *v* is an item of type var_t, the careful programmer will check that the region defined by v.offset and v.len is completely contained within the buffer or object that allegedly contains it.

- `cko_name` names a coprocessor-generated keypair. See “Keypair Names” on page 3-112 for details.
- `parent_name` names a keypair whose public key is contained in an Operating System Key Certificate. See “Keypair Names” for details.
- `device_name` uniquely identifies the coprocessor that generated the keypair whose public key is contained in the certificate. See “Device Names and Device Descriptors” on page 3-113 for details.
- `vDescA` is reserved.
- `vDescB` specifies the offset and length of a block of data supplied by the application to be associated with the certificate when the keypair was created. See “sccOAGenerate - Generate Application Keypair and OA Certificate” on page 3-121 for details.
  - `vDescB.offset` is the offset in bytes from the start of the `vDescB` field to the first byte of the block of data supplied by the application.[84]
  - `vDescB.len` is the length in bytes of the block of data supplied by the application.[85]

Keypair Names

The interface to the OA Manager defines the `sccOA_CKO_Name_t` type to hold the name of a keypair. The contents of the fields in a `sccOA_CKO_Name_t` structure depend on which type of keypair the structure names.

IBM Root Keypairs

The fields in a `sccOA_CKO_Name_t` structure that names an IBM Root Keypair are set as follows:

- `name_type` is OA_IBM_ROOT.
- `index` is an integer that distinguishes the IBM Root Keypair named by the structure from all other IBM Root Keypairs.
- `creation_boot` is not used.

IBM Class Root Keypairs

The fields in a `sccOA_CKO_Name_t` structure that names an IBM Class Root Keypair are set as follows:

- `name_type` is OA_IBM_CLASS_ROOT.
- `index` is an integer that distinguishes the IBM Class Root Keypair named by the structure from all other IBM Class Root Keypairs.
- `creation_boot` is not used.

---

[84] If *v* is an item of type `var_t`, the address of the item *v* describes is `((char *)&(v)) + v.offset`. By convention, if `v.offset` is zero, the item *v* describes is empty or missing. Also by convention if *x* and *y* are `var_t` structures and *y* is a part of the item *x* describes, the item *y* describes is also a part of the item *x* describes (that is, “nested” `var_t` structures describe nested items).

[85] If *v* is an item of type `var_t`, the careful programmer will check that the region defined by `v.offset` and `v.len` is completely contained within the buffer or object that allegedly contains it.

Coprocessor-Generated Keypairs

The fields in a sccOA_CKO_Name_t structure that names a keypair that was generated on a coprocessor (that is, any keypair except an IBM Root Keypair or an IBM Class Root Keypair) are set as follows:

- name_type is OA_STANDARD_NAME.
- index is an integer that distinguishes the keypair named by the structure from all other keypairs generated by the same coprocessor that have the same value for creation_boot.
- creation_boot is the value the boot counter on the coprocessor that generated the keypair that the structure names had when the keypair was generated. See “Coprocessor Architecture” on page 3-95 for details.

Note that the names of two keypairs generated on a single coprocessor are distinct, but that the name a keypair generated on one coprocessor may match the name of a keypair generated on another coprocessor. In general, the device_name field in an OA certificate must be used to distinguish keys generated on one coprocessor from keys generated on another coprocessor.

Device Names and Device Descriptors

The interface to the OA Manager defines the sccOADeviceName_t type to hold the name of a particular coprocessor. The fields in a sccOADeviceName_t structure are set as follows:

- struct_id.name is SCCOADEVICENAME_T.
- struct_id.version is the value to which SCCOADEVICENAME_VER is defined in the header file that defines the version of sccOADeviceName_t that maps the header.[86]
- padbytes is two bytes of zeros.
- adapterID is a serial number that uniquely identifies the coprocessor. It matches the value of the AdapterID field returned by sccGetConfig. See “sccGetConfig - Get Coprocessor Configuration” on page 3-89 for details.
- when_certified is a timestamp that indicates when the Device Key Certificate was loaded into the coprocessor during manufacture. See “Timestamps” on page 3-115 for details.

The interface to the OA Manager defines the sccOADeviceDesc_t type to hold a description of a particular coprocessor. The fields in a sccOADeviceDesc_t structure are set as follows:

- struct_id.name is SCCOADEVICEDESC_T.
- struct_id.version is the value to which SCCOADEVICEDESC_VER is defined in the header file that defines the version of sccOADeviceDesc_t that maps the header.[86]
- padbytes is two bytes of zeros.

[86] For example, if the struct_id.version field in a structure of type SCCOATime_t is not equal to SCCOATIME_VER, the definition of SCCOATime_t used to build the code that performs the comparison does not match the definition used to build the code that created the structure, and the code that performs the comparison must not attempt to parse the structure unless it has another way to know how the structure is mapped.

- vpd is a description of the coprocessor. The first 128 bytes match the value of the AMCC_EEPROM field returned by sccGetConfig and the next 128 bytes match the value of the VPD field returned by sccGetConfig. See "sccGetConfig - Get Coprocessor Configuration" on page 3-89 for details.
- device_name is the coprocessor's name.

Layer Names and Layer Descriptors

The interface to the OA Manager defines the sccOALayerName_t type to hold an identifier that uniquely identifies the software loaded into a particular segment of a particular coprocessor. The fields of a layer name are set as follows:

- struct_id.name is SCCOALAYERNAME_T.
- struct_id.version is the value to which SCCOALAYERNAME_VER is defined in the header file that defines the version of sccOALayerName_t that maps the timestamp.[87]
- padbytes is two bytes of zeros.
- epoch_start marks the beginning of the epoch in which the software that the structure names was loaded. In particular, epoch_start is
  - the value of the boot counter
  - on the coprocessor into which the software that the structure names was loaded
  - at the point the epoch in which the software that the structure names was loaded began.

  See "Overview of the Authentication Scheme" on page 3-96 for details.[88]
- config_start marks the start of the configuration that includes the software that the structure names. In particular, config_start is
  - the value the boot counter
  - on the coprocessor into which the software the structure names was loaded
  - at the point the software that the structure names was loaded.

  See "Overview of the Authentication Scheme" on page 3-96 for details.[88]

---

[87] For example, if the struct_id.version field in a structure of type SCCOATime_t is not equal to SCCOATIME_VER, the definition of SCCOATime_t used to build the code that performs the comparison does not match the definition used to build the code that created the structure, and the code that performs the comparison must not attempt to parse the structure unless it has another way to know how the structure is mapped.

[88] As mentioned in footnote [73] on page 3-97, epochs and configurations are measured with respect to a particular segment. Thus, the values of the recorded boot counter values in a layer 2 descriptor in an Operating System Certificate may differ from the corresponding values in the layer 3 descriptor in the same certificate. Consider the following sequence of operations:

1. The operating system is loaded into an empty coprocessor when the boot counter is 0x60c. This begins a new segment 2 epoch and a new segment 2 configuration. The segment 2 configuration count is initialized to 1.
2. An application is loaded into segment 3 for the first time when the boot counter is 0x60d. This begins a new segment 3 epoch and a new segment 3 configuration. The segment 3 configuration count is initialized to 1.
3. The debug kernel is loaded into the coprocessor when the boot counter is 0x612. This begins a new segment 2 configuration *and* a new segment 3 configuration. Both configuration counts are incremented.
4. A new application is loaded into the coprocessor when the boot counter is 0x620. This begins a new segment 3 configuration and the segment 3 configuration count is incremented.

The Operating System Certificate created during step 4 will have a layer descriptor for segment 2 whose fields have the following values:

- epoch_start = 0x60c
- config_start = 0x612
- config_count = 2

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

- epoch_start = 0x60d
- config_start = 0x620
- config_count = 3

- config_count specifies how many configurations there have been during the epoch whose beginning epoch_start defines. This figure includes the configuration that began when the software the structure names was loaded. See “Overview of the Authentication Scheme” on page 3-96 for details.[88]

The interface to the OA Manager defines the sccOALayerDesc_t type to hold a description of the software loaded into a particular segment of a particular coprocessor. The fields of a layer description are set as follows:

- struct_id.name is SCCOALAYERDESC_T.
- struct_id.version is the value to which SCCOALAYERDESC_VER is defined in the header file that defines the version of sccOALayerDesc_t that maps the layer description.[89]
- padbyte is one byte of zeros.
- layer_number is 1 if the software the structure describes is loaded into segment 1, 2 if the software is loaded into segment 2, and 3 if the software is loaded into segment 3.
- ownerID is the owner identifier associated with the segment into which the software is loaded. See “Overview of the Authentication Scheme” on page 3-96 for details.
- image_name is the name associated with the software. See “Overview of the Authentication Scheme” on page 3-96 for details.
- image_revision is the revision number associated with the software. See “Overview of the Authentication Scheme” on page 3-96 for details.
- image_hash is the SHA-1 hash of the software. See “Overview of the Authentication Scheme” on page 3-96 for details.
- layer_name uniquely identifies the software.

Timestamps

The interface to the OA Manager defines the sccOATime_t type to hold a timestamp. The fields of a timestamp are set as follows:

- struct_id.name is SCCOATIME_T.
- struct_id.version is the value to which SCCOATIME_VER is defined in the header file that defines the version of sccOATime_t that maps the timestamp.[89]
- year is a BCD representation of the year (for example, 0x2000 represents the year 2000).[90]
- month is a BCD representation of the month (for example, 0x12 represents December).
- day is a BCD representation of the day of the month (for example, 0x10 represents the 10th).
- hour is a BCD representation of the hour using a 24-hour clock (for example, 0x17 represents 5 p.m.).
- minute is a BCD representation of the minute (for example, 0x25 represents 25 minutes past the hour).

---

89 For example, if the struct_id.version field in a structure of type SCCOATime_t is not equal to SCCOATIME_VER, the definition of SCCOATime_t used to build the code that performs the comparison does not match the definition used to build the code that created the structure, and the code that performs the comparison must not attempt to parse the structure unless it has another way to know how the structure is mapped.

90 year is a normal arithmetic item and is stored in little-endian order. For example, if *t* is a timestamp for a date in the year 2000, ((char *)&(t.year))[0] is 0 and ((char t.year))[1] is 0x20.

Timestamps created on a coprocessor are set to the date and time provided by the coprocessor's real-time clock, which should be synchronized with an external clock if an accurate timestamp is required.

Class Root Descriptions

The interface to the OA Manager defines the `sccOA_CKO_Descr_t` type to hold the description of an IBM Class Root Keypair. The fields in the `sccOA_CKO_Descr_t` structure are set as follows:

- `struct_id.name` is SCCOA_CKO_DESCR_T.
- `struct_id.version` is the value to which SCCOA_CKO_DESCR_VER is defined in the header file that defines the version of `sccOA_CKO_Descr_t` that maps the description.[91]
- `cert_qualifier` is an integer that identifies the keypair. Recognized values are listed in *scc_oa.h*.
- `descr` is a text description of the keypair.

[91] For example, if the `struct_id.version` field in a structure of type `SCCOATime_t` is not equal to `SCCOATIME_VER`, the definition of `SCCOATime_t` used to build the code that performs the comparison does not match the definition used to build the code that created the structure, and the code that performs the comparison must not attempt to parse the structure unless it has another way to know how the structure is mapped.

sccOAGetDir - Count and List OA Certificates

sccOAGetDir determines the total number of OA Certificates the OA Manager has saved. Information about the certificates can also be retrieved.

Function Prototype

```
long sccOAGetDirAsync(unsigned long *pCount,
                      void          *pBuffer,
                      unsigned long *pLen,
                      unsigned long *pMsgID);

#define sccOAGetDir(pc,pb,pl) sccOAGetDirAsync(pc,pb,pl,NULL)
```

Input

On entry to this routine:

pCount must contain the address of a variable in which an item of type unsigned long can be stored.

pBuffer must contain the address of a buffer in which information about all of the OA Certificates the OA Manager has saved can be returned if this information is desired and must be NULL otherwise.

pLen must contain the address of a variable in which an item of type unsigned long can be stored. If pBuffer is not NULL, *pLen must be the length in bytes of the buffer referenced by pBuffer.

pMsgID determines whether the function is performed synchronously or asynchronously:

- If pMsgID is NULL, the function is performed synchronously. The call does not return until the requested information has been retrieved.
- If pMsgID is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the OA Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the areas of memory referenced by pCount, pBuffer, or pLen before the operation is complete.

Output

On successful exit from this routine:

If pMsgID is NULL, *pCount is the number of OA Certificates the OA Manager has saved. If pBuffer is not NULL, the buffer it references contains an array of items of type sccOA_DirItem_t and *pLen is the length in bytes of this array. The fields of the *i* th entry in the array are set as follows:

- struct_id.name is SCCOA_DIRITEM_T.
- struct_id.version is the value to which SCCOA_DIRITEM_VER is defined in the header file that defines the version of sccOA_DirItem_t that maps the entry.[92]

[92] For example, if the struct_id.version field in a structure of type SCCOATime_t is not equal to SCCOATIME_VER, the definition of SCCOATime_t used to build the code that performs the comparison does not match the definition used to build the code that created the structure, and the code that performs the comparison must not attempt to parse the structure unless it has another way to know how the structure is mapped.

- `padbytes` is two bytes of zeros.
- `cko_name` identifies the keypair whose public key is contained in the OA Certificate the *i* th entry describes. See "Keypair Names" on page 3-112 for details.[93]
- `algorithm` is OA_RSA if the keypair identified by `cko_name` is an RSA keypair and is OA_DSA if the keypair is a DSA keypair.
- `cko_status` is OA_CKO_ACTIVE if the private key in the keypair identified by `cko_name` exists and is OA_CKO_INACTIVE if the private key does not exist (for example, because the software configuration has changed since the keypair was created).
- `length` is the length in bytes of the OA Certificate the *i* th entry describes (that is, the minimum size of a buffer that could hold the OA Certificate).
- `parent_index` is the index within the array referenced by `pBuffer` of the entry that describes the OA Certificate that contains the public key corresponding to the private key that was used to create the cryptographic signature that covers the body of the OA Certificate the *i* th entry describes. If `parent_index` is negative, there is no such entry in the array referenced by `pBuffer` (for example, because the certificate is for an IBM Class Root key).

If `pBuffer` is NULL, `*pLen` is the length in bytes of a buffer that is just large enough to hold an array of items of type `sccOA_DirItem_t` that contains an entry for each OA Certificiate the OA Manager has saved.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the OA Manager to initiate the desired operation. When the operation is complete, the OA Manager will send the coprocessor application a message whose type field (`Msg.h.msg_type`) contains this identifier and whose first (and only) data item (`Msg.msg_data[0]`) contains the return code generated by the routine.[94] If the operation was successful, `*pCount` and `*pLen` (and `*pBuffer`, if appropriate) contain the result. The message is placed on the default CP/Q message queue for the task that called sccOAGetDirAsync.

Return Codes

Common return codes generated by this routine are:

OAGood (i.e., 0) The operation was successful.

OABadParm An argument is not valid.

OANoSpace The operation failed due to lack of space. (for example, `*pLen` is too small).

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

---

[93] Since the keypair in question was perforce generated on the coprocessor on which the application that calls sccOAGetDir is running, `cko_name` is unambiguous regardless of what kind of keypair it names. There is no need for a Device Name.

[94] The return code from the call to sccOAGetDirAsync indicates whether or not the initial message to the OA Manager was successfully enqueued.

sccOAGetCert - Retrieve an OA Certificate

sccOAGetCert either returns the length of an OA Certificate the OA Manager has saved or retrieves the certificate itself.

Function Prototype

```
long sccOAGetCertAsync(sccOA_CKO_Name_t *pName,
                       void             *pBuffer,
                       unsigned long    *pLen,
                       unsigned long    *pMsgID);

#define sccOAGetCert(pn,pb,pl) sccOAGetCertAsync(pn,pb,pl,NULL)
```

Input

On entry to this routine:

*pName is the name of the keypair whose public key is contained in the OA Certificate that is to be retrieved or whose length is to be returned. See “Keypair Names” on page 3-112 for details.

pBuffer must contain the address of a buffer in which the OA Certificate identified by *pName can be returned if the OA Certificate is to be retrieved and must be NULL otherwise.

pLen must contain the address of a variable in which an item of type unsigned long can be stored. If pBuffer is not NULL, *pLen must be the length in bytes of the buffer referenced by pBuffer.

pMsgID determines whether the function is performed synchronously or asynchronously:

- If pMsgID is NULL, the function is performed synchronously. The call does not return until the requested information has been retrieved.
- If pMsgID is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the OA Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the areas of memory referenced by pName, pBuffer, or pLen before the operation is complete.

Output

On successful exit from this routine:

If pMsgID is NULL, *pLen is the length in bytes of the OA Certificate. If pBuffer is not NULL, the buffer pBuffer references contains a copy of the desired OA Certificate. See “OA Certificates” on page 3-106 for details.

If pMsgID is not NULL, *pMsgID uniquely identifies the message that was sent to the OA Manager to initiate the desired operation. When the operation is complete, the OA Manager will send the coprocessor application a message whose type field (Msg.h.msg_type) contains this identifier and whose first (and only) data item (Msg.msg_data[0]) contains the return code generated by the routine.[95] If the

[95] The return code from the call to sccOAGetCertAsync indicates whether or not the initial message to the OA Manager was successfully enqueued.

operation was successful, `*pLen` (and `*pBuffer`, if appropriate) contain the result. The message is placed on the default CP/Q message queue for the task that called sccOAGetCertAsync.

Return Codes

Common return codes generated by this routine are:

OAGood (i.e., 0)	The operation was successful.
OABadParm	An argument is not valid.
OANoSpace	The operation failed due to lack of space.
OANotFound	`*pName` does not identify an OA Certificate that the OA Manager has saved.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccOAGenerate - Generate Application Keypair and OA Certificate

sccOAGenerate generates a new Application Keypair and an OA Certificate that contains the keypair's public key.

Function Prototype

```
long sccOAGenerateAsync(sccOAGen_RB_t *pOAGRB,
                        unsigned long  OAGRBlen,
                        void          *pKGRB,
                        unsigned long  KGRBlen,
                        unsigned long *pMsgID);

#define sccOAGenerate(po,ol,pk,kl) sccOAGenerateAsync(po,ol,pk,kl,NULL)
```

Input

On entry to this routine:

pOAGRB must contain the address of an OA Generate request block whose fields are initialized as follows:

- struct_id.name is SCCOAGEN_RB_T.
- struct_id.version is the value to which SCCOAGEN_RB_VER is defined in the header file that defines the version of sccOAGen_RB_t that maps the request block.[96]
- padbytes is two bytes of zeros.
- algorithm specifies the cryptosystem used to generate the keypair and must be either OA_RSA (to generate an RSA keypair) or OA_DSA (to generate a DSA keypair).
- cko_type specifies what kind of Application Keypair is generated and must be either OA_CKO_SEG3_CONFIG (to generate a configuration key) or OA_CKO_SEG3_EPOCH (to generate an epoch key).
- vSeg3Field specifies the offset and length of a block of data to be stored in the new OA Certificate. The block is copied to the body of the certificate and the certificate's vDescB field describes the block's location and length.
  - vSeg3Field.offset is the offset in bytes from the start of the vSeg3Field field to the first byte of the block of data.[97]
  - vSeg3Field.len is the length in bytes of the block of data.[98]
- pCKO_name must contain the address of a buffer in which an item of type sccOA_CKO_Name_t can be stored.

OAGRBlen is the length in bytes of the request block referenced by pOAGRB. OAGRBlen includes the length of the block of data whose offset and length are specified in OAGRB->vSeg3Field.

---

[96] For example, if the struct_id.version field in a structure of type SCCOATime_t is not equal to SCCOATIME_VER, the definition of SCCOATime_t used to build the code that performs the comparison does not match the definition used to build the code that created the structure, and the code that performs the comparison must not attempt to parse the structure unless it has another way to know how the structure is mapped.

[97] If *v* is an item of type var_t, the address of the item *v* describes is ((char *)&(v)) + v.offset. By convention, if v.offset is zero, the item *v* describes is empty or missing. Also by convention if *x* and *y* are var_t structures and *y* is a part of the item *x* describes, the item *y* describes is also a part of the item *x* describes (that is, "nested" var_t structures describe nested items).

[98] If *v* is an item of type var_t, the careful programmer will check that the region defined by v.offset and v.len is completely contained within the buffer or object that allegedly contains it.

pKGRB must contain the address of a public key algorithm key generate request block:

- If pOAGRB->algorithm is OA_RSA, *pKGRB must be an RSA key generate request block (an item of type sccRSAKeyGen_RB_t) whose key_type, mod_size, and public_exp fields are initialized as required by sccRSAKeyGenerate (see "sccRSAKeyGenerate - Generate RSA Key Pair" on page 3-43 for details). public_exp must not be RSA_EXPONENT_FIXED. The remaining fields in *pKGRB are ignored.
- If pOAGRB->algorithm is OA_DSS, *pKGRB must be a DSA key generate request block (an item of type sccDSAKeyGen_RB_t) whose prime_p_size field is initialized as required by sccDSAKeyGenerate (see "sccDSAKeyGenerate - Generate DSA Key Pair" on page 3-54 for details). The remaining fields in *pKGRB are ignored.

KGRBlen is the length in bytes of the request block referenced by pKGRB (that is, sizeof(sccRSAKeyGen_RB_t) if pOAGRB->algorithm is OA_RSA and sizeof(sccDSAKeyGen_RB_t) if pOAGRB->algorithm is OA_DSS).

pMsgID determines whether the function is performed synchronously or asynchronously:

- If pMsgID is NULL, the function is performed synchronously. The call does not return until the requested information has been retrieved.
- If pMsgID is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the OA Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the areas of memory referenced by pOAGRB, pOAGRB->pCKO_name, or pKGRB before the operation is complete.

Output

On successful exit from this routine:

If pMsgID is NULL, *pCKO_name identifies the newly generated Application Keypair. See "Keypair Names" on page 3-112 for details.

If pMsgID is not NULL, *pMsgID uniquely identifies the message that was sent to the OA Manager to initiate the desired operation. When the operation is complete, the OA Manager will send the coprocessor application a message whose type field (Msg.h.msg_type) contains this identifier and whose first (and only) data item (Msg.msg_data[0]) contains the return code generated by the routine.[99] If the operation was successful, *pCKO_name contains the result. The message is placed on the default CP/Q message queue for the task that called sccOAGenerateAsync.

Notes

Signature on New OA Certificate

The cryptographic signature for the OA Certificate generated by sccOAGenerate is created using the private key from the current Operating System Keypair (that is, the private key corresponding to the public key contained in the unique OA

[99] The return code from the call to sccOAGenerateAsync indicates whether or not the initial message to the OA Manager was successfully enqueued.

Certificate whose `cko_type` field is OA_CKO_SEG2_SEG3 and whose `cko_status` field is CKO_ACTIVE).

Use Separate Keys to Encrypt and to Sign

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

Return Codes

Common return codes generated by this routine are:

OAGood (i.e., 0)	The operation was successful.
OABadParm	An argument is not valid.
OANotAllowed	`pOAGRB->cko_type` is neither OA_CKO_SEG3_CONFIG nor OA_CKO_SEG3_EPOCH.
OANoSpace	The operation failed due to lack of space.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccOADelete - Delete Application Keypair and OA Certificate

sccOADelete deletes an Application Keypair and the OA Certificate that contains the keypair's public key.

Function Prototype

```
long sccOADeleteAsync(sccOA_CKO_Name_t *pName,
                      unsigned long   *pMsgID);

#define sccOADelete(p) sccOADeleteAsync(p,NULL)
```

Input

On entry to this routine:

*pName is the name of the keypair to delete. *pName must identify an Application Keypair. See "Keypair Names" on page 3-112 for details.

pMsgID determines whether the function is performed synchronously or asynchronously:

- If pMsgID is NULL, the function is performed synchronously. The call does not return until the requested information has been retrieved.
- If pMsgID is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the OA Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the area of memory referenced by pName before the operation is complete.

Output

On successful exit from this routine:

If pMsgID is NULL, the keypair identified by *pName and the OA Certificate that contains the keypair's public key have been deleted.

If pMsgID is not NULL, *pMsgID uniquely identifies the message that was sent to the OA Manager to initiate the desired operation. When the operation is complete, the OA Manager will send the coprocessor application a message whose type field (Msg.h.msg_type) contains this identifier and whose first (and only) data item (Msg.msg_data[0]) contains the return code generated by the routine.[100] If the operation was successful, the keypair and OA Certificate that contains the keypair's public key have been deleted. The message is placed on the default CP/Q message queue for the task that called sccOADeleteAsync.

[100] The return code from the call to sccOADeletetAsync indicates whether or not the initial message to the OA Manager was successfully enqueued.

Return Codes

Common return codes generated by this routine are:

OAGood (i.e., 0)	The operation was successful.
OABadParm	An argument is not valid.
OANotAllowed	`*pName` does not identify an Application Keypair.
OANotFound	`*pName` does not identify an OA Certificate that the OA Manager has saved.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccOAPrivOp - Perform Cryptographic Operation with an Application Key

sccOAPrivOp directs the OA Manager to perform a cryptographic operation with an Application Key. The private key can be used to decrypt or sign a block of data, and the public key can be used to encrypt a block of data or verify a cryptographic signature.

Function Prototype

```
long sccOAPrivOpAsync(sccOA_CKO_Name_t *pName,
                      void             *pPKARB,
                      unsigned long     PKARBlen,
                      unsigned long    *pMsgID);

#define sccOAPrivOp(pn,pp,pl) sccOAPrivOpAsync(pn,pp,pl,NULL)
```

Input

On entry to this routine:

*pName is the name of the keypair to be used in the cryptographic operation. *pName must identify an Application Keypair. See “Keypair Names” on page 3-112 for details.

pPKARB must contain the address of a public key algorithm operation request block:

- If the keypair identified by *pName is an RSA keypair, *pPKARB must be an RSA operation request block (an item of type sccRSA_RB_t) whose options, data_in, data_out, and data_size fields are initialized as required by sccRSA (see “sccRSA - Encipher/Decipher Data or Wrap/Unwrap X9.31 Encapsulated Hash” on page 3-46 for details). options must include RSA_DONT_BLIND. The remaining fields in *pPKARB are ignored.
- If the keypair identified by *pName is a DSA keypair, *pPKARB must be a DSA operation request block (an item of type sccDSA_RB_t) whose options, sig_token, sig_token_size, data, and data_size fields are initialized as required by sccDSA (see “sccDSA - Sign Data or Verify Signature for Data” on page 3-57 for details). The remaining fields in *pPKARB are ignored.

The options field of the request block determines whether the cryptographic operation is performed using the public half of the keypair or the private half. The request block must conform to the key used in the operation as required by sccRSA or sccDSA, as appropriate.

PKARBlen is the length in bytes of the request block referenced by pPKARB (that is, sizeof(sccRSA_RB_t) if the keypair identified by *pName is an RSA keypair and sizeof(sccDSA_RB_t) if the keypair identified by *pName is a DSA keypair).

pMsgID determines whether the function is performed synchronously or asynchronously:

- If pMsgID is NULL, the function is performed synchronously. The call does not return until the requested information has been retrieved.
- If pMsgID is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the OA Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the area of memory referenced by pName, pPKARB, or ((sccRSA_RB_t *)pPKARB)->data_in and

((sccRSA_RB_t *)pPKARB)->data_out or ((sccDSA_RB_t *)pPKARB)->sig_token and ((sccDSA_RB_t *)pPKARB)->data, as appropriate, before the operation is complete.

Output

On successful exit from this routine:

If pMsgID is NULL,

- if the keypair identified by *pName is an RSA keypair, *(((sccRSA_RB_t *)pPKARB)->data_out) contains
  - *(((sccRSA_RB_t *)pPKARB)->data_in) transformed using the public half of the keypair identified by *pName if ((sccRSA_RB_t *)pPKARB)->options specifies RSA_PUBLIC and
  - *(((sccRSA_RB_t *)pPKARB)->data_in) transformed using the private half of the keypair identified by *pName if ((sccRSA_RB_t *)pPKARB)->options specifies RSA_PRIVATE.
- if the keypair identified by *pName is a DSA keypair and ((sccDSA_RB_t *)pPKARB)->options specifies DSA_SIGNATURE_SIGN, *(((sccDSA_RB_t *)pPKARB)->sig_token) contains the digital signature produced by signing *(((sccDSA_RB_t *)pPKARB)->data) with the private half of the keypair identified by *pName.
- if the keypair identified by *pName is a DSA keypair and ((sccDSA_RB_t *)pPKARB)->options specifies DSA_SIGNATURE_VERIFY, a return code of zero implies that the signature in *(((sccDSA_RB_t *)pPKARB)->sig_token) was produced by signing *(((sccDSA_RB_t *)pPKARB)->data) with the private half of the keypair identified by *pName.

If pMsgID is not NULL, *pMsgID uniquely identifies the message that was sent to the OA Manager to initiate the desired operation. When the operation is complete, the OA Manager will send the coprocessor application a message whose type field (Msg.h.msg_type) contains this identifier and whose first (and only) data item (Msg.msg_data[0]) contains the return code generated by the routine.[101] If the operation was successful, *(((sccRSA_RB_t *)pPKARB)->data_in) or *(((sccDSA_RB_t *)pPKARB)->sig_token) contains the result (if appropriate). The message is placed on the default CP/Q message queue for the task that called sccOAPrivOpAsync.

Return Codes

Common return codes generated by this routine are:

OAGood (i.e., 0)	The operation was successful.
OABadParm	An argument is not valid.
OANotAllowed	*pName does not identify an Application Keypair.
OANotFound	*pName does not identify an OA Certificate that the OA Manager has saved.
OANoSpace	The operation failed due to lack of space.

[101] The return code from the call to sccOAPrivOpAsync indicates whether or not the initial message to the OA Manager was successfully enqueued.

PKADSASigIncorrect The keypair identified by `*pName` is a DSA keypair and `((sccDSA_RB_t *)pPKARB)->options` specifies DSA_SIGNATURE_VERIFY but the signature in `*(((sccDSA_RB_t *)pPKARB)->sig_token)` was not produced by signing `*(((sccDSA_RB_t *)pPKARB)->data)` with the private half of the keypair identified by `*pName`.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccOAVerify - Verify OA Certificate Chain

sccOAVerify verifies that the cryptographic signature that covers the body of one OA Certificate was generated using the private key corresponding to the public key contained in another OA Certificate.

Function Prototype

```
long sccOAVerify(void          *pParent,
                 unsigned long  ParentLen,
                 void          *pChild,
                 unsigned long  ChildLen);
```

Input

On entry to this routine:

pParent must contain the address of a buffer occupied by the OA Certificate that contains the public key that purportedly corresponds to the private key that was used to generate the cryptographic signature that covers the body of the OA Certificate referenced by pChild. See “OA Certificates” on page 3-106 for details.

ParentLen is the length in bytes of the OA Certificate referenced by pParent.

pChild must contain the address of a buffer occupied by the OA Certificate whose cryptographic signature was purportedly generated by the private key that corresponds to the public key contained in the OA Certificate referenced by pParent. See “OA Certificates” on page 3-106 for details.

ChildLen is the length in bytes of the OA Certificate referenced by pChild.

Output

On successful exit from this routine:

A return code of zero implies that the cryptographic signature that covers the body of the OA Certificate referenced by pChild was generated by the private key that corresponds to the public key contained in the OA Certificate referenced by pParent.

A nonzero return code implies that the cryptographic signature that covers the body of the OA Certificate referenced by pChild was not generated by the private key that corresponds to the public key contained in the OA Certificate referenced by pParent.

Notes

SHA1_ISO_COMPLEMENT_RSA Signatures Not Supported

At present, sccOAVerify will not verify the signature in an OA Certificate whose tSig field is SHA1_ISO_COMPLEMENT_RSA.

Using sccOAVerify

A number of e-commerce (the part of e-business that focuses on transactions) and other applications require that an application inside the coprocessor authenticate a message from the host. It is critical that the application perform all the required computations *inside* the coprocessor. sccOAVerify performs the requisite computations on a coprocessor if the message was signed by the OA Manager on

another coprocessor and if that coprocessor's OA Certificate chain is available. (The IBM Class Root Certificate on the two coprocessors must contain the same public key.)

In addition to verifying signatures, an application should examine the various fields in each certificate before concluding that a valid signature implies that the certificate to which the signature attests is meaningful.

Return Codes

Common return codes generated by this routine are:

OAGood (i.e., 0) The signature is valid.

OASigFail The signature is not valid.

OABadParm An argument is not valid.

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

sccOAStatus - Get Coprocessor Status

sccOAStatus either returns information about the status of the coprocessor and the software (if any) that is loaded into each segment or returns the amount of space this status information would occupy.

Function Prototype

```
long sccOAStatusAsync(void          *pBuffer,
                      unsigned long *pLen,
                      unsigned long *pMsgID);

#define sccOAStatus(pb,pl) sccOAStatusAsync(pb,pl,NULL)
```

Input

On entry to this routine:

pBuffer must contain the address of a buffer in which information about the status of the coprocessor can be returned if this information is desired and must be NULL otherwise.

pLen must contain the address of a variable in which an item of type unsigned long can be stored. If pBuffer is not NULL, *pLen must be the length in bytes of the buffer referenced by pBuffer.

pMsgID determines whether the function is performed synchronously or asynchronously:

- If pMsgID is NULL, the function is performed synchronously. The call does not return until the requested information has been retrieved.
- If pMsgID is not NULL, the function is performed asynchronously. The call returns as soon as a message has been sent to the OA Manager instructing it to perform the desired operation. In this case, the caller must not modify, deallocate, or reuse any portion of the area of memory referenced by pBuffer or pLen before the operation is complete.

Output

On successful exit from this routine:

If pMsgID is NULL, *pLen is the length in bytes of the status information. If pBuffer is not NULL, the buffer it references contains a header of type sccOAStatus_t followed by undefined. The fields of the header are set as follows:

- struct_id.name is SCCOASTATUS_T.
- struct_id.version is the value to which SCCOASTATUS_VER is defined in the header file that defines the version of sccOAStatus_t that maps the entry.[102]
- padbytes is two bytes of zeros.
- rom_status contains information about the basic health of the coprocessor and the state of each segment. The fields of rom_status are set as follows:
  - struct_id.name is zero.
  - struct_id.version is zero.

[102] If *v* is an item of type var_t, the address of the item *v* describes is ((char *)&(v)) + v.offset. By convention, if v.offset is zero, the item *v* describes is empty or missing. Also by convention if *x* and *y* are var_t structures and *y* is a part of the item *x* describes, the item *y* describes is also a part of the item *x* describes (that is, "nested" var_t structures describe nested items).

  - pic_version is a version number stored in the coprocessor's programmable interrupt controller. This number matches the value of "PIC ver" reported by the CLU ST command.
  - rom_version is a version number stored in the coprocessor's ROM. This number matches the value of "ROM ver" reported by the CLU ST command.
  - page1_certified is nonzero, indicating that the coprocessor possesses a Device Keypair and an OA Certificate for the keypair signed by the appropriate IBM Class Root private key.
  - boot_count_left should be zero, indicating that the coprocessor has never wiped all nonvolatile memory upon detection of a tamper event.
  - boot_count_right is the current value of the coprocessor's boot counter. See "Coprocessor Architecture" on page 3-95 for details.
  - adapterID is a serial number that uniquely identifies the coprocessor. It matches the value of the AdapterID field returned by sccGetConfig. See "sccGetConfig - Get Coprocessor Configuration" on page 3-89 for details.
  - vpd is a description of the coprocessor. The first 128 bytes match the value of the AMCC_EEPROM field returned by sccGetConfig and the next 128 bytes match the value of the VPD field returned by sccGetConfig. See "sccGetConfig - Get Coprocessor Configuration" on page 3-89 for details.
  - init_state is 0x01.
  - seg2_state and seg3_state indicate the status of segment 2 and segment 3, respectively. Possible values are 0 (UNOWNED), 1 (OWNED_BUT_UNRELIABLE), 2 (RUNNABLE), and 3 (RUNNABLE_BUT_UNRELIABLE). Refer to Appendix F, Using Signer and Packager of the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer's Toolkit Guide* for details.
  - owner2 and owner3 are the owner identifiers associated with segment 2 and segment 3, respectively. An owner identifier is undefined if the corresponding segment is UNOWNED. Refer to "Appendix F, Using Signer and Packager" in the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer's Toolkit Guide* for details.
  - active_seg1 indicates which half of the memory dedicated to segment 1 will be overwritten the next time the software in segment 1 is reloaded. (This scheme permits segment 1 to be reloaded in an atomic fashion.)
- vSeg_ids undefined.
- free_space indicates the amount of free code and system space in each segment. This is the total size in bytes of the segment minus the size in bytes of the code, public key, and other information that the system software in segment 1 has saved in the segment. The first entry in the array (that is, free_space[0]) specifies the amount of free space in segment 1, the second entry in the array specifies the amount of free space in segment 2, and the third entry in the array specifies the amount of free space in segment 3.

  The figure for segment 3 has no relation to the amount of space in flash available to hold data items the PPD Manager stores on behalf of an application unless the figure is negative. In that case, the system software in segment 1 has found it necessary to "borrow" some space that would ordinarily be at the disposal of the PPD Manager. The absolute value of the figure for segment 3 is the number of bytes that were borrowed in this manner.
- layer_name is an array of identifiers that uniquely identify the software loaded into each segment of the coprocessor. See "Layer Names and Layer Descriptors" on page 3-114 for details.

The first entry in the array (that is, `layer_name[0]`) identifies the software in segment 1, the second entry in the array identifies the software in segment 2, and the third entry in the array identifies the software in segment 3.

If `pMsgID` is not NULL, `*pMsgID` uniquely identifies the message that was sent to the OA Manager to initiate the desired operation. When the operation is complete, the OA Manager will send the coprocessor application a message whose type field (`Msg.h.msg_type`) contains this identifier and whose first (and only) data item (`Msg.msg_data[0]`) contains the return code generated by the routine.[103] If the operation was successful, `*pLen` (and `*pBuffer`, if appropriate) contain the result. The message is placed on the default CP/Q message queue for the task that called sccOAStatusAsync.

Return Codes

Common return codes generated by this routine are:

OAGood (i.e., 0) The operation was successful.

OABadParm An argument is not valid.

OANoSpace The operation failed due to lack of space (for example, `*pLen` is too small to hold the entire status structure)

Refer to *scc_err.h* and the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for a comprehensive list of return codes.

[103] The return code from the call to sccOAStatusAsync indicates whether or not the initial message to the OA Manager was successfully enqueued.

Chapter 4. Coprocessor Interface for Host Device Drivers

The coprocessor provides an interface that allows a host device driver to initialize the coprocessor, test its operation, and transfer data from a host application to the coprocessor and from the coprocessor to a host application. The device driver and coprocessor communicate across the host's PCI bus using the facilities provided by the AMCC S5933 PCI Controller chip (or AMCC).[1] This chapter describes the events that occur when the coprocessor is reset, and the protocol the host device driver uses to send commands and data to the coprocessor and to obtain results and status information from the coprocessor. The functions, types, and constants described in this chapter are contained in *scctypes.h*, *scc_host.h*, *scc_int.h*, *scc_dd.h*, or *knownans.h*. These files are included in the IBM 4758 Application Program Development Toolkit. Refer to the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer's Toolkit Guide* for details.

PCI Communication

The host issues commands to the coprocessor and the coprocessor issues commands and notifications to the host via a set of registers (called mailboxes) provided by the AMCC. Each side of the interface has four 32-bit read/write outgoing mailboxes and four 32-bit read-only incoming mailboxes. The outgoing mailboxes on one side of the interface are connected to the incoming mailboxes on the other side of the interface, as shown in the following illustration.

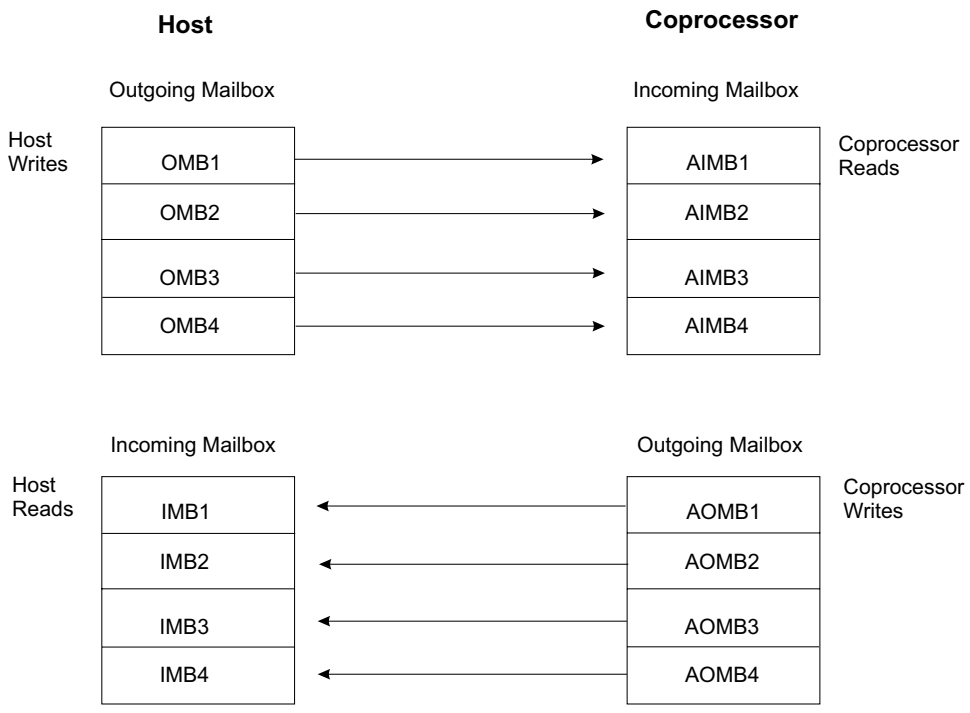

Figure 4-1. AMCC Mailbox Registers

[1] See "Custom Software Publications" on page ix for information on how to obtain technical details about the AMCC S5933.

On the host side mailboxes are byte-, word-, and dword-addressable. That is, the host device driver can write to or read from each byte in a mailbox, either of the 16-bit words in a mailbox, or the entire 32-bit mailbox in a single operation. On the coprocessor side, mailboxes are word- and dword-addressable.

The coprocessor may ask the host to send a block of data (for example, the contents of a buffer a host application has made available in a call to sccRequest) or may send a block of data to the host (for example, the contents of a buffer a coprocessor application has passed to sccPutBufferData). The data is exchanged using the AMCC's FIFOs. The host side of the transfer may be effected by configuring the AMCC to act as initiator on the PCI bus (busmastering) or by direct reads from or writes to the FIFOs (programmed I/O). The coprocessor side of the transfer may be effected by reading from or writing to the AMCC FIFOs using programmed I/O or the 2KB IBM 4758 FIFOs[2] using DMA or programmed I/O.[3] Refer to chapter 11 of the *AMCC S5933 PCI Controller Data Book* for details. The remainder of this chapter assumes that the host uses busmastering.

The host operating system determines how the host device driver reads and writes mailboxes, FIFOs, and control and status registers on the AMCC. For example, the operating system may allow the device driver direct access to the hardware, or the operating system may supply a system call for this purpose.

---

2 See Figure 4-2 on page 4-13.

3 Software on the coprocessor normally accesses the AMCC FIFOs indirectly through the IBM 4758 FIFOs because DMA and the associated interrupts are available only when using the IBM 4758 FIFOs.

Use of the Mailboxes

The host writes commands to the coprocessor into OMB4 byte 2.[4] Arguments associated with the command are written into the remaining OMB registers. 16-bit arguments are written into the outgoing mailboxes in the following order:

1. OMB4 low-order word (bytes 1 and 0)
2. OMB3 high-order word (bytes 3 and 2)
3. OMB3 low-order word
4. OMB2 high-order word
5. OMB2 low-order word
6. OMB1 high-order word
7. OMB1 low-order word

32-bit arguments are written into the outgoing mailboxes in the following order: OMB3, OMB2, OMB1. If a command has both 16-bit and 32-bit arguments, the arguments are written in the required order, which may cause the low-order word of an outgoing mailbox to be unused. For example, the arguments for a command whose first two arguments are 16 bits and whose third argument is 32 bits would be written into OMB4 low-order word, OMB3 high-order word, and OMB2, respectively. OMB3 low-order word would be unused in this case.

Similarly, the coprocessor writes commands and notifications into AOMB4 byte 2. Additional information associated with the command or notification is written into the remaining AOMB registers in the same order that the host uses for arguments. For example:

- a single 16-bit piece of information is written into AOMB4 low-order word, and
- two 16-bit pieces and a 32-bit piece are written into AOMB4 low-order word, AOMB3 high-order word, and AOMB2.

Tamper Status Bits

IMB4 byte 3 and AIMB4 byte 3 always reflect the status of the tamper status bits. Values written into OMB4 byte 3 and AOMB4 byte 3 are ignored. The host should check the value of these bits at the start of initialization and any time the host writes a command into OMB4 byte 2 and the coprocessor fails to read AIMB4 byte 2 within a reasonable period. (The device drivers supplied by IBM timeout after three seconds.)

The bits are active low and from most significant to least significant are: [5]

- 0x80 - (always zero)
- 0x40 - Intrusion Latch
- 0x20 - X-ray Tamper or Dead Battery
- 0x10 - Temperature Tamper
- 0x08 - Overvoltage Tamper
- 0x04 - Coprocessor reset signal asserted[6]

---

4 The low order byte is byte 0.

5 The values given are for the initial version of the coprocessor and may change as changes are made to the design of the hardware.

6 This bit is asserted if the Add-on reset pin (SYSRST#) is asserted. It is also asserted whenever any of the “Tamper” bits is asserted.

- 0x02 - Mesh Tamper
- 0x01 - Low Battery Warning Latch

Only the “Tamper” bits (0x3A) need be examined, since the other conditions do not reset the coprocessor.

Mailbox Overrun

After writing a command into OMB4 byte 2, the device driver must not write a second command until the coprocessor has finished reading the first. By convention, outgoing mailbox 4 byte 2 (OMB4 byte 2 or AOMB4 byte 2) is always written last and incoming mailbox 4 byte 2 (IMB4 byte 2 or AIMB4 byte 2) is always read last.[7] Thus, once the device driver determines that OMB4 byte 2 has been read, the driver may write another command into the OMB registers.

The device driver may either use the AMCC Interrupt Control/Status Register (INTCSR) to configure the AMCC to generate an interrupt when OMB4 byte 2 is read or may poll the Mailbox Empty/Full Status Register (MBEF) to determine when OMB4 byte 2 has been read. The remainder of this chapter assumes that the driver has adopted the first approach.

Use of the FIFOs

When a host application sends a request to a coprocessor application, the host application can supply as many as four buffers of data that the coprocessor application may read by calling sccGetBufferData and as many as four buffers to which the coprocessor application may write data by calling sccPutBufferData or sccEndRequest.[8] When a coprocessor application requests a transfer (in either direction), the coprocessor issues a request to the host device driver that identifies the buffer to be transferred and indicates the direction of transfer. The host device driver writes the physical address of the buffer and the number of bytes to transfer to the appropriate AMCC PCI Controlled Bus Master registers (MWAR and MWTC for transfers from the coprocessor to the host and MRAR and MRTC for transfers from the host to the coprocessor). The transfer takes place when the host device driver sets the Read Transfer Enable bit or the Write Transfer Enable bit (whichever applies) in the AMCC Master Control/Status Register (MCSR).[9]

A buffer may be arbitrarily long, so the host device driver typically breaks each buffer into a number of fixed-sized pieces and transfers each piece. The device driver must ensure that each piece is paged in (that is, pinned in physical memory) during the entire time transfer of the piece is taking place.

The coprocessor may ask to transfer several buffers in a single request or may issue a request before a transfer initiated by a previous request has been completed. The host device driver must ensure that it transfers buffers in the order they are requested. However, transfers *to* the host and transfers *from* the host are

7 Reading or writing mailboxes in sequence and as dwords ensures this convention is followed.

8 This assumes the coprocessor operating system is CP/Q++. If another operating system is used, a coprocessor application will likely use different functions to exchange data with a host application.

9 It appears bits 8 and 12 of the MCSR must both be set (that is, the transfer priority must alternate between reads and writes) in order for busmaster transfers to work properly. This may be due to a bug in the AMCC.

independent operations, so the host device driver may maintain two queues of pending transfers - one for incoming buffers and one for outgoing buffers.

Host-Generated Commands

This section describes the commands the host device driver issues when a host application wants to send a request to a coprocessor application and when the device driver finds it necessary to abort a request before it is complete.

GOT_HEADERS - Signal Pending Requests

The host device driver sends a GOT_HEADERS command to the coprocessor to indicate the host device driver has one or more requests from one or more host applications that have not yet been sent to the coprocessor. This command takes a single 16-bit argument: the number of requests pending.

When a host application calls sccRequest, the host device driver uses the information in the request block to construct a request header (of type `sccRequestHeader_t`), which then awaits transmission to the coprocessor just like any other outgoing data buffer. Most of the request header fields are simply copied directly from the request block (of type `sccRB_t`). However, the host device driver does stamp each header with a `RequestID` (RID), a number that uniquely identifies the request.

The device driver sets the `RequestID` field of the request header to an arbitrary number that must distinguish the request from any other outstanding request (both currently outstanding requests and requests that may be issued before the coprocessor completes its reply to this request). The low-order four bits of this value must be zeros.[10]

The device driver sets the `rsvd` field of the request header as follows:

- The low-order bit is a flag (IGNORE_REQUEST) that indicates the request header and the request to which it applies are no longer valid. This bit may be used during abort processing. See “Abort Processing” on page 4-9 for details.
- The remaining bits of the field are reserved and must be zero.

The coprocessor copies the request header when a coprocessor application calls sccGetNextHeader and so the host device driver must ensure all arithmetic fields in the request header (`RequestID`, `rsvd`, `UserDefined`, and each element of `OutBufferLength` and `InBufferLength`) are in little-endian order (low order byte at lowest address) before the request header is sent to the coprocessor.

Once the host device driver has sent a GOT_HEADERS command to the coprocessor, the host device driver may not send another GOT_HEADERS command until the coprocessor has sent a START_BUFFERS command to tell the host device driver to transfer one or more headers for pending requests. The host device driver may then send another GOT_HEADERS command (even before the first header is transferred); in this case, the count of pending requests passed as

[10] The host device driver may find it expedient to use an index into a table of active requests to identify each request. It is unlikely, however, that the table will contain 4096 entries, so several of the bits in the high-order word will be unused. In this case, the unused bits could be used as an “incarnation” count, starting at zero and incrementing each time the index is reused. This will allow the host device driver to detect an erroneous attempt to transfer a buffer associated with a request that has already ended.

an argument must not include the requests whose headers are to be sent to the coprocessor in response to its START_BUFFERS command.

The host device driver must transmit request headers to the coprocessor in the order in which the associated requests were received from host applications.

ABORT_REQUEST - Abort a Specific Request

The host device driver sends an ABORT_REQUEST command to the coprocessor to abort a specific request. It takes a single 16-bit argument: the identifier of the request to be aborted (as passed in the `RequestID` field of the request header for the request). See “Abort Processing” on page 4-9 for details on the use of this command.

ABORT_END - Signal End of Abort Requests

The host device driver sends an ABORT_END command to the coprocessor to notify the coprocessor that the host device driver has no more requests to abort. It takes no arguments. See “Abort Processing” on page 4-9 for details on the use of this command.

Coprocessor - Generated Commands and Notifications

This section describes the commands and notifications information the coprocessor sends to the host device driver.

START_BUFFERS - Transfer Data Buffers

The coprocessor sends a START_BUFFERS command to the host device driver to tell the device driver to schedule the transfer of one or more buffers of data between the host and the coprocessor. The device driver adds each buffer to the appropriate transmission queue (incoming or outgoing) and writes the appropriate values to the appropriate AMCC PCI Controlled Bus Master registers to effect the transfer of each buffer after all its predecessors on the queue have been transferred.

This command is actually a family of commands, since the low-order nibble of the command indicates how many buffers the coprocessor wants to transfer. START_BUFFERS is #defined to be 0x20 in *scc_dd.h*. The coprocessor writes 0x21 into AOMB4 byte 2 if it wants to transfer a single buffer, 0x22 to transfer two buffers, and so on. A single START_BUFFERS command may schedule the transfer of as many as seven buffers.

Associated with a START_BUFFERS command are the buffer IDs (BIDs) for each of the buffers to be transferred. The number of bytes to transfer is also provided for transfers from the coprocessor to the host.

A BID is a 16-bit quantity that uniquely identifies a buffer. The low-order 4 bits of a BID indicate whether the transfer is from the host to the coprocessor or from the coprocessor to the host and identify the data to be transferred, as shown below:

- **(BID & 0x0C) == 0x00 - Transfer outgoing buffer**

  The coprocessor sets the top half of the low-order nibble of a BID to zero to schedule the transfer of a data buffer from the host. The high-order 12 bits of the BID contain the high-order 12 bits of the RID of the request with which the

buffer is associated.[11] The bottom half of the low-order nibble of the BID is the index of the desired buffer (that is, the value passed by the coprocessor application in the `bufIdx` argument in the call to sccGetBufferDataAsync that caused the transfer to be requested).

The number of bytes transferred is the number of bytes in the buffer as specified by the host application in the `pRequestBlock->OutBufferLength[BID & 0x03]` argument in the call to sccRequest that made the buffer available to the coprocessor application.

- **(BID & 0x0C) == 0x04 - Transfer incoming buffer**

  The coprocessor sets the top half of the low-order nibble of a BID to binary 01 to schedule the transfer of a data buffer to the host. The high-order 12 bits of the BID contain the high-order 12 bits of the RID of the request with which the buffer is associated. The bottom half of the low-order nibble of the BID is the index of the desired buffer (that is, the value passed by the coprocessor application in the `bufIdx` argument in the call to sccPutBufferDataAsync that caused the transfer to be requested).

  The number of bytes to transfer (a 32-bit quantity) is passed immediately following the BID. Thus, a BID of this type cannot appear in IMB1. The number of bytes to transfer will be greater than zero and less than or equal to the number of bytes in the buffer as specified by the host application in the `pRequestBlock->InBufferLength[BID & 0x03]` argument in the call to sccRequest that made the buffer available to the coprocessor application.

- **(BID & 0x0C) == 0x0C - Transfer final incoming buffer**

  The coprocessor sets the top half of the low-order nibble of a BID to binary 11 to schedule the transfer of a data buffer that marks the end of a request to the host. The high-order 12 bits of the BID contain the high-order 12 bits of the RID of the request with which the buffer is associated. The bottom half of the low-order nibble of the BID is the index of the desired buffer (that is, the value passed by the coprocessor application in the `bufIdx` argument in the call to sccEndRequest that caused the transfer to be requested).

  The number of bytes to transfer (a 32-bit quantity) is passed immediately following the BID and the request status[12] (a 32-bit quantity) is passed immediately following the number of bytes to transfer. Thus, a BID of this type cannot appear in IMB1 or IMB2. The number of bytes to transfer will be greater than or equal to zero and less than or equal to the number of bytes in the buffer as specified by the host application in the `pRequestBlock->InBufferLength[BID & 0x03]` argument in the call to sccRequest that made the buffer available to the coprocessor application. If the number of bytes to transfer is zero, the bottom half of the low-order nibble of the BID should be ignored.

---

[11] Recall that the low-order 4 bits of a RID must be zero.

[12] The value passed by the coprocessor application in the `status` argument in the call to sccEndRequest that caused the transfer to be requested.

- **(BID & 0x0C) == 0x08 - Special case transfers**

  The coprocessor sets the top half of the low-order nibble of a BID to binary 10 for certain special case transfers. The bottom half of the low-order nibble of the BID identifies the desired operation, as follows:

  – **(BID & 0x03) == 0x00 - Send headers**

    The coprocessor sets the low-order nibble of a BID to 0x08 to schedule the transfer of a buffer from the host containing one or more request headers. The high-order byte of the BID contains the number of headers to send and will be less than or equal to the number of pending requests argument in the GOT_HEADERS command most recently sent by the host device driver. The remaining bits of the BID are reserved and set to zero.

  – **(BID & 0x03) != 0x00 - Reserved**

    The other special case transfer constants (0x09, 0x0A, and 0x0B) are reserved.

Buffers are added to the appropriate queue in the order their BIDs appear in the START_BUFFERS request as described in “Use of the FIFOs” on page 4-4.

For example, the following command schedules the transfer of outgoing buffer 1 from request 0x220, 0x12345678 bytes to incoming buffer 2 from request 0x1120, and outgoing buffer 0 from request 0x3320:

```
IMB4 - 0xtt232221  (tt are the Tamper Status bits)
IMB3 - 0x11260000
IMB2 - 0x12345678
IMB1 - 0x33200000
```

The device driver must send request 0x220 outgoing buffer 1 before it sends request 0x3320 outgoing buffer 0. The transfer of data for request 0x1120 incoming buffer 2 is scheduled independently.

The following command schedules the transfer of 0x00112233 bytes to incoming buffer 0 of request 0x1230 (and signifies the end of the request and returns a status of 0xDEAD0BAD), the next five request headers, and outgoing buffer 2 from request 0x4560:

```
IMB4 - 0xtt23123C  (tt are the Tamper Status bits)
IMB3 - 0x00112233
IMB2 - 0xDEAD0BAD
IMB1 - 0x05084562
```

The device driver must send the request headers before it sends request 0x4560 outgoing buffer 2. Again, the transfer of data for request 0x1230 incoming buffer 0 is scheduled independently.

INVALID_MB_CMD - Command Not Recognized

The coprocessor sends an INVALID_MB_CMD notification to the host device driver if the coprocessor reads a command it does not recognize in AIMB4 byte 2. The notification is sent before the coprocessor reads any other commands from the host. Associated with this notification is the command that was not recognized

(which is written into AOMB4 byte 0).[13] The coprocessor ignores the unrecognized command.

Upon receipt of this notification, the host device driver should evaluate the problem and proceed accordingly. In general, the device driver should mark the coprocessor as “bad” and discontinue use of the coprocessor.

ABORT_COMPLETE - Request Successfully Aborted

The coprocessor sends an ABORT_COMPLETE notification to the host device driver after the coprocessor has finished processing an ABORT_REQUEST command. Associated with this notification is the 16-bit request identifier passed as an argument in the ABORT_REQUEST command.[13] See “Abort Processing” for details on the use of this notification.

GOODNIGHT_JUAN - System Error Occurred

The coprocessor sends a GOODNIGHT_JUAN notification to the host device driver if an unhandled exception occurs in one of the CP/Q++ extensions. Possible causes include hardware failure and breach of security. The coprocessor then halts and will not respond to further commands until it is reset. The host device driver must cancel all pending requests.

The following status information is supplied with this notification:

- CP/Q fault type and task stop code (16 bits)
- Final coprocessor status code (32 bits)
- Reserved (32 bits)
- CP/Q supervisor ID (SVID) of the task in which the exception occurred (32 bits)

CPQ_ABEND - Kernel Error Occurred

The coprocessor sends a CPQ_ABEND notification to the host device driver if a problem in the CP/Q++ kernel makes continued operation of the coprocessor infeasible. Possible causes include hardware failure and exhaustion of kernel resources. The coprocessor then halts and will not respond to further commands until it is reset. The host device driver must cancel all pending requests.

The following status information is supplied with this notification:

- CP/Q abend code (32 bits)
- CP/Q supervisor ID (SVID) of the kernel component in which the exception occurred (32 bits)

Abort Processing

The host device driver may find it necessary to abort a request from a host application, either because the host application terminates before the result of the request can be delivered or because the device driver encounters a problem while processing the request. There are several possibilities:

If the host device driver has not yet notified the coprocessor that the request is outstanding (that is, has not yet sent the coprocessor a GOT_HEADERS command

[13] Some of the first IBM 4758 coprocessors produced do not supply this information.

whose argument includes the request in the count of outstanding requests), the host device driver can simply discard the request. No interaction with the coprocessor is necessary.

If the host device driver has notified the coprocessor that the request is outstanding but has not yet sent the request header associated with the request to the coprocessor, the host device driver can either set the IGNORE_REQUEST flag in the header's `rsvd` field or replace the request header with the header for another outstanding request whose existence has not already been announced to the coprocessor. No further interaction with the coprocessor is necessary.

If the host device driver has sent the request header to the coprocessor and the response to the request has not yet been transferred to the host from the coprocessor (that is, no START_BUFFERS command has been received that includes a "transfer final incoming buffer" BID containing the request's identifier or such a request has been received but the transfer has not yet begun), the host device driver must wait until any buffer transfer from the host to the coprocessor presently underway is complete (that is, must wait for the current busmaster read to complete), then send an ABORT_REQUEST command containing the identifier of the request to abort. The coprocessor will respond with an ABORT_COMPLETE notification. The coprocessor will not subsequently issue any START_BUFFERS command containing a BID that refers to the aborted request.

Once the coprocessor receives an ABORT_REQUEST command, it enters an "abort processing" mode. While in this mode, the coprocessor will not issue any START_BUFFERS commands and expects the host to issue either additional ABORT_REQUEST commands or an ABORT_END command. Upon receipt of an ABORT_END command from the host, the coprocessor resumes normal operations.

If the host device driver encounters a problem while sending a buffer to the coprocessor, the host device driver can simply stop the busmaster read (that is, clear the Read Transfer Enable bit in the AMCC Master Control/Status Register (MCSR)) and send an ABORT_REQUEST command for the request with which the buffer is associated. In other words, transfers from the host to the coprocessor need not be completed. (Upon receipt of an ABORT_REQUEST command, the coprocessor continues to read data from the FIFOs until the FIFOs are empty. Only then does the coprocessor begin its abort processing.)

However, the host cannot exercise such fine control over transfers from the coprocessor to the host. Once the coprocessor has written data to the FIFOs, the transfer cannot be stopped or interrupted. After sending an ABORT_REQUEST command, the host must continue to read data from the coprocessor according to the pre-existing schedule until an ABORT_COMPLETE notification is received.

If a buffer provided by the host application to receive data from the coprocessor proves unsuitable (for example, cannot be pinned in memory), the host device driver must supply a buffer of its own of the appropriate size so that the data may be read from the FIFOs, and the FIFOs allowed to drain.

Once the host device driver has successfully aborted a request (for example, upon receipt of an ABORT_COMPLETE notification), the host device driver can cancel any scheduled transfers of buffers associated with the request.

Initialization

Following power-on or reset, the coprocessor performs a four-phase power-on self test (POST) and initialization sequence. The host device driver must interact with the coprocessor during this time to test the data path between the host to the coprocessor and to enable the transition from one phase to the next. The sequence is as follows:

- The coprocessor is reset, either by hardware power-on or by the host device driver using bits 24 through 27 of the AMCC Master Control/Status Register (MSCR). The host device driver should start a timer anytime the coprocessor is reset in order to detect problems encountered during the first phase of the power-on self test (POST 0). Due to testing of the random number generator, it takes approximately 25 seconds from the removal of reset before POST 0 sends the first notification to the host device driver.

  In the remainder of this section we will assume that the host device driver sets the appropriate bits in the AMCC Interrupt Control/Status Register (INTCSR) to cause an interrupt on the host whenever the coprocessor writes into AOMB4 byte 2.

- Upon reset, the coprocessor begins the first phase of the power-on self test (POST 0). POST 0 initializes the CPU chipset, including the interrupt controller and CPU clock generator, and ensures the coprocessor hardware can perform its basic functions properly. Among the items tested are the CPU (including cache), the interrupt controller, the realtime clock, the DES chip, the serial UART, the onboard RAM, and the random number generator. POST 0 also verifies the ROM and CMOS checksums and clears all the mailboxes to zero, which may generate an interrupt on the host. The host device driver should ignore this notification if it occurs.[14]

- Upon completion of all its standalone tests, POST 0 writes 0x600D into AOMB4 low-order word and a POST0_ACTION notification into AOMB4 byte 2, which should generate an interrupt on the host. If the timer started by the host device driver when the coprocessor is reset expires before an interrupt occurs, POST 0 has failed. POST 0 writes a checkpoint identifier into AOMB4 low-order word at the completion of each stand-alone test, so if the timer expires the host device driver can examine IMB4 low-order word to determine the point at which the failure occurred.

  After writing the POST0_ACTION notification, POST 0 monitors AOMB4 byte 2 and waits for the host device driver to read the notification. Once this occurs, POST 0 delays 100 milliseconds and then begins a “walking 1’s” test of the AMCC mailboxes, as follows:

  1. POST 0 writes 0x00000001 to AOMB1.
  2. The host device driver determines that IMB1 has changed, either by polling the AMCC Mailbox Empty/Full Status Register (MBEF), configuring the AMCC to generate an interrupt when AOMB1 is written, or using a timeout.[15] The device driver then reads IMB1, ensures that the proper value (0x00000001) was read, and writes the same value into OMB1.[16]

---

[14] Some of the first IBM 4758 coprocessors produced also write 0x34 into AOMB4 byte 2. The last device driver should ignore this notification if it occurs.

[15] A timeout is required in any event to verify that the MBEF register or interrupt generation logic is functioning properly.

[16] The device driver should also verify that IMB2, IMB3, and IMB4 are zero, although the drivers supplied by IBM do not do so.

3. When POST 0 determines that AIMB1 has changed, it reads AIMB1 and ensures that the proper value was read. Steps 1, 2, and 3 are then repeated for each bit in mailbox 1 (that is, POST 0 writes 0x00000002, then 0x00000004, then 0x00000008, and so on).
4. Once all the bits in mailbox 1 have been tested, steps 1 through 4 are repeated for mailboxes 2, 3, and 4 in that order. Only 24 bits of mailbox 4 can be tested since the upper 8 bits always reflect the state of the tamper status bits.

If the walking 1's test is successful, POST 0 writes 0x600E into AOMB4 low-order word and waits for the host device driver to begin testing the AMCC FIFO registers.

- To test the AMCC FIFO registers, the host device driver writes the following eight 32-bit patterns into the AMCC FIFO Register Port (FIFO), delaying 10 milliseconds between each pattern: 0x00000000, 0xFFFFFFFF, 0x12345678, 0x9ABCDEF0, 0x5A5A5A5A, 0xA5A5A5A5, 0x5555AAAA, and 0xAAAA5555. POST 0 validates each pattern as it is received and sends it back through the FIFOs.

  After the host device driver writes the last of the patterns, it reads data from the FIFOs and validates each pattern received. The host device driver continues reading data until the AMCC MCSR indicates the Add-on to PCI FIFO is empty. The host device driver then writes 0x00000000 to the FIFO twice. This notifies POST 0 that the FIFO test is complete.

  The host device driver then waits 10 milliseconds, resets the AMCC Add-on to PCI FIFO, waits another 10 milliseconds, and resets the AMCC PCI to Add-on FIFO.

- After POST 0 reads from the FIFO the two zeros that mark the end of the FIFO tests, POST 0 writes a POST_WARNING notification into AOMB 4 byte 2. This indicates the end of POST 0. POST 0 then monitors AOMB4 byte 2 and waits for the host device driver to read the notification. Once all the mailboxes are empty, POST 0 transfers control to the first phase of system initialization, Miniboot 0.

- Miniboot 0 writes zeros into AOMB4 low-order word and MBOOT_0_ACTION into AOMB4 byte 2 to notify the host device driver that Miniboot 0 has started. The host device driver should respond by writing 0x00000000 to OMB4.[17]

  Miniboot 0 proceeds to initialize the system to the point that the status of the coprocessor can be obtained, the code that implements POST 1 and Miniboot 1 can be updated if appropriate, and the coprocessor can be re-enabled after it has detected an attempt to tamper with the hardware and shut down. If Miniboot 0 encounters a problem, it writes a halt code into AOMB4 low-order word, writes GOODNIGHT_DAVE into AOMB4 byte 2, and halts. No further activity occurs until the coprocessor is reset. If Miniboot 0 completes its work successfully, it writes 0xFFFF into AOMB4 low-order word and MBOOT_0_ACTION into AOMB4 byte 2. Miniboot 0 then transfers control to the second phase of the power-on self test, POST 1.

  Upon receipt of notification of the end of Miniboot 0, the host device driver should start a timer in order to detect problems encountered during POST 1.

[17] The host device driver may enter "Miniboot mode" by sending a command rather than zeros. See "Miniboot Mode" on page 4-16 for details.

- POST 1 tests the coprocessor hardware that POST 0 did not including interrupts, DMA, and the RSA chip.
- Upon completion of all its stand-alone tests, POST 1 writes 0x600F into AOMB4 low-order word and a POST1_ACTION notification into AOMB4 byte 2, which should generate an interrupt on the host. If the timer started by the host device driver at the end of Miniboot 0 expires before the POST1_ACTION notification is received, POST 1 has failed. Like POST 0, POST 1 writes a checkpoint identifier into AOMB4 low-order word at the completion of each stand-alone test, so if the timer expires the host device driver can examine IMB4 low-order word to determine the point at which the failure occurred.
- Upon receipt of the POST1_ACTION notification with 0x600F in IMB4 low-order word, the host device driver should write 0x600F into OMB4 low-order word and 0x00 into OMB4 byte 2 as an acknowledgement. The host device driver may then test the data paths shown in Figure 4-2.

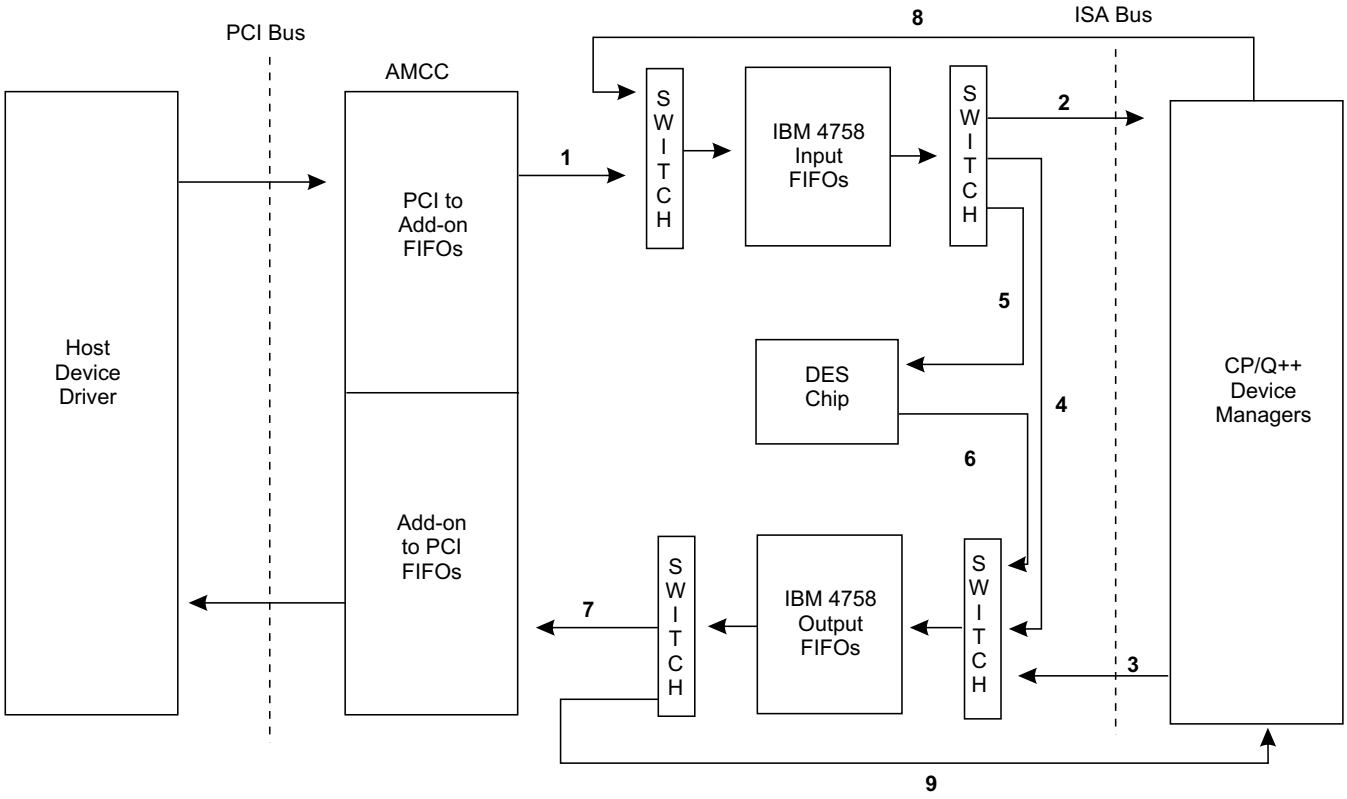

Figure 4-2. Host-Coprocessor Data Paths

The host device driver initiates a test by ensuring the AMCC FIFOs are empty. The host device driver then writes the number of words of data to transfer (that is, the number of bytes of data divided by 2) into OMB1, the number of times POST1 is to perform the test into OMB2, and values that depend on the test to be performed into OMB3 and OMB4. The host device driver then waits for POST1 to read OMB4 byte 2 (which should happen almost immediately) and waits for POST1 to write an acknowledgement of zeros into AOMB4. (If POST1 fails to perform either of these operations there is a problem.) The host device driver then performs the following actions the number of times POST1 is to perform the test:

1. Sets up the AMCC PCI Controlled Bus Master registers to transfer data to the coprocessor (if the host requires data to be transferred to the coprocessor) via a busmaster read and to transfer data from the coprocessor (if the test requires data to be transferred from the coprocessor) via a busmaster write.
2. Sets the Read Transfer Enable bit in the MCSR (if the host requires data to be transferred to the coprocessor) and the Write Transfer Enable bit (if the test requires data to be transferred from the coprocessor) to initiate the busmaster operations.
3. Waits for the AMCC to assert the PCI to Add-on Transfer Count Equals Zero bit in the MCSR (if the host requires data to be transferred to the coprocessor) and the Add-on to PCI Transfer Count Equals Zero bit (if the test requires data to be transferred from the coprocessor), signifying that all data transfers are complete.
4. Clears the MCSR bits set in step 2.
5. Waits for POST1 to write a final completion code into AOMB4 and verifies that the final completion code is zero.
6. Verifies that any data transferred from the coprocessor matches the expected results.

The following tests are supported:

Packet Mode Test

The Packet Mode Test transfers data from the host to the coprocessor, through the IBM 4758 Input FIFOs, across the IBM 4758's ISA bus to the Communications Manager and back to the IBM 4758 Output FIFOs, and back to the host (path 1-2-3-7 in Figure 4-2 on page 4-13). At least 4160 bytes of data should be transferred and the host device driver should verify that the data received matches the data sent.

The host device driver initiates this test by writing POST1_PACKET_CMD into OMB4. OMB3 is not used.

The device drivers supplied by IBM perform this test using a 4160-byte long buffer containing the little-endian 16-bit numbers 0 through 2079 in sequence (that is, 0x0000 0100 0200 0300 ... 1C08 1D08 1E08 1F08).

Bypass Test

The Bypass Test transfers data from the host to the coprocessor, through the IBM 4758 Input FIFOs to the IBM 4758 Output FIFOs, and back to the host (path 1-4-7 in Figure 4-2 on page 4-13). At least 4160 bytes of data should be transferred and the host device driver should verify that the data received matches the data sent.

The host device driver initiates this test by writing POST1_BYPASS_CMD into OMB4. OMB3 is not used.

The device drivers supplied by IBM perform this test three times, once with 4160 bytes of zeros, once with 4160 bytes of 0xFF, and once with a 4160-byte long buffer containing the little-endian 16-bit numbers 0 through 2079 in sequence (that is, 0x0000 0100 0200 0300 ... 1C08 1D08 1E08 1F08).

DES Tests

The DES tests transfer data through the IBM 4758 Input FIFOs to the DES chip and thence to the IBM 4758 Output FIFOs. The input data may be read from the host or from the IBM 4758's ISA bus and the output data may be written to the host or to the IBM 4758's ISA bus. The input data may be encrypted or

decrypted using 0xFEDCBA9876543210 as the key and 0xD776D2F27992341D as the initial vector.

For encryption, the input should be a 512-byte buffer containing the little-endian 16-bit numbers 0 through 255 in sequence (that is, 0x0000 0100 0200 0300 ... FC00 FD00 FE00 FF00). The expected output appears in *knownans.h*.

For decryption, the input should be a 512-byte buffer containing the value specified in *knownans.h*. The expected result is the little-endian 16-bit numbers 0 through 255 in sequence (that is, 0x0000 0100 0200 0300 ... FC00 FD00 FE00 FF00).

The host device driver initiates this test by writing a value that identifies the input and output paths into OMB3 and POST1_DES_ENCRYPT_CMD (to encrypt the input data) or POST1_DES_DECRYPT_CMD (to decrypt the input data) into OMB4. Recognized values for OMB3 are:

– **POST1_P2P - PCI-to-PCI**

  Input data is read from the host and output data is written to the host (path 1-5-6-7 in Figure 4-2 on page 4-13).

– **POST1_P2I - PCI-to-ISA**

  Input data is read from the host and output data is written to the ISA bus (path 1-5-6-9 in figure xx). POST1 verifies that the result matches the expected result in *knownans.h* and writes a nonzero final completion code if it does not.

– **POST1_I2P - ISA-to-PCI**

  Input data is read from the ISA bus and output data is written to the host (path 8-5-6-7 in Figure 4-2 on page 4-13).

– **POST1_I2I - ISA-to-ISA**

  Input data is read from the ISA bus and output data is written to the ISA bus (path 8-5-6-9 in Figure 4-2 on page 4-13). POST1 verifies that the result matches the expected result in *knownans.h* and writes a nonzero final completion code if it does not.

When the host device driver has completed all desired tests, it writes POST1_DONE_CMD into OMB4. POST1 responds by writing zeros into AOMB4 low-order word and into AOMB4 byte 2, then transfers control to the second phase of system initialization, Miniboot 1.

- Miniboot 1 writes zeros into AOMB4 low-order word and MBOOT_1_ACTION into AOMB4 byte 2 to notify the host device driver that Miniboot 1 has started.[18]

  If Miniboot 1 encounters a problem, it writes a halt code into AOMB4 low-order word, writes GOODNIGHT_DAVE into AOMB4 byte 2, and halts. No further activity occurs until the coprocessor is reset. If Miniboot 1 completes its work successfully, it writes 0xFFFF into AOMB4 low-order word and MBOOT_1_ACTION into AOMB4 byte 2. Miniboot 1 then transfers control to the operating system loaded in segment 2.

[18] If the coprocessor is in Miniboot mode, Miniboot 1 writes a START_BUFFERS request for the final transfer of a single buffer (that is, the low-order nibble of the BID in AOMB4 low-order word is 0xC). The host device driver may then issue additional Miniboot commands. See “Miniboot Mode” on page 4-16 for details.)

- When the coprocessor is sufficiently initialized and is prepared for normal interaction with the host (for example, GOT_HEADERS/START_BUFFERS), it writes GOOD_MORNING into AOMB4 byte 2.

Miniboot Mode

A host application may interact with the Miniboot software on the coprocessor during initialization. The host library that exports the functions in the host-side portion of the SCC API also exports two functions that support this interaction: sccMBOpenAdapter and sccMBRequest. These two functions take the same arguments and behave in the same manner as sccOpenAdapter and sccRequest, respectively, but direct the requests to the Miniboot software rather than to an application running on the coprocessor.

A call to sccMBOpenAdapter can be allowed to succeed only if there are no active applications that have called sccOpenAdapter or sccMBOpenAdapter and have not yet called sccCloseAdapter. Once the host device driver has placed the coprocessor in Miniboot mode, the host device driver must return a bad return code (for example, HDDDeviceBusy) if an application subsequently calls sccOpenAdapter or sccMBOpenAdapter.

The commands that may be issued to the Miniboot software and its response to them are proprietary to IBM and can only be described under the terms of an appropriate contract; this section describes the protocol the host device driver uses to pass Miniboot commands to the coprocessor.

To cause the coprocessor to enter Miniboot mode (or to return to Miniboot mode after a reset), the host device driver writes a GOT_HEADERS command with a count of 1 into OMB4 after receiving the MBOOT_0_ACTION notification that indicates Miniboot 0 has started. The host device driver and Miniboot software then exchange commands and requests and transfer buffers using the same protocol that applies when a host application interacts with a coprocessor application. (that is, GET_HEADERS/START_BUFFERS)

Certain Miniboot commands may cause Miniboot 0 to transfer control to POST 1, at which time the normal testing of the coprocessor continues. The host device driver must be able to handle interaction with POST 1 while in Miniboot mode. Furthermore, certain Miniboot commands may cause the Miniboot phase that currently has control (Miniboot 0 or Miniboot 1) to halt and shut down the coprocessor. In this case, Miniboot writes GOODNIGHT_DAVE into AOMB4 byte 2 before halting. Before sending the GOT_HEADERS command for a request, the host device driver must check to see whether Miniboot has halted and, if so, reset the coprocessor and perform the normal POST 0 sequence until Miniboot 0 again gets control.

If a Miniboot phase halts in this manner, the host device driver informs the host application that a halt has occurred.

If Miniboot 1 starts before the coprocessor has sent a START_BUFFERS command to the host for the final transfer for the Miniboot command (that is, Miniboot 1 starts while a Miniboot command is outstanding), Miniboot 1 sends the START_BUFFERS command for the final transfer. The host device driver may then send a GOT_HEADERS command to forward the next request to Miniboot 1.

Note that in order to enter Miniboot Mode the host device driver must send a GOT_HEADERS command at the beginning of Miniboot 0. It is the host application's responsibility to cause the coprocessor to proceed to Miniboot 1 if the host application wants to interact with Miniboot 1 rather than Miniboot 0.

Host - POST/Miniboot Interaction Flow Diagrams

Normal Mode

```
HOST/4758 INTERACTION DURING POST/MINIBOOT       - Normal flow
                                                    (No Miniboot requests)

 ============================================================================


Host Device Driver                                IBM 4758
------------------                   -----------------------------

                      --- Reset -->


                                     POST 0
                                     ------
                    <-      MB4 --    checkpoints

                    <- Irpt MB4 --    xx34xxxx (POST0 message)**

                    <-      MB4 --    checkpoints

                    <- Irpt MB4 --    xx10600D (POST0_ACTION [Start Walking 1's Test])
 do Walking 1's Test*
                    <-      MB4 --    xxxx600E (Start AMCC FIFO Test)
 do AMCC FIFO Test*
                    <- Irpt MB4 --    xx127000 (POST_WARNING [end POST 0])



                                     MiniBoot 0
                                     ----------
                    <- Irpt MB4 --    xx180000 (MBOOT_0_ACTION [Miniboot 0 starting])
 xx000000 (No MB cmds) -- Irpt MB4 ->
                    <- Irpt MB4 --    xx18FFFF (MBOOT_0_ACTION [Miniboot 0 ending])


                                     POST 1
                                     ------
                                         checkpoints
                    <- Irpt MB4 --    xx11600F (POST1_ACTION [Start host demand tests])
 xx00600F (ack)     --      MB4 ->

 (execute host demand                                          ***
  tests)

 00000000           --      MB1 ->
 00000000           --      MB2 ->
 00000000           --      MB3 ->
 xx00000A
  (POST1_DONE_END [end tests])  --      MB4 ->

                    <- Irpt MB4 --     xx000000 (POST 1 ending)****


                                     MiniBoot 1
                                     ----------
                    <- Irpt MB4 --    xx190000 (MBOOT_A_ACTION [Miniboot 1 starting])
```

```
                        <- Irpt MB4 --    xx19FFFF (MBOOT_1_ACTION [Miniboot 1 ending])


                                    CP/Q++
                                    ------
                        <- Irpt MB4 --    xx200000 (GOOD_MORNING)

  (start normal request
   transfers)
```

* The sequences for the Walking 1's test and AMCC FIFO test are in "Walking 1's Test" and "AMCC FIFO Test" on page 4-19.
** May not occur.
*** This will change when we include POST 1 Interactive Tests.
**** This may change to to xx11FFFF.

Walking 1's Test

Test each bit of each 32-bit Mailbox. (MB4 has only 24 testable bits.)

The following example uses host interrupts to detect when the IBM 4758 writes data. For this method, the host driver must enable host interrupts for each Mailbox as it is being tested.

```
  Host Device Driver                        IBM 4758
  ------------------                  ------------------

  wait 100+ ms
                        <-      MB1 --    00000001
  enable MB1 Irpts
  00000001              --      MB1 ->
                        <- Irpt MB1 --    00000002
  00000002              --      MB1 ->
                        <- Irpt MB1 --    00000004
  00000004              --      MB1 ->
    .                       .               .
    .                       .               .
    .                       .               .
                        <- Irpt MB1 --    20000000
  20000000              --      MB1 ->
                        <- Irpt MB1 --    40000000
  40000000              --      MB1 ->
                        <- Irpt MB1 --    80000000
  enable MB2 Irpts
  80000000              --      MB1 ->



                        <- Irpt MB2 --    00000001
  00000001              --      MB2 ->
                        <- Irpt MB2 --    00000002
  00000002              --      MB2 ->
                        <- Irpt MB2 --    00000004
  00000004              --      MB2 ->
    .                       .               .
    .                       .               .
    .                       .               .
                        <- Irpt MB2 --    20000000
  20000000              --      MB2 ->
                        <- Irpt MB2 --    40000000
  40000000              --      MB2 ->
                        <- Irpt MB2 --    80000000
```

```
enable MB3 Irpts
80000000              --      MB2 ->


                      <- Irpt MB3 --    00000001
00000001              --      MB3 ->
                      <- Irpt MB3 --    00000002
00000002              --      MB3 ->
                      <- Irpt MB3 --    00000004
00000004              --      MB3 ->
  .                       .               .
  .                       .               .
  .                       .               .
                      <- Irpt MB3 --    20000000
20000000              --      MB3 ->
                      <- Irpt MB3 --    40000000
40000000              --      MB3 ->
                      <- Irpt MB3 --    80000000
enable MB4 Irpts
80000000              --      MB3 ->


                      <- Irpt MB4 --    xx000001
xx000001              --      MB4 ->
                      <- Irpt MB4 --    xx000002
xx000002              --      MB4 ->
                      <- Irpt MB4 --    xx000004
xx000004              --      MB4 ->
  .                       .               .
  .                       .               .
  .                       .               .
                      <- Irpt MB4 --    xx200000
xx200000              --      MB4 ->
                      <- Irpt MB4 --    xx400000
xx400000              --      MB4 ->
                      <- Irpt MB4 --    xx800000
xx800000              --      MB4 ->
```

AMCC FIFO Test

Test by sending any data pattern to the IBM 4758 through the AMCC PCI to Add-on FIFO. The IBM 4758 sends the data received back through the AMCC Add-on to PCI FIFO. The host then reads the data coming back and compares it to the original pattern to make sure the data path is good.

Finally, the host writes to the AMCC FIFO two more times, clears the addon-to-PCI FIFO flags, and then clears the PCI-to-addon FIFO flag. This is done so the IBM 4758 can test the FIFO full/empty flags and other status bits.

The entire test is paced via timer. Each step is separated by 10 ms.

```
Host Device Driver                              IBM 4758
-----------------                       -----------------------------

Reset FIFO flags
Set FIFO R/W priorities
wait 10 ms
12345678              --     FIFO ->
wait 10 ms
9ABCDEF0              --     FIFO ->
wait 10 ms
5A5A5A5A              --     FIFO ->
wait 10 ms
A5A5A5A5              --     FIFO ->
wait 10 ms
55AA55AA              --     FIFO ->
wait 10 ms
AA55AA55              --     FIFO ->
wait 10 ms
5555AAAA              --     FIFO ->
wait 10 ms
AAAA5555              --     FIFO ->
wait 10 ms
                      <-     FIFO --    12345678  (returns same data pattern)
wait 10 ms
                      <-     FIFO --    9ABCDEF0
wait 10 ms
                      <-     FIFO --    5A5A5A5A
wait 10 ms
                      <-     FIFO --    A5A5A5A5
wait 10 ms
                      <-     FIFO --    55AA55AA
wait 10 ms
                      <-     FIFO --    AA55AA55
wait 10 ms
                      <-     FIFO --    5555AAAA
wait 10 ms
                      <-     FIFO --    AAAA5555
00000000              --     FIFO ->
00000000              --     FIFO ->
wait 10 ms
clear Add-on to PCI FIFO flags
wait 10 ms
clear PCI to Add-on FIFO flags
```

Miniboot Mode

```
HOST/IBM 4758 INTERACTION DURING POST/MINIBOOT       - Miniboot flow
                                                    (with Miniboot requests)
================================================================================


Host Device Driver                              IBM 4758
------------------                     -----------------------------
sccMBOpenAdapter()

sccMBRequest()
                        --- Reset -->


                          POST 0
                          ------

                       <-      MB4 --  checkpoints

                       <- Irpt MB4 --  xx34xxxx (POST0 message)**

                       <-      MB4 --  checkpoints

                       <- Irpt MB4 --  xx10600D (POST0_ACTION [Start Walking 1's Test])
do Walking 1's Test*
                       <-      MB4 --  xxxx600E (Start AMCC FIFO Test)
do AMCC FIFO Test*
                       <- Irpt MB4 --  xx127000 (POST_WARNING [end POST 0])


                         MiniBoot 0
                         ----------

                       <- Irpt MB4 --  xx180000 (MBOOT_0_ACTION [Miniboot 0 starting])
xx200001 (GOT_HEADERS)  -- Irpt MB4 ->
                       <- Irpt MB4 --  xx210108 (START_BUFFERS- send headers)
send header data       --     FIFO ->

                                       (execute MiniBoot 0 commands)
                                       (transfer data as required)


                                       if cmd == query command
                       <- Irpt MB4 --    xx21xxxC (MB-request ending)
                                         get next command (loop)
                            or
                                       if cmd != CONTINUE
                       <- Irpt MB4 --    xx21xxxC (MB-request ending)
                                         halt
                            or

                       <- Irpt MB4 --  xx18FFFF (MBOOT_0_ACTION [Miniboot 0 ending])


                           POST 1
                           ------
                                       checkpoints
                       <- Irpt MB4 --  xx11600F (POST1_ACTION [Start host demand tests])
xx00600F (ack)         --      MB1 ->

(execute host demand                                          ***
 tests)

00000000               --      MB1 ->
00000000               --      MB2 ->
00000000               --      MB3 ->
xx00000A
 (POST1_DONE_CMD [end tests])  --      MB4 ->
                       <- Irpt MB4 --  xx000000 (POST 1 ending)****
```

```
                          MiniBoot 1
                          ----------

                        <- Irpt MB4 --  xx190000 (MBOOT_1_ACTION [Miniboot 1 starting])

                        <- Irpt MB4 --  xx21xxxC (CONTINUE request ending)
  return to application
  sccMBRequest()

  xx200001 (GOT_HEADERS)  -- Irpt MB4 ->
                        <- Irpt MB4 --  xx210108 (START_BUFFERS-send headers)
  send header data        --     FIFO ->

                                        (execute MiniBoot 1 commands)
                                        (transfer data as required)

                        <- Irpt MB4 --  xx21xxxC (request ending)

                             or
                                        if cmd == query command
                        <- Irpt MB4 --    xx21xxxC (request ending)
                                          get next command (loop)
                             or
                                        if cmd != CONTINUE
                        <- Irpt MB4 --    xx21xxxC (request ending)
                                          halt
                             or

                        <- Irpt MB4 --  xx19FFFF (MBOOT_1_ACTION [Miniboot 1 ending])



                            CP/Q++
                            ------

                        <- Irpt MB4 --  xx200000 (GOOD_MORNING)



  sccCloseAdapter()
                        --- Reset -->


  (Start over with POST 0,
   following normal flow.
   See "Normal Mode" on page 4-17.)
```

* The sequences for the “Walking 1’s Test” on page 4-18 and “AMCC FIFO Test” on page 4-19.

** May not occur.

*** This will change when we include POST 1 Interactive Tests.

**** This may change to xx11FFFF.

Host - IBM 4758 Normal Interaction

```
Host Device Driver                              IBM 4758
------------------                      -----------------------------

                        <- Irpt MB4 --  xx200000 (GOOD_MORNING)

sccOpenAdapter()

Loop:

sccRequest()

xx200001 (GOT_HEADERS) -- Irpt MB4 ->
                                        sccGetBufferData ()
                        <- Irpt MB4 --  xx210108 (START_BUFFERS- send header)
send header data        --     FIFO ->

                                        (application receives the header and
                                         starts processing the request, which
                                         will usually result in one or more
                                         requests for host buffer data
                                         transfers.)

                        (zero or more ...)
                                        sccPutBufferData ()
                        <- Irpt MB4 --  xx21???? (START_BUFFERS request)

                                        sccEndRequest ()
                        <- Irpt MB4 --  xx21???C (START_BUFFERS request
                                                  with  ending)

                        goto Loop
```

Appendix A. Error Code Formatting

Return codes for function calls follow the normal CP/Q format:

0xWXYYzzzz

where:

W Eight indicates a negative number; an error has occurred

X Used by the error-generating module; usually zero

YY Code number of the error-generating module

zzzz Actual error code determined by the entity detecting the error

Common code combinations for WXYY:

CP/Q Error Codes

8001 SVC Handler
8002 Memory Manager
8003 Resource Manager
8004 Session Manager
8006 Loader
8007 File API Stubs
8107 File Router
8207 File System
8240 POST Error
8307 Device Router
8407 Device Driver
8507 Server File Router
8607 Client File Router

SCC Error Codes

8041 SCC Manager
8042 Comm Manager
8043 PPD Manager
8044 DES Manager
8045 PKA Manager
8046 RNG Manager

Reserved for IBM Use

806x CCA modules

Programmer Defined

8X8x Used by applications

Note: A return code of zero indicates a successful operation.

Appendix B. DES Weak, Semi-Weak, and Possibly Weak Keys

sccGetRandomNumber (on page 3-65) will not return any of the 64-bit numbers in the following list if the `options` argument specifies RANDOM_NOT_WEAK.

```
01010101 01010101
01011F1F 01010E0E
0101E0E0 0101F1F1
0101FEFE 0101FEFE
011F011F 010E010E
011F1F01 010E0E01
011FE0FE 010EF1FE
011FFEE0 010EFEF1
01E001E0 01F101F1
01E01FFE 01F10EFE
01E0E001 01F1F101
01E0FE1F 01F1FE0E
01FE01FE 01FE01FE
01FE1FE0 01FE0EF1
01FEE01F 01FEF10E
01FEFE01 01FEFE01
1F01011F 0E01010E
1F011F01 0E010E01
1F01E0FE 0E01F1FE
1F01FEE0 0E01FEF1
1F1F0101 0E0E0101
1F1F1F1F 0E0E0E0E
1F1FE0E0 0E0EF1F1
1F1FFEFE 0E0EFEFE
1FE001FE 0EF101FE
1FE01FE0 0EF10EF1
1FE0E01F 0EF1F10E
1FE0FE01 0EF1FE01
1FFE01E0 0EFE01F1
1FFE1FFE 0EFE0EFE
1FFEE001 0EFEF001
1FFEFE1F 0EFEFE0E
E00101E0 F10101F1
E0011FFE F1010EFE
E001E001 F101F101
E001FE1F F101FE0E
E01F01FE F10E01FE
E01F1FE0 F10E0EF1
E01FE01F F10EF10E
E01FFE01 F10EFE01
E0E00101 F1F10101
E0E01F1F F1F10E0E
E0E0E0E0 F1F1F1F1
E0E0FEFE F1F1FEFE
E0FE011F F1FE010E
E0FE1F01 F1FE0E01
E0FEE0FE F1FEF1FE
E0FEFEE0 F1FEFEF1
FE0101FE FE0101FE
FE011FE0 FE010EF1
```

```
FE01E01F FE01F10E
FE01FE01 FE01FE01
FE1F01E0 FE0E01F1
FE1F1FFE FE0E0EFE
FE1FE001 FE0EF101
FE1FFE1F FE0EFE0E
FEE0011F FEF1010E
FEE01F01 FEF10E01
FEE0E0FE FEF1F1FE
FEE0FEE0 FEF1FEF1
FEFE0101 FEFE0101
FEFE1F1F FEFE0E0E
FEFEE0E0 FEFEF1F1
FEFEFEFE FEFEFEFE
```

Appendix C. The IBM Root Public Key

As of the date of this document, the key IBM uses to sign the certificates for the class keys used with the IBM 4758 Model 002 and Model 023 is a 1024-bit RSA key whose public exponent is 65537 (decimal) and whose modulus in hex is as follows:

```
80000000 00000000
00000000 00000010
0CACBAED FCEB4A2D
1FCE8B0F 42AA10DE
B9405685 C800156C
000D4635 811F34D4
375F17F0 3445EC7B
C2516182 20F75391
D0F91FE6 AA52CA9A
463FE87B F78FF842
A770EEC4 B8B07FD5
55BC54DF 194F3FC6
CE1B4936 EE0BAA1E
4E7E6D57 494E8334
26185CD3 6440ED2B
03963DBC 432DF717
```

The most significant byte of the modulus is 0x80 and the least significant byte is 0x17.

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

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Copying and Distributing Softcopy Files

For online versions of this book, we authorize you to:

- Copy, modify, and print the documentation contained on the media, for use within your enterprise, provided you reproduce the copyright notice, all warning statements, and other required statements on each copy or partial copy.
- Transfer the original unaltered copy of the documentation when you transfer the related IBM product (which may be either machines you own, or programs, if the program's license terms permit a transfer). You must, at the same time, destroy all other copies of the documentation.

You are responsible for payment of any taxes, including personal property taxes, resulting from this authorization.

THERE ARE NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING THE WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Some jurisdictions do not allow the exclusion of implied warranties, so the above exclusion may not apply to you.

Your failure to comply with the terms above terminates this authorization. Upon termination, you must destroy your machine readable documentation.

Trademarks

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

AIX
IBM
OS/2
Operating System/2

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

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

List of Abbreviations and Acronyms

AIX	Advanced Interactive Executive (operating system)
API	application program interface
BBRAM	battery-backed random access memory
CBC	cipher block chain
CCA	Common Cryptographic Architecture
CDMF	Commercial Data Masking Facility
CP/Q	Control Program/Q
DES	Data Encryption Standard
DMA	direct memory access
EPROM	erasable programmable read-only memory
FIFO	first-in-first-out
FIPS	Federal Information Processing Standard
IBM	International Business Machines
MB	megabyte
MAC	message authentication code
ODM	object data manager
OS/2	Operating System/2
PCI	peripheral component interconnect
PDF	portable document format
PKA	public key algorithm
POST	power-on self-test
PPD	program proprietary data
RAM	random access memory
RNG	random number generator
ROM	read-only memory
RSA	Rivest-Shamir-Adleman (algorithm)
SCC	secure cryptographic coprocessor
TOD	time-of-day (clock)

Glossary

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

- The *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42 Street, New York, New York 10036. Definitions are identified by the symbol (A) following the definition.
- The *Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) following the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) following the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

A

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

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

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

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

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

AIX operating system. Advanced Interactive Executive operating system.

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

ANSI. American National Standards Institute.

API. Application program interface.

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

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

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

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

B

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

BBRAM. Battery-backed random access memory.

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

C

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

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

CBC. Cipher block chain.

CCA. Common Cryptographic Architecture.

CDMF algorithm. Commercial Data Masking Facility algorithm.

ciphertext. (1) Data that has been altered by any cryptographic process.

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) See also *ciphertext*.

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

Comm_Mgr. Communications Manager.

Common Cryptographic Architecture (CCA). A comprehensive set of cryptographic services that furnishes a consistent approach to cryptography on major IBM computing platforms. Application programs can access these services through the CCA application program interface.

Common Cryptographic Architecture (CCA) API. The application program interface used to call Common Cryptographic Architecture functions; it is described in the *IBM 4758 CCA Basic Services Reference and Guide*, &frefform..

Communications Manager (Comm_Mgr). A CP/Q^{++} extension for the IBM 4758 PCI Cryptographic Coprocessor that manages communication among the host device driver, SCC applications, and CP/Q^{++}. It handles the receipt and delivery of request headers, and the inbound and outbound data buffers.

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

coprocessor. (1) A supplementary processor that performs operations in conjunction with another processor. (2) A microprocessor on an expansion card that extends the address range of the processor in the host system, or adds specialized instructions to handle a particular category of operations; for example, an I/O coprocessor, math coprocessor, or a network coprocessor.

CP/Q. Control Program/Q.

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

cryptographic node. A node that provides cryptographic services such as key generation and digital signature support.

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

D

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

Data Encryption Standard (DES). The National Institute of Standards and Technology (NIST) Data Encryption Standard, adopted by the U.S. government as Federal Information Processing Standard (FIPS) Publication 46, which allows only hardware implementation of the data encryption algorithm.

Data Encryption Standard Manager (DES_Mgr). A CP/Q^{++} extension that manages the IBM 4758 PCI Cryptographic Coprocessor DES processing hardware.

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

DES. Data Encryption Standard.

DES_Mgr. Data Encryption Standard Manager.

device driver. (1) A file that contains the code needed to use an attached device. (2) A program that enables a computer to communicate with a specific peripheral device; for example, a printer, videodisc player, or a CD drive.

direct memory access (DMA). The transfer of data between memory and input/output units without processor intervention.

DMA. Direct memory access.

E

encipher. (1) To scramble data or convert it to a secret code that masks its meaning. (2) Contrast with *decipher*.

enciphered data. (1) Data whose meaning is concealed from unauthorized users or observers. (2) See also *ciphertext.*

EPROM. Erasable programmable read-only memory.

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

expansion board. Synonym for *expansion card*.

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

expansion slot. One of several receptacles in a PC or RS/6000 machine into which a user can install an expansion card.

F

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

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

FIFO. First-in-first-out.

FIPS. Federal Information Processing Standard

first-in-first-out (FIFO). A queuing technique in which the next item to be retrieved is the item that has been in the queue for the longest time. (A)

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

H

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

host. As regards to the IBM 4758 PCI Cryptographic Coprocessor, the workstation into which the coprocessor is installed.

I

inline code. In a program, instructions that are executed sequentially without branching to routines, subroutines, or other programs.

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

intrusion latch. A software-monitored bit that can be triggered by an external switch connected to a jumper on the IBM 4758 PCI Cryptographic Coprocessor. This latch can be used, for example, to detect when the cover of the coprocessor host workstation has been opened. The intrusion latch does not trigger the destruction of data stored within the coprocessor.

J

jumper. A wire that joins two unconnected circuits.

K

key. In computer security, a sequence of symbols used with an algorithm to encipher or decipher data.

M

MAC. Message authentication code.

master key. In computer security, the top-level key in a hierarchy of KEKs.

message authentication code (MAC). In computer security, (1) a number or value derived by processing data with an authentication algorithm, (2) the cryptographic result of block cipher operations, on text or data, using the cipher block chain (CBC) mode of operation.

miniboot. Software within the IBM 4758 PCI Cryptographic Coprocessor designed to initialize the CP/Q^{++} operating system and to control updates to flash memory.

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

N

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

NIST. National Institute of Science and Technology.

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

NT. See *Windows NT*.

O

object data manager (ODM). In the AIX operating system, a data manager intended for the storage of system data.

ODM. Object data manager.

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

OS/2. Operating System/2.

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.

PKA. Public key algorithm.

PKA_Mgr. Public Key Algorithm Manager.

POST. Power-on self-test.

power-on self-test (POST). A series of diagnostic tests that runs automatically when device power is turned on.

PPD. Program proprietary data.

PPD_Mgr. Program Proprietary Data Manager.

private key. (1) In computer security, a key that is known only to the owner and used with a public key algorithm to decipher data. Data is enciphered using the related public key. (2) Contrast with *public key*. (3) See also *public key algorithm*.

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

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

Program Proprietary Data Manager (PPD_Mgr). A CP/Q^{++} extension for the IBM 4758 PCI Cryptographic Coprocessor that manages the persistent data associated with a particular SCC application. Persistent data is stored in flash memory or battery-backed RAM, and is protected from other SCC applications.

public key. (1) In computer security, a key that is widely known and used with a public key algorithm to encipher data. The enciphered data can be deciphered only with the related private key. (2) Contrast with *private key*. (3) See also *public key algorithm*.

public key algorithm (PKA). (1) In computer security, an asymmetric cryptographic process that uses a public key to encipher data and a related private key to decipher data. (2) See also *RSA algorithm*.

Public Key Algorithm Manager (PKA_Mgr). A CP/Q^{++} extension that manages the IBM 4758 PCI Cryptographic Coprocessor PKA processing hardware.

R

RAM. Random access memory.

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

random number generator (RNG). A system designed to output values that cannot be predicted. Since software-based systems generate predictable, pseudo-random values, the IBM 4758 PCI Cryptographic Coprocessor uses a hardware-based system to generate true random values for cryptographic use.

Random Number Generator Manager (RNG_Mgr). A CP/Q^{++} extension that manages the IBM 4758 PCI Cryptographic Coprocessor hardware-based random number generator.

read-only memory (ROM). Memory within which stored data cannot be modified routinely.

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

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

RNG. Random number generator.

RNG_Mgr. Random Number Generator Manager.

ROM. Read-only memory.

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

S

SCC. Secure cryptographic coprocessor.

SCC_Mgr. Secure Cryptographic Coprocessor Manager.

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

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

Secure Cryptographic Coprocessor Manager (SCC_Mgr). A CP/Q^{++} extension that provides high-level management of all agents running within a IBM 4758 PCI Cryptographic Coprocessor. As the “traffic cop”, the SCC_Mgr identifies agents and and controls the delivery of their messages and data.

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

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

T

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

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

TOD clock. Time-of-day clock.

U

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

V

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

vital product data (VPD). A structured description of a device or program that is recorded at the manufacturing site.

VPD. Vital product data.

W

Windows NT. A Microsoft operating system for personal computers.

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

IBM 4758. IBM 4758 PCI Cryptographic Coprocessor.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W