IBM 4767 PCIe Cryptographic Coprocessor Custom Software Developer's Toolkit Guide

Note: Before using this information and the products it supports, be sure to read the general information under “Notices” on page 93.

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

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

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

Contents

Figures

Index of Tables

About this document

The IBM 4767 PCIe Cryptographic Coprocessor Custom Software Developer's Toolkit Guide describes the Toolkit and its components, including the tools that enable developers to:

- build applications for the IBM 4767 PCIe Cryptographic Coprocessor,
- load applications under development into a coprocessor, and
- debug applications under development running within a coprocessor.

This document is intended for developers who are creating applications to use with the coprocessor. People who are interested in packaging, distribution, and security issues for custom software should also read this document.

Prerequisite knowledge

The reader of this document 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 coprocessor hardware, the Linux operating system that runs within the coprocessor hardware, and the use of the IBM's Common Cryptographic Architecture (CCA) application and support program are also helpful.

People who are interested in packaging, distribution, and security issues for custom software need to understand the use of the CCA Support Program and should be familiar with the coprocessor's security architecture. See “Related publications” in this section for details.

Organization of this document

This document is organized as follows:

Chapter 1, “Introduction” describes the documentation available to a developer of a coprocessor application, lists the prerequisites for development, describes the development process, and lists the tools used during development.

Chapter 2, “Installation and setup” describes how to install the Toolkit and how to prepare an IBM 4767 PCIe Cryptographic Coprocessor for use as a development platform.

Chapter 3, “Developing and debugging a coprocessor application” discusses in detail the use of each of the tools used during development of a coprocessor application.

Chapter 4, “Packaging and releasing a coprocessor application” describes how to prepare a coprocessor application to be distributed to end users.

Chapter 5, “Overview of the development process” lists the steps a developer needs to perform during development and testing of a coprocessor application.

Chapter 6, “Using CLU” briefly describes the use of the Coprocessor Load Utility (CLU).

Chapter 7, “How to reboot the coprocessor” describes several ways to reboot a cryptographic coprocessor. If an application has been loaded into the coprocessor, it starts to run after the reboot is complete.

Chapter 8, “Using Signer and Packager” describes the use of the signer and packager utilities and explains why the design of the coprocessor makes these utilities necessary.

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

Typographic conventions

This publication uses the following typographic conventions:

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

For readability in file and directory path specifications that apply to both Linux® and Windows®, this publication uses Linux naming conventions instead of showing both Linux and Windows conventions. For example, *cctk/<version>/samples* would be *cctk\<version>\samples* on Windows.

Syntax diagrams

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

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

A value for *firstarg* must be specified. A value for *secondarg* may be omitted. Either **a** or **b** must be specified.

<CLU> is used generically throughout this document to indicate either *csulclu* on Linux or *csufclu* on IBM AIX®, depending on the operating system for the machine in which the adapter is installed.

Related publications

Publications about IBM's family of cryptographic coprocessors are available at the IBM CryptoCards website.

Publications specific to the IBM 4767 PCIe Cryptographic Coprocessor and to CCA are available at the IBM PCIeCC2 library page. The *CCA Basic Services Reference* has a section titled "Related Publications" that describes cryptographic standards, research, and practices relevant to the coprocessor.

Summary of changes

This edition of the *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Developer's Toolkit Guide* contains product information that is current with the IBM 4767 PCIe Cryptographic Coprocessor announcements.

-

1 Introduction

The Toolkit is a set of libraries, include files, and utility programs that help a developer build, load, package, and debug applications written in C or assembler for the IBM 4767 PCIe Cryptographic Coprocessor. An application that runs within the coprocessor is known as an “agent” or a “coprocessor-side application”.

The following constitute a complete development environment for the IBM 4767 PCIe Cryptographic Coprocessor:

- The Toolkit.
- At least one machine that meets the requirements of the x86 server list running an operating system that is supported for the current release of the toolkit.
  - Card-side development must take place on a supported machine running one of the supported versions of Linux.
  - Host-side development typically occurs on the same Linux machine as the card-side development. For customers wishing to deploy on Windows or AIX, an additional machine for host development on the target deployment operating system is also required.
  - Customers typically develop, debug, and test entirely on Linux, and then port the host-side code to the target operating system once a stable configuration has been developed.
  - IBM recommends performing as much of the development activity as possible on Linux as there are significantly more resources available for diagnosing and debugging common problems as compared to other operating systems.
- A C compiler and linker that can cross-compile to the target PowerPC Linux. This is often referred to in the documentation as the “cross-compiler”. A script for building a PowerPC cross compiler is included in the cross_compiler_scripts directory of the toolkit.
- A C compiler and linker that can compile applications for the chosen host platform. Typically, this is gcc for Linux, xlc for AIX, and cl.exe for Windows. Please note not all Toolkit versions may contain AIX and Windows support. Contact your Toolkit provider for additional information.

IBM's CCA Support Program feature is required in order to create and debug toolkit applications. See the Download software page on the IBM CryptoCards website for details.

The CCA Support Program for Linux is required because it contains the host device driver, CCA node management tools, and other associated tools that are prerequisites for Toolkit development.

In order to comply with federal export regulations, customers need to specify their customer order number and adapter serial number when downloading the device driver and CCA installation package. In addition, this information is required when IBM processes a warranty replacement request for the adapter. Customers are strongly urged to maintain this information for all adapters purchased from IBM.

This chapter includes:

- A description of the documentation available to a developer of a coprocessor application and suggestions on the order in which the introductory material should be read.
- A list of hardware and software necessary to develop and release coprocessor applications.
- An overview of the development process.
- A description of the software that constitutes the development environment.

- A description of the software used to prepare a coprocessor application for release.

1.1 Available documentation

"Related publications" lists several publications, many of which are of particular interest to the developer of a coprocessor application.

Before and during development, the following manuals will be of use:

- This document, which describes the overall development process and the tools used in the development process.
- The *IBM 4767 PCIe Cryptographic Coprocessor CCA Support Program Installation Manual*, which describes how to install, load, and begin to use CCA. Throughout this document, this manual will be referred to as the *CCA Installation Manual*.
- The *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference*, which describes the function calls supplied by the coprocessor device drivers that manage communication, encryption and decryption, random number generation, nonvolatile memory, and other coprocessor services.
- Developers writing extensions for IBM's CCA application will also need the *IBM 4767 PCIe Cryptographic Coprocessor CCA User Defined Extensions Reference and Guide*.

1.2 Prerequisites

Prior to the start of development a developer must obtain and install the following:

1. An IBM 4767 PCIe Cryptographic Coprocessor. Refer to the How to order page for ordering information.

   The IBM 4767 PCIe should be installed in a host following the instructions in the *IBM 4767 Cryptographic Coprocessor Installation Manual*, which also lists the hardware and software requirements for the host. For application development, the host must be running a supported operating system on a machine that meets the requirements listed on the x86 server list.

   Note that the Linux host device driver requires the kernel development packages to be installed. This is because the device driver is compiled from source with dependencies on kernel headers as part of the installation package. Typically these packages are the kernel-source and kernel-devel packages matching the EXACT version of the kernel in use on the machine.

   Additionally, for Linux, note that kernel updates will require the host device driver to be rebuilt from source. This can be accomplished by following the instructions for compiling the device driver in the installation readme.txt file. If you have questions, contact your Toolkit provider.

2. A cross-compiler for Linux running on the PowerPC chip and the associated tools, which should be installed following the instructions provided with the compiler. Only the compiler and linker need be installed; other components (visual build environments and so on) are not required. The cross-compiler should be built to target the version of GLIBC used by the Linux OS on the adapter. Instructions for obtaining the cross-compiler should be obtained directly from your IBM Toolkit provider. Also, a sample script for building the cross-compiler is in the Toolkit in *cctk/<version>/cross_compiler_scripts*. Please read the prologue of *build_4767_cross_compiler.sh* in that directory for details about building the cross-compiler. Note that the cross-compiler will require a couple of GB of storage, and may take an hour or more to build. Additionally, compiler development tools such as flex, yacc, bison, makeinfo and so forth may need to be installed with the appropriate

package management tools as they are typically not part of a standard Linux installation.

3. Developers also need a copy of the IBM CCA Installer, which is available from the Download software page on the IBM CryptoCards website. This installer will include the device driver for the coprocessor as well as certain utilities (such as CLU) needed for development.

4. A compiler for code running on the host system, such as gcc for Linux, the Microsoft C/C++ Optimizing Compiler for Windows (cl.exe), or xlc for AIX. For application development, the host must be running a supported operating system on a machine that meets the requirements listed on the x86 server list.

   Only the compiler and linker need to be installed; other components (visual build environments and so on) are not required.

5. The IBM 4767 PCIe Application Program Development Toolkit (the Toolkit), also referred to as the cctk, available from IBM, which should be installed on the same host as the compiler following the instructions in chapter 2 of this manual. The Developer's Toolkit includes Interactive Code Analysis Tool (ICAT) for the IBM 4767 (ICATPZX).

6. A copy of MKFS for JFFS2 to build an image that can be downloaded onto the coprocessor for development and production. Depending on the Linux installation, MKFS for JFFS2 may not be installed. For information, visit http://www.sourceware.org/jffs2/ or use the *download_and_build_mkfs_jffs2.sh* script located in the *build_seg3_image* directory. As of this publication date, version 1.50 or later can be used. MKFS for JFFS2 is a Linux utility and should be invoked from a Linux system. This means all JFFS2 images to be downloaded into the coprocessor must be built from a Linux host.

Note: Refer to the Custom Programming page of the IBM CryptoCards website for more information about the toolkits. To contact IBM concerning availability of a toolkit, or to verify whether a desired host operating system is available for a given toolkit release, refer to the Contact page or contact your Toolkit provider.

1.3 Development overview

The host-side piece of a coprocessor application packages application specific request data into structures that can be processed by the host device driver software and kernel module. The data in these structures is then routed to the appropriate card-side coprocessor application through host device driver interactions with the coprocessor side communications manager running inside the adapter.

The host-side piece of a coprocessor application is compiled and linked using a supported compiler on the host PC.

The coprocessor-side piece of a coprocessor application receives the request data from the host side piece of a coprocessor application, unpackages and verifies the request data, processes the request data and then packages up and returns a response to the card side communications manager which then returns that reply to the host device driver which then routes the reply back to the host side coprocessor application. The coprocessor side piece of a coprocessor application is compiled on the host using a cross-compiler that targets the card's PowerPC architecture.

As illustrated in Figure 1, the coprocessor-side piece of a coprocessor application is compiled and linked using include and library files customized for the coprocessor environment. The executable and other application-related files are then packaged into a JFFS2 image which can be understood by the coprocessor and is downloaded to the coprocessor.

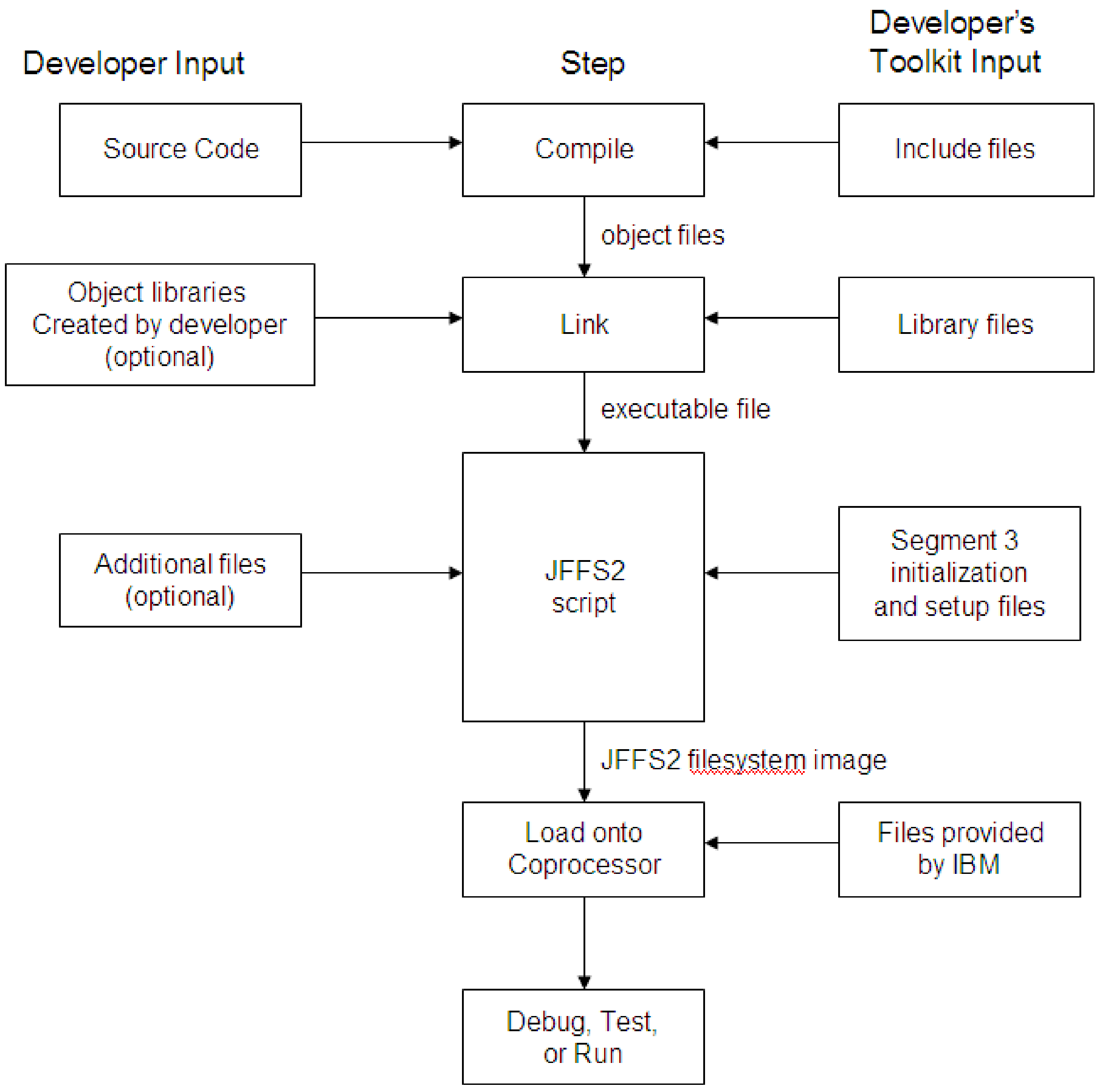


Figure 1 Development process overview

The following steps are required to build and load coprocessor applications for development:

1. Write the host-side and coprocessor-side pieces of a coprocessor application in C, using the Toolkit headers as necessary.

   Note: The sample applications in *cctk/<version>/samples* provide many examples of how to write, compile, and link both pieces of a coprocessor application.

- Customers writing Toolkit applications may access the host-side coprocessor application routines through the use of a Java Native Interface layer as demonstrated in cctk/<version>/samples/toolkit/java.
- Customers writing custom UDXes can write a similar JNI for their UDX verb host side entry points.

2. Compile the host-side code using one of the supported native compilers.
3. Link the host-side code using one of the supported native linkers.
4. Compile the coprocessor-side code using a supported cross-compiler.
5. Link the coprocessor-side code using the linker shipped with the cross-compiler.
6. Use **`mkfs.jffs2`**, along with the makefile and other files provided in *cctk/<version>/build_seg3_image* to create a JFFS2 image.
7. Load the JFFS2 filesystem image into the coprocessor using DRUID, which is included with the Developer's Toolkit.

1.4 Development components

The development environment software consists of the following items, most of which are contained in the Toolkit:

Compiler and linker

Use the appropriate compiler (gcc or cl) and linker for the host-side code. These are not shipped with the Toolkit. On Linux, Yum, YaST, zypper, your operating system's package management software, can be used to install the compiler for the host code. On Windows, the compiler will typically be a part of an MSDN subscription. On AIX, xlc can be downloaded from the Passport Advantage and Entitled Software Support websites or contact your AIX support provider.

Use a gcc-based cross-compiler for the coprocessor-side code. A sample script for building the cross-compiler is in the Toolkit in *cctk/<version>/cross_compiler_scripts*. Please read the prologue of *build_4767_cross_compiler.sh* in for details about building the cross-compiler.

Note: Since the cross-compiler is gcc-based, the only scripts provided with the toolkit to build a cross-compiler are intended to be run on Linux. IBM does not officially support a cross compiler for the coprocessor on Windows or AIX. Note that building a cross-compiler will require root-level authority (to install in a system-wide accessible area) and may require a couple of GB of storage as well as an hour or more to compile and install.

Libraries and include files

IBM-specific card and host-side libraries necessary for development and deployment of Toolkit applications can be found in cctk/<version>/lib/card and cctk/<version>/lib/host/.

When developing an application to be run on the IBM 4767, use the Toolkit include files in addition to the include files associated with the compiler and assembler. These files furnish the prototypes that coprocessor-side code uses to interface with the cryptographic extensions. Developers should structure the order of the toolkit includes in the same manner as in the toolkit examples.

Utilities

Use the following utilities to prepare and load the coprocessor-side piece of a coprocessor application:

- **mkfs.jffs2:** Use the appropriate JFFS2 image creation utility mkfs.jffs2 to create a JFFS2 image to load onto the coprocessor using DRUID. See "Building JFFS2 filesystem Images" on page 29 for a sample invocation of the JFFS2 utility to create an image.

On the 4767, mkfs.jffs2 is used to create a directory structure which will reside in /flashS3 when loaded onto the coprocessor using DRUID or CLU. Additionally, based on the directives of init.sh, files may be copied into /lib, /bbram, and /ramS3.

Warning

It is not possible to use a mkfs.jffs2 image created for the 4765 on a 4767 coprocessor.

mkfs.jffs2 is not shipped with the toolkit. It can be downloaded from http://www.sourceware.org/jffs2/ or built using the *download_and_build_mkfs_jffs2.sh* script in the *build_seg3_image* directory of the Toolkit.

Consult the man page for the installed JFFS2 image creation utility for more details.

- **Device Reload Utility for Insecure Development (DRUID):** Use DRUID to load an application into a coprocessor configured as a development platform. This program can be found in the *bin/host/<platform>* subdirectory of the toolkit. DRUID is intended for development and debugging use only. DRUID should never be used to load production images onto a coprocessor which will then be used in a production environment. Doing so would mean that your production environment uses IBM's publicly known test keys and owner ID' instead of your organization's privately held signing keys and owner ID.

- **Coprocessor Load Utility (CLU):** CLU verifies and loads digitally signed system software and coprocessor commands into a coprocessor. This program is provided with the IBM CCA application and has different names depending on the host operating system (csulclu on Linux, csunclu.exe on Windows, and csufclu on AIX). On Linux, CLU will typically be located in */opt/ibm/4767/clu*. For convenience, a copy of CLU is also included in cctk/<version>/bin/host/linux. On Windows, CLU typically will be located in "C:\Program Files\IBM\4767\clu".

  Be careful not to confuse CLU for the IBM 4764 or 4765 with CLU for the IBM 4767. The commands are the same and the file names are similar.

- **init.sh:** The toolkit includes various versions of an initialization shell script (**init.sh*) in the shells subdirectory. There are scripts for different configurations, including debug and non-debug versions of both regular coprocessor applications and CCA UDXes. An appropriate script must be incorporated in the JFFS2 image loaded onto the coprocessor. Once embedded in the JFFS2 image, the name must be init.sh and the file must have a full filename of /flashS3/init.sh. Customers editing this file should take special precautions not to insert Windows-style line feeds, which look like ^M in Linux. The Linux shell interpreter on the coprocessor cannot interpret these line feeds. When editing the initialization script to be built into the JFFS2 image, ensure that the file is saved with UNIX style line feeds. The easiest way to ensure this is to run the dos2unix command or equivalent on the file after editing or to use vim's ":set list!" and visually search for ^M characters.

  The initialization script must set any environment variables that the coprocessor-side piece of your application needs to establish, as well as environment variables that the debugger daemon [*zdaemon*] needs, if applicable. The script must then launch the coprocessor-side piece of your 4767 application, first launching *zdaemon*, if applicable. The various scripts available have been constructed so that most customers will not need to alter them, and they will be renamed appropriately by the build process outlined by the segment-3 image creation makefile.

- **Segment-3 Image Creation Makefile:** The *cctk/<version>/build_seg3_image* subdirectory includes a makefile for automated creation of a JFFS2 image to be loaded onto the coprocessor using DRUID. This makefile also creates helper scripts that export environment variables used when debugging with ICAT. Developers should pay special attention to the various sanity checks and restrictions imposed by this makefile and associated include files and exercise caution when making modifications, as changes may result in difficult to diagnose errors.

- (Linux) **Parseclu:** This utility can be used to parse a CLU file into a human-readable (XML) format, which may prove useful during the signing process to verify the contents of signed images. The parseclu utility is described in Parsing a CLU file on page 33.

CLU input files

The Toolkit includes several files used as input to CLU during the development process. These include files to load the development environment onto the coprocessor and to remove it from the coprocessor. For a full listing of CLU files included with the Toolkit, please refer to CLU files directory on page 11.

Debugger

The Toolkit includes the IBM Interactive Code Analysis Tool (ICAT), a program (icatpzx on Linux) that controls and debugs the coprocessor-side piece of a coprocessor application. For more information on ICAT, refer to the *IBM 4767 PCIe Cryptographic Coprocessor ICAT Debugger Getting Started* document.

ICAT also includes zdaemon, the coprocessor-side debugger daemon which must be built into the JFFS2 image loaded onto the coprocessor and launched by init.sh. This daemon enables ICAT to communicate with the coprocessor.

Host-side applications can be debugged with the native debugger associated with the compiler / tool set used to build the host-side coprocessor application. For example, gdb or dd can be used on Linux.

Coprocessor operating system

The Toolkit includes two copies of the embedded Linux operating system residing in segment 2 of the coprocessor, one signed with keys and owner identifiers corresponding to a production environment and the other signed with keys and owner identifiers corresponding to a development environment. The development and production versions of the embedded Linux operating system are identical, and can be loaded into the coprocessor by CLU. The copy of the operating system corresponding to the development environment is intended to be used in conjunction with DRUID for more rapid development. For a description of the coprocessor segments, see Packaging and releasing a coprocessor application on page 35.

1.5 Release components

The software required to prepare an application for release to end users, most of which is contained in the Toolkit, is as follows.

Note: For the 4767, all host-side components are 64-bit, and all card-side components are 32-bit (because the coprocessor itself uses 32-bit PowerPC processors).

Utilities

Use the following utilities to prepare the coprocessor-side piece of a coprocessor application for release. These utilities are provided on Linux:

- Signer, a program (CRUSIGNR) that
    - uses CCA to generate ECC keypairs and performs other cryptographic operations,
    - incorporates a JFFS2 image into a coprocessor command, and
    - uses the developer's private key to digitally sign the command (using CCA) so the developer's customers can download the JFFS2 image to a coprocessor.

- Packager, a program (CRUPKGR) that combines one or more signed commands (outputs from Signer) into a single file for download to the coprocessor. Packager is included as a convenience utility to allow developers to simplify the CLU loading process for production machines. It provides no additional security functionality. It is not required to package a file generated by Signer before loading it onto the coprocessor using CLU. Packager should only be used to package individual signed outputs from Signer. You cannot use Packager recursively to package already packaged outputs.

- Load and unload scripts (ibm4767_load / ibm4767_unload) that properly load and unload the device driver for the coprocessor.

  Refer to How to change the host device driver timeout on page 55 for details on how to set the default host device driver timeout.

CCA application and support program

Signer and Packager use IBM's Common Cryptographic Architecture (CCA) application to generate keypairs and the digital signatures required for signing and packaging. See “Prerequisites” on page 2 for more information. The CCA application and support program can be downloaded from the Download software page. In order to download the software, users will need to know their IBM customer number and adapter tracking serial number. For questions about the CCA application and support program download and installation, please contact IBM Crypto Support at crypto@us.ibm.com.

2 Installation and setup

The Toolkit includes utilities used to build a coprocessor application and prepare the coprocessor-side piece it to be loaded into a coprocessor. This chapter describes how to install the Toolkit, discusses the toolkit's directory structure and contents, and lists many of the files used during development.

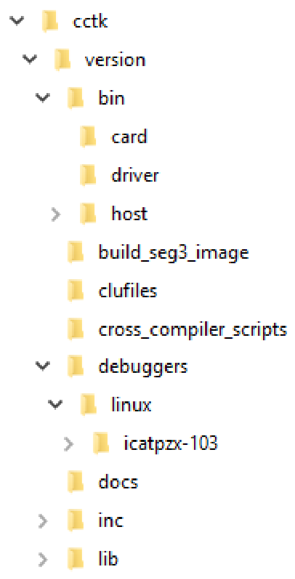
2.1 Installing the toolkit

The Toolkit is shipped as a compressed tarball and a corresponding md5sum file. To install it:

- Verify the md5sum of the compressed tarball against the shipped md5sum file. On Linux, this is done by using the *md5sum –c* command. Developers can use this or any other MD5 utility to verify the zipped tarball before using it. If the md5sums do not verify, contact your Toolkit provider before proceeding or try to verify the contents of the failing tarball against what was officially provided by IBM.
- Linux: Extract the Toolkit using xzcat and tar through the following invocation:
  ```
  xzcat cctk-<version>-<date>.xz | tar -xv
  ```
- Windows: Extract the Toolkit using a utility such as winzip or pkzip.
- Typically, users will store a master copy of the Toolkit tarball on their machine in a globally readable, but not writable location, then extract a copy of the contents to their home directory for development. The toolkit samples and development procedures write output to directories within the toolkit, and as such, developers must have write authority for the entire toolkit. For convenience, a master file listing hashes of all files as originally shipped by IBM for this release is provided with the toolkit files.md5sum file located in cctk\<version>\.

2.1.1 Directories and files

The Toolkit is contained in the directory structure similar to that as depicted in Figure 2. The “version” directory specifies the toolkit version, and will be denoted <*version*> in the following discussion.



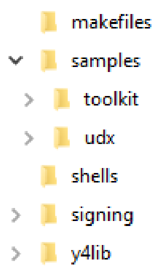


Figure 2 Toolkit directory structure

Note: The makefiles and y4lib directories are now in the Toolkit *version* directory, which is pointed to by *CCTK_FS_ROOT*. This makes it easier for customers to write their own Toolkit code without having to place their code in the *samples* directory.

2.1.1.1 card directory

The *cctk/<version>/bin/card* directory contains various daemons that a developer may wish to include in the JFFS2 image downloaded to the coprocessor and a copy of the CCA executable which is necessary when developing and deploying UDXes.

- *csulccaW* is the workstation version of the CCA card-side executable. Customers writing standalone toolkit applications should not include csulccaW in their JFFS2 image.

  Note: When debugging a UDX, this application will be renamed to csulccaA on the card due to the nature of the CDU process. To attach to your UDX, enter csulccaA as the application name.

- *cryptologkd* allows kernel messages to be passed back to the host device driver, which on Linux emits them to */var/log/messages*. This utility must be started as part of the segment-3 initialization and startup process for all applications. Failure to start cryptologkd may result in segmentation faults or kernel panics. This occurs because even if your application does not directly log messages back to the host, the communications manager and other on-card services may do so. Should these other services try to log a message and be unable to do so, an error will be thrown which will be escalated by the signal handler on the card to prevent further program execution.

- *startcdud* starts the Concurrent Data Update (CDU) daemon, which enables an update to a properly written application to be downloaded while an earlier version continues to run, and seamlessly starts the newer version while terminating the older one.

  Note: Only regular CCA applications are CDU-able. UDX applications can use CDU safely to start CCA, but if an update is loaded onto the coprocessor that require a new UDX library to be moved into /lib, CDU CANNOT do this. In this case, the device driver must be unloaded and reloaded so the entire init.sh file can be executed. This will copy the library to */lib*. Toolkit-only (non-UDX) applications do not need to use *startcdud*.

2.1.1.2 cctk/<version>/bin/host/<platform> directory

The *cctk/<version>/bin/host/<platform>* directory contains the following:

- The Device Reload Utility for Insecure Development (DRUID), used to load JFFS2 images onto a coprocessor that has been prepared for use as a development platform. DRUID is only capable of loading images onto adapters with the Toolkit Development load of segments 2 and 3 (segment 2's owner ID must be 3, and segment 3's owner ID must be 6). DRUID works by combining the REMBURN3 functionality of SIGNER with the PL functionality of CLU to sign and load JFFS2 images.

- (Linux) The signer utility (CRUSIGNR), used when preparing the production version of a coprocessor application for release. See "The Signer utility" on page 74 for details.
- (Linux) The packager utility (CRUPKGR), also used when preparing the production version of a coprocessor application for release. See "The Packager utility " on page 88 for details.
- (Linux) The parseclu utility, which can be used to dump the contents of any CLU file (packaged or otherwise) into XML format. This may prove useful during the signing process to verify the contents of signed images. The parseclu utility is described in Parsing a CLU file on page 33.

2.1.1.3 Host Device Driver Scripts Directory

(Linux) The *cctk/<version>/bin/driver* directory contains scripts (ibm4767_unload / ibm4767_load) used to load and unload the device driver. These scripts must be run as root or via sudo.

ibm4767_load loads the device driver:

```
sudo ibm4767_load
```

Note: the procedure for using *ibm4767_load* to set the default host device driver timeout has changed from the IBM 4765. See How to change the host device driver timeout on page 55 for details.

A user may want to add *cctk/<version>/bin/host/* and *cctk/<version>/bin/driver* to the PATH environment variable so that these scripts and utilities such as DRUID are easily accessible.

2.1.1.4 Image directory

The *cctk/<version>/build_seg3_image* directory contains a makefile used to generate a JFFS2 image to be loaded onto a coprocessor. The directory also serves as a root in which the JFFS2 /flashS3 filesystem image is built.

Note: The base directory build is */flashS3*. Once loaded into the coprocessor, and before execution, files in */flashS3* are copied to */ramS3*. See the *init.sh* file for more details.

2.1.1.5 CLU files directory

The *cctk/<version>/clufiles* directory contains files to be used as input to CLU, including those listed below. See "Using CLU" on page 53 for a complete description of how to use CLU.

Note: Compared to older generations of the toolkit, customers may notice that certain toolkit files are no longer packaged together to form a combined CLU file containing more than one ESTOWN / EMBURN / REMBURN miniboot command file. Instead, each ESTOWN, EMBURN, REMBURN, and SUROWN command resides individually in its own file and performs a single Miniboot command. The naming convention has been simplified so that all official Toolkit CLU files end with the .clu extension. Developers can use Packager to package these CLU files together to form a load sequence if desired.

The CLU file naming convention is *functionname_r.j.m.clu*, where *r* signifies release, *j* signifies major revision, and *m* signifies minor revision. In this section, file names are color-coded to denote production files and development files:

Green	Production
Red	Development

Warning

> Under no circumstances should a production environment use a CLU file with ownerID 3, nor should a development environment use a CLU file with ownerID 243.

- *reload_seg1_xipz_factory_to_prod_keyswap_r.j.m.clu (Production),* which loads IBM's system software into a coprocessor [1]. This file can only be loaded into a coprocessor in the factory-fresh state [2]. This file is included in the toolkit as a convenience for developers and is also shipped as part of the CCA installation package.

- *reload_seg1_xipz_r.j.m.clu (Production)*, which updates the system software in a coprocessor. This file loads IBM's system software into a coprocessor into which system software has previously been loaded. This file is included in the toolkit as a convenience for developers and is also shipped as part of the CCA installation package.

Warning

> *reload_seg1_xipz_r.j.m.clu* updates the public key associated with segment 1. This key can only be updated a limited number of times before the coprocessor runs out of memory in which to store the certificate chain connecting the segment 1 public key to the original key installed at the factory. Users should update the system software in a coprocessor as seldom as possible. Note that *reload_seg1_xipz_r.j.m.clu* need be loaded only once per adapter and that segment 1 revision does not necessarily need to match the segment 2/segment 3 revision for the coprocessor to function properly. Segments 2 and 3 can be updated as many times as desired. Please contact your Toolkit provider if you have any questions about loading segment 1.

[1] This action also sets the public key associated with segment 1.
[2] In particular, the public key associated with segment 1 must be the key installed during manufacture.

- *establish_ownership_seg2_xipz_OID2_r.j.m.clu (Production)*, which establishes ownership of segment 2 as ownerID 2, which is associated with IBM. This file is shipped with the CCA installation package. For development purposes, it is needed by IBM Z UDXes only, but is included with all Toolkits as a convenience.

- *establish_ownership_seg2_toolkit_OID243_r.j.m.clu (Production)*, which prepares a coprocessor for use with segment 2 ownerID 243. The version of this file with ownerID 243 can be incorporated in CLU files a developer prepares for release to customers.

- *establish_ownership_seg2_toolkit_OID3_r.j.m.clu (Development)*, which prepares a coprocessor for use with segment 2 ownerID 3 [3]. The version of this file with ownerID 3 prepares the coprocessor for use during development.

  Warning

Under no circumstances should a production environment use a CLU file with ownerID 3, nor should a development environment use a CLU file with ownerID 243.

  Either *establish_ownership_seg2_toolkit_OIDnnn_r.j.m.clu* file (OID243 or OID3) can be loaded into a coprocessor only after *reload_seg1_xipz_ecc_r.j.m.clu* has been loaded, and once loaded cannot be reloaded until the corresponding *surrender_ownership_seg2_toolkit_OIDnnn_r.j.m.clu* file has been loaded [4].

- *emergency_reload_seg2_xipz_OID2_r.j.m.clu (Production)*, which loads the embedded OS into the coprocessor segment 2 for ownerID 2, which is associated with IBM. This file is shipped with the CCA installation package. For development purposes, it is needed by IBM Z UDXes only, but is included in all Toolkits as a convenience.

- *emergency_reload_seg2_toolkit_OID243_r.j.m.clu*, which loads the embedded OS into the coprocessor for release to customers.

- *emergency_reload_seg2_toolkit_OID3_r.j.m.clu (Development)*, which loads the embedded OS into the coprocessor for use during development.

  This file can only be loaded into a coprocessor after the corresponding *establish_ownership_seg2_toolkit_OIDnnn_r.j.m.clu* file (OID243 or OID3) has been loaded. Loading either of these files clears BBRAM.

[3] This action sets the public key and owner identifier (OID) associated with segment 2. The toolkit development OID assigned to segment 2 is 3.

[4] In particular, segment 2 must be empty and the public key associated with segment 1 must be the key loaded by *reload_seg1_xipz_factory_to_prod_keyswap_r.j.m.clu* or *reload_seg1_xipz_r.j.m.clu*. Loading CCA also causes the key associated with segment 1 to be set to the proper value.

Warning

Under no circumstances should a production environment use a CLU file with ownerID 3, nor should a development environment use a CLU file with ownerID 243.

- *reload_seg2_toolkit_OID243_r.j.m.clu (Production)*, which reloads the embedded OS into the coprocessor.

  This file can only be loaded into a coprocessor after the corresponding *emergency_reload_seg2_toolkit_OID243_r.j.m.clu* file has been loaded. Loading this file does not clear BBRAM.

- *reload_seg2_toolkit_OID3_r.j.m.clu (Development)*, which reloads the embedded OS into the coprocessor.

  This file can only be loaded into a coprocessor after the corresponding *emergency_reload_seg2_toolkit_OID3_r.j.m.clu* file has been loaded. Loading this file does not clear BBRAM.

- *establish_ownership_seg3_toolkit_OID6_r.j.m.clu (Development)*, which prepares a coprocessor for use in development.

  This file can only be loaded into a coprocessor after the *emergency_reload_seg2_toolkit_OID3_r.j.m.clu* file has been loaded, and once loaded cannot be reloaded until the *surrender_ownership_seg2_toolkit_OID3_r.j.m.clu* file or the *surrender_ownership_seg3_toolkit_OID6_r.j.m.clu* has been loaded.

- *emergency_reload_seg3_toolkit_OID6_r.j.m.clu (Development)*, which loads the initial development environment JFFS2 image onto the coprocessor (a prerequisite for use of the coprocessor in development).

  This file can only be loaded into a coprocessor after the *establish_ownership_seg3_toolkit_OID6_r.j.m.clu* file has been loaded. Loading this file clears BBRAM.

- *reload_seg3_toolkit_OID6_r.j.m.clu (Development)*, which reloads the initial development environment JFFS2 image onto the coprocessor.

  This file can only be loaded into a coprocessor after the *emergency_reload_seg3_toolkit_OID6_r.j.m clu* file has been loaded. Loading this file does not clear BBRAM.

- *surrender_ownership_seg2_toolkit_OID243_r.j.m.clu (Production)*, which removes the current JFFS2 image and the embedded OS from the coprocessor, clears BBRAM, and relinquishes ownership of segment 2 (and segment 3 if it is currently owned). This file can be incorporated in CLU files a developer prepares for release to customers.

- *surrender_ownership_seg2_toolkit_OID3_r.j.m.clu (Development)*, which removes the current JFFS2 image and the embedded OS from the coprocessor, clears BBRAM, and relinquishes ownership of segment 2 (and segment 3 if it is currently owned). This file prepares the coprocessor for use during development.

  Loading either of these files (OID243 or OID3) places the coprocessor in the same state as immediately after the *reload_seg1_xipz_r.j.m.clu* has been loaded for the first time.

  Warning

  If the adapter has any stored state information, either from CCA, if loaded, or your application, if loaded, that information will be lost permanently after any surrender ownership command. For CCA, examples include master keys and other Security Relevant Data Items (SRDIs), such as roles and profiles. For customer applications, this includes whatever state information your application accumulates and saves.

  Each of these files (OID243 or OID3) can be loaded into a coprocessor only after the corresponding *emergency_reload_seg2_toolkit_OIDnnn_r.j.m.clu* file has been loaded. The ownerID nnn must correspond to the currently loaded segment 2 ownerID to use this file.

  Warning

  Under no circumstances should a production environment use a CLU file with ownerID 3, nor should a development environment use a CLU file with ownerID 243.

- *surrender_ownership_seg2_xipz_r.j.m.clu (Production)*, which removes the current JFFS2 image and the embedded OS from the coprocessor, clears BBRAM, and relinquishes ownership of segment 2 (and segment 3 if it is currently owned). This places the coprocessor in the same state as immediately after the *reload_seg1_xipz_r.j.m.clu* has been loaded for the first time. This file is shipped with the CCA installation package and is included with the Toolkit as a convenience.

  This file can only be loaded into a coprocessor that has IBM's CCA application loaded (which corresponds to segment 2 ownerID = 2).

- *surrender_ownership_seg3_toolkit_OID6.clu (Development)*, which removes the current JFFS2 image from the coprocessor, clears BBRAM, and relinquishes ownership of segment 3 if the development platform (segment 2 ownerID = 3) is loaded. This places the coprocessor in the same state as immediately before the *establish_ownership_seg3_toolkit_OID6_r.j.m.clu* file is loaded.

  This file can only be loaded into a coprocessor after the *emergency_reload_seg3_toolkit_OID6_r.j.m.clu* file has been loaded.

  After the developer completes the setup of the coprocessor and installation of the toolkit, a CLU ST command should be run to ensure that the coprocessor is in the expected state. See “Determining coprocessor status” on page 40 for details.

Note: The “emergency” part of the names of several of the CLU file names refers to the state of the coprocessor. From the coprocessor’s point of view, a segment that is owned but unreliable is in an incomplete state that requires the assistance of the next lower segment before the coprocessor can be used as intended. For example, if the state of segment 3 is owned but unreliable, segment 2’s permission is required to proceed. Any time a segment owner must involve another owner to proceed, the word “emergency” is used as the terminology to describe that this is not a normal segment reload.

2.1.1.6 Cross-compiler directory

The *cctk/<version>/cross-compiler* directory contains a sample build script and associated files that show how to download and build the cross-compiler. The cross-compiler must be built on a (supported) Linux operating system. The prologue of the script contains additional information about building the cross-compiler.

2.1.1.7 debuggers directory

The *cctk/<version>/debuggers* directory contains the ICAT debugger, which runs on the host development platform and debugs the coprocessor-side piece of a coprocessor application remotely. At present, the Toolkit includes only the Linux version of the debugger in *cctk/<version>/debuggers/linux*/*icatpzx-nnn,* where *nnn* is the version of the debugger.

See the *IBM 4767 PCIe Cryptographic Coprocessor ICAT Debugger Getting Started* document, located in the *cctk/<version>/debuggers/linux/icatpzx-nnn/docs* directory, for information on how to use ICAT.

2.1.1.8 docs directory

The *cctk/<version>/docs* directory contains instructions on how a developer can generate and obtain certificates for the public keys that are required to release a production version of a coprocessor application. It also contains this document, the *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference*, and the *IBM 4767 PCIe Cryptographic Coprocessor CCA User Defined Extensions Reference and Guide*.

2.1.1.9 inc directory

The *cctk/<version>/inc* directories contain include files (.h) that are required to build coprocessor applications. Note: For the IBM 4767, UDX and Toolkit headers have been merged into this directory and its subdirectories. The proper INCLUDE path specifications are documented in the Toolkit makefiles. It is recommended that your build system include search path order matches the sample makefiles.

Customers writing standalone toolkit applications may also wish to incorporate headers from cctk/<version>/samples/y4lib/inc if they wish to use the y4lib shim layer code which shields toolkit customers from some of the impacts of the removal of the coprocessor-side pthreads library functionality. UDX developers should NOT include these headers. Additionally, these headers should NOT be merged into the inc directory because by design, they must be included separately from standard toolkit headers in order for the compiler to pick up the proper defines in the correct order.

2.1.1.10 lib directory

The *cctk/<version>/lib* directories contain shared object and other library files that are required to build host and card side coprocessor applications. All host-side libraries are 64-bit. All card-side libraries are 32-bit only, as this is the architecture of the adapter.

- Libraries that the host-side piece of an application may link against are in *cctk/<version>/lib/host/<platform>/debug*.

  The libraries in the debug directory should be used whenever the developer wants to trace the contents of the Common Processing Request Blocks (CPRBs) going to the coprocessor when running a UDX. This can be helpful when debugging if it appears that the data passed from the host-side piece of the UDX is not reaching the coprocessor in an appropriate fashion.

  To turn on this feature, set the CSUDUMP environment variable to YES (export CSUDUMP=YES) before running the UDX. The CPRB contents will be printed out to a file named SSDEBUG.LOG. Note: This file can become large very quickly, and these libraries should be used only during development and testing if necessary. The majority of developers do not need to use these files, but they are included as an additional convenience in case of difficulty.

- Libraries that the coprocessor-side piece links against are in *cctk/<version>/lib/card/.* Existing customers may notice that stub libraries are no longer required as these libraries are now built as part of the segment-3 image instead of residing in segment-2. Currently the following libraries are shipped in the lib/card directory:
  - libxccmn.so (and associated symbolic links) – Common cryptographic API routines used by coprocessor side applications to perform cryptographic tasks such as ECC/RSA key generate and others.
  - libxcoa.so (and associated symbolic links) – Contains the Outbound Authentication (OA) library routines.

2.1.1.11 samples directories

The *cctk/<version>/samples* directories contain the source for several sample coprocessor applications (both host-side and coprocessor-side pieces).

- *cctk/<version>/makefiles* contains common files used when building both pieces of each sample. The files in the makefiles directory contain examples of how to set the appropriate compiler options for building a toolkit or UDX sample. Together with the makefiles in each toolkit sample, these files can be used as a starting point for a customer toolkit or UDX application.

- For the 4767, Toolkit samples have been expanded to use the y4lib layer which is a convenience layer provided to assist existing Toolkit customers. Please note that the handling of fibers is performed by the y4lib code. Should a pure fibers example be required, IBM can provide one upon request to crypto@us.ibm.com.

- Customers writing first time applications for the 4767 are encouraged to use the y4lib layer. IBM expects customers to be more familiar with blocking API calls instead of the asynchronous fiber protocol.

- Additional documentation has been added to all toolkit samples in an attempt to clarify what customers should and should not change as well as required initial setup for all card side coprocessor applications should they decide to use one of these samples as an initial skeleton for their application.

- *cctk/<version>/samples/toolkit* contains samples for base (non-UDX) applications.

- *cctk/<version>/samples/udx* contains samples for UDX applications.

Base toolkit samples

The toolkit sample directory contains the sample programs shown in Table 1.

Table 1 Toolkit sample programs

Sample name	Description / purpose
count	Counts the number of installed adapters.
diagnostics	Allows a user to run commands on the adapter and have the output displayed to the host. This sample is useful to better understand the contents of the Linux segment-2 image as well as your custom segment-3 image. Example usage scenarios include: • Exploring the linux filesystem on the card through various ls statements (ls /etc ls /lib ls /proc, etc.) • Determining the available space left on the device through the df command • Determining a list of running applications on the adapter through ps commands Verifying the contents of your segment-3 image through ls or cat commands (for example, “cat /flashS3/init.sh”)
drbg	Exercises the Deterministic Random Bit Generator (DRBG) random number generator (RNG) functions.
getvpdata	Queries coprocessor Vital Product Data, which includes information such as the part number and serial number for the 4767 adapter(s) in your system.
java	(Linux) Demonstrates how to use the RTE sample with the Java JNI layer. This allows customers to communicate with the cryptographic coprocessor from Java applications. UDX customers can create their own JNI layer should they wish to call their UDX(es) from Java applications.
logging	Demonstrates how to log messages from the coprocessor into */var/log/messages* on the host machine. Customers can include these types of log messages in their applications for both debug and production environments which can provide useful information as to the state of your application. **Warning** All data you log to the host machine is visible on the host. Do not log sensitive data, such as any private keys or data you may have stored inside the adapter.
oa	(Linux) Uses Outbound Authentication to display OA certificate information which can be used to validate the certificate chain created when loading coprocessor segments.
oem	Demonstrates iterative host to coprocessor communication.
rte	“Reverse then echo” text is passed from the host to the coprocessor. This sample is an excellent starting point that can be used to verify that a developer can correctly build and load a sample application onto the coprocessor.
skeleton	“Multithreaded” (multi-fiber) program that demonstrates various API functionality, including: • DES Encryption, Decryption, and MAC functions • SHA Hash functions • Large Integer Math functions • PKA functions such as key generate, sign, and verify for RSA, ECC, and

	DSA keys • Deterministic Random Number Generator functions • Reverse Then Echo (RTE) text from host • AES Encryption, Decryption, and MAC functions • ECC key generation, signature creation and verification. This sample provides many working examples of how to call various coprocessor-side API functions.
stat	Queries coprocessor status information including: • vital product data such as serial number, part number, manufacturing location, etc. • POST and Miniboot version information • Embedded Linux O/S Name and version information • Hardware and tamper status registers • Flash dram and bbram sizes
time	Gets and sets the time on the coprocessor.
usbserial	Demonstrates how to use the adapter's USB port with a USB → Serial dongle to send and receive data. Contact Crypto (crypto@us.ibm.com) for additional information.

y4lib

IBM provides a stub library to assist customers who are implementing (or porting) standalone Toolkit applications. This library, called y4lib, can be used to minimize changes when porting applications from previous adapters as well as minimize the impact of the removal of the pthreads library from coprocessor side applications. Please refer to the internal documentation in the y4lib sample (and others) for a more in depth explanation.

UDX sample

The UDX sample directory contains a sample program that demonstrates how to write a simple user-defined extension to CCA for the workstation. UDXes for IBM Z installations must be written by IBM per policy dictated by IBM management.

2.1.1.12 shells directory

The *cctk/<version>/shells* directory contains various versions of the init.sh segment 3 startup script invoked when control is passed to segment 3 on the adapter.

Control is passed from segment-2 to segment-3 during the Linux boot process inside the coprocessor. Once segment-2 (Linux) successfully boots, the last step in the process is to call /flashS3/init.sh to transfer control to segment-3. The completion of EMBURN or REMBURN commands can cause control to be transferred as well as CLU RS operations, powering on the server in which the adapter resides, and resetting the host device driver.

Depending on the options specified during the image creation process one of these makefiles will be incorporated into the JFFS2 image downloaded to the coprocessor. Typically, developers will not need to modify these scripts, since the toolkit and UDX samples are standardized and most coprocessor applications do not take command line arguments due to the nature of the coprocessor. However, customers can modify this script if it suits their needs to do so. Customers should note that Linux requires that the initialization script be a plain text file saved with UNIX style line feeds, rather than DOS style line feeds. When editing the initialization script to be built into the JFFS2 image, ensure that the file is saved with UNIX style line feeds. See the JFFS2 discussion in “Development components” on page 5 for details about JFFS2.

2.1.1.13 Signing directories

The *cctk/<version>/signing* directories contain makefiles and test keys which can be used to sign development images for testing and test keys that can be used as a demonstration of an example signing scenario. These keys should only be used for development and testing of the signing process, as they have been certified by IBM for use ONLY with the Toolkit development CLU files.

Warning

Under no circumstances should these keys be used for production.

The signing samples are simply an example of how to use Signer.

Depending on the customer’s security needs, additional steps may need to be taken to improve the overall security of the signing process. For example, CCA roles and profiles may need to be installed on the adapter and the user may need to login with a specified password in order to sign.

Additionally, to make the signing sample easier to use for all customers, the sample keys have been returned in the clear. It is recommended that for actual production applications, the signing keys be enciphered under some other key, typically the master key for the adapter. Security precautions may also need to be taken to limit physical access to the keys and to separate roles and responsibilities so that the proper security officers are in place for each aspect of the signing process. If there are any questions about the security process, each customer should contact their security architect or toolkit provider.

See Roles and Profiles on page 63 for a sample setup of the roles and profiles needed to implement a reasonable security process.

2.2 Installing the coprocessor device driver

The device driver is part of the CCA install downloaded from the Download software page of the IBM CryptoCards website and it is required that the device driver has been installed as a prerequisite for using the Toolkit.

In order to comply with federal export regulations, customers need to specify their customer order number and adapter serial number when downloading the device driver and CCA installation package. In addition, this information is required when IBM processes a warranty replacement request for the adapter. Customers are strongly urged to maintain this information for all adapters purchased from IBM.

2.2.1 Linux

To verify that the device driver has been installed and is currently loaded, the lsmod command can be used as follows:

```
      lsmod | grep ibm4767
```

If this command returns with a line similar to:

```
      ibm4767              176206  0
```

Then the device driver has been loaded. The first column represents the module name. The second column (176206) is memory size of the module in bytes. This can vary slightly between operating systems and releases. The third column (0) is a use count. The value is zero when the module is not in use and nonzero when the module is in use.

If this command returns no output, then the *ibm4767_load* script can be run as root to attempt to load the device driver. If *lsmod* returns no output after an *ibm_4767_load* command, it is likely that the host support program has not been installed or not been installed correctly. To check for installation problems, /tmp/IBM4767_driver_install_log.txt may contain information as to what (if anything) failed during the driver build process. Additionally, these scripts along with a script to rebuild and install the device driver from source can typically be found in /usr/src/ibm4767/scripts.

2.2.2 Windows

To verify that the device driver has been installed and is currently loaded, use the Windows Device Manager. The IBM 4767 should be displayed as “IBM 4767 PCIe Cryptographic Adapter”.

3 Developing and debugging a coprocessor application

This chapter describes how to use the Toolkit to create the coprocessor-side piece of a coprocessor application and load it into the coprocessor.

The host-side piece of a coprocessor application may be built in the same manner as any other application. The only requirements are to use the appropriate compiler options to ensure the directories listed in “Include file directory search order” on page 28 are searched in the proper order, define the appropriate host-side variables listed in this chapter, and link in with the host-side library *libcsulcca.so*.

This chapter is a high-level overview of the major steps of the development process, including:

- Each step in the development process
- Special coding requirements for development
- Required option and switch settings for the compiler, assembler, linker
- How to build a JFFS2 filesystem image containing the coprocessor-side piece of the coprocessor application
- How to load the JFFS2 filesystem into the coprocessor
- How to start the debugger

“Overview of the development process” on page 40 has an in-depth tutorial of the process.

3.1 Environment variables

The examples and the syntax diagrams for the toolkit utilities in this chapter assume the directory that contains the various utilities shipped with the toolkit is in the search path for executable files (i.e., the PATH environment variable includes *cctk/<version>/bin/host/<platform>*).

It may be necessary to make other changes to the path or set other environment variables in order to invoke the compiler, assembler, and linker from the command line.

The following environment variable must be set to build either piece of a coprocessor application and to build the segment 3 JFFS2 image.

- CCTK_FS_ROOT must be set to point to the root of the Toolkit (that is, the fully qualified path to *cctk/<version>*). For example, if the toolkits are unzipped/untarred from /home/user, then CCTK_FS_ROOT becomes */home/user/cctk/<version>*.

To build the coprocessor-side piece of a coprocessor application (which is supported on Linux), additional environment variables that tell the makefile where to find the cross-compiler must be defined:

- CROSS must be set to the root directory containing the cross-targeted compiler, assembler, and loader. For example, if the cross-compiler is located in /home/user/cross, CROSS becomes /home/user/cross, as in *export CROSS=/home/user/cross*.

- GCC_NAME must be set to the prefix to use with the standard compiler, assembler, and loader names to create the corresponding cross-targeted tool names. For example, GCC_NAME is ppc476-ibm-linux-gnu-, as in *export GCC_NAME=ppc476-ibm-linux-gnu-*.

  To build the segment-3 image, the following environment variables must be set:

- ICAT_FS_ROOT must be set to point to the root of the ICAT installation when building a segment-3 binary image (that is, the fully qualified path to *cctk/<version>/debuggers/<platform>/icatpzx-nnn*, where *nnn* is the version of the debugger and where *platform* specifies the OS being used). For example, if the toolkits are unzipped/untarred from /home/user on Linux, then ICAT_FS_ROOT becomes */home/user/cctk/<version>/debuggers/linux/icatpzx-nnn*.

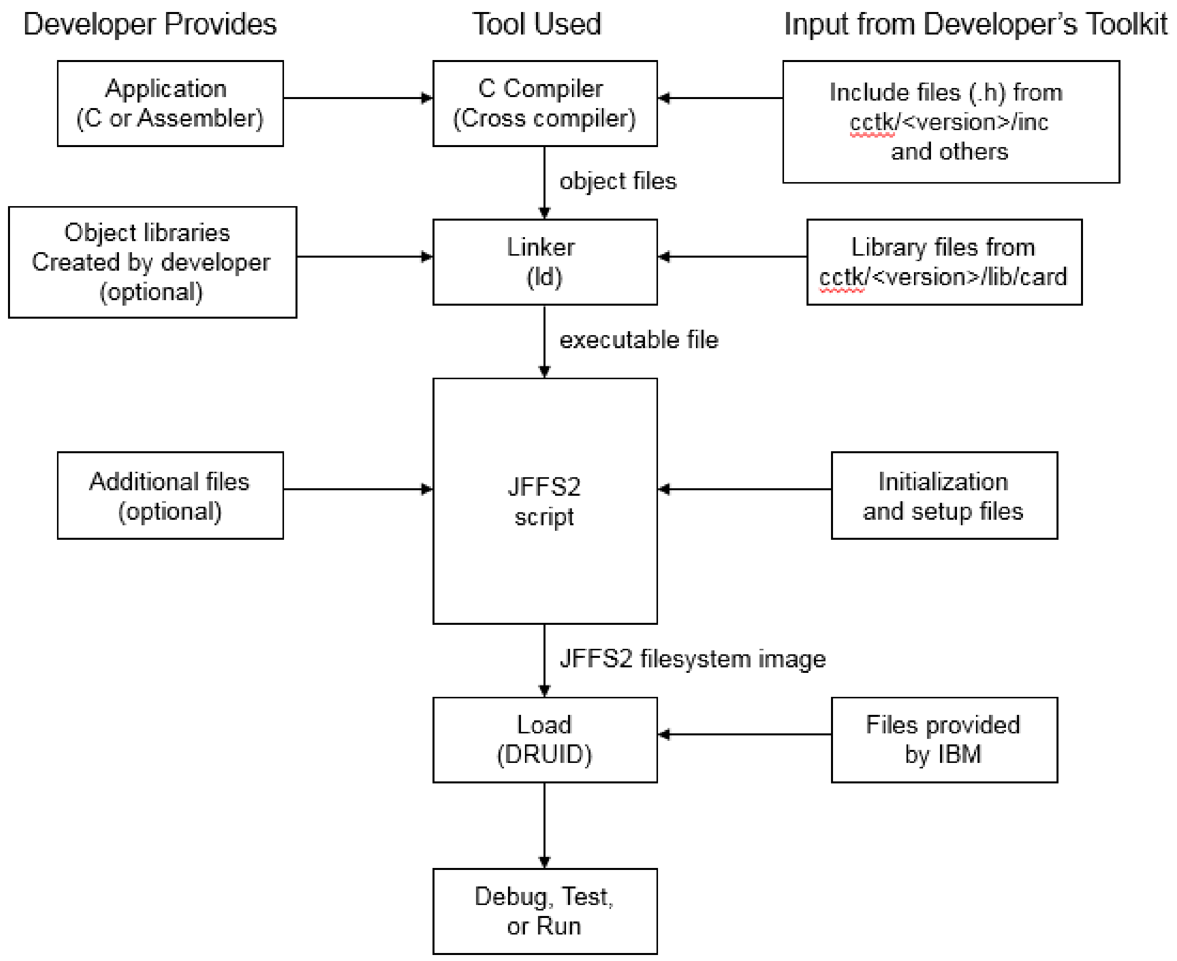
- CCTK_JFFS2_DIR must be set to point to the directory that contains the mkfs.jffs2 utility. For example, if mkfs.jffs2 is installed in /usr/sbin, then CCTK_JFFS2_DIR becomes */usr/sbin*.

3.2 Coprocessor-side development process road map

As discussed in “Introduction”, the procedure to build the coprocessor-side piece of a coprocessor application and load it into the development coprocessor consists of the following steps:

1. Cross-compile and link your coprocessor-side source code into an executable.
2. Build a JFFS2 filesystem containing your coprocessor-side application as well as all files required by IBM.
3. Load the JFFS2 filesystem into the coprocessor.

Figure 3 illustrates the development process, and indicates the name of the tool and input needed to perform each step. The process is identical to that shown in Figure 1 on page 4; this chart simply provides more detail.



Developer Provides
Tool Used
Input from Developer's Toolkit
Application (C or Assembler)
C Compiler (Cross compiler)
Include files (.h) from cctk/<version>/inc and others
object files
Object libraries Created by developer (optional)
Linker (ld)
Library files from cctk/<version>/lib/card
executable file
Additional files (optional)
JFFS2 script
Initialization and setup files
JFFS2 filesystem image
Load (DRUID)
Files provided by IBM
Debug, Test, or Run

Figure 3 Development process road map

The following sections detail how to use the Toolkit to perform the Development Process Road Map steps.

3.3 Special coding requirements during development

3.3.1 Developer identifiers for non-UDX toolkit applications

The coprocessor-side piece of a coprocessor application must register with the Communication Driver on the coprocessor by calling xcAttachWithCDUoption and then calling xcInitMappings. This must be done to direct the coprocessor's communication manager to map the calling application's address space into buffers the communications manager uses to hold requests and replies before the coprocessor-side piece can receive requests from the host.

The coprocessor-side piece must supply a "developer identifier" that uniquely identifies the application as part of the registration process [1]. During development, a developer may use any unused value for the developer identifier [2]. Before an application can be released, the developer must obtain a unique identifier from IBM and must rebuild the application and any host application that interacts with it to use the true identifier. UDX applications inherit the developer identifier from CCA and do not need to specify an identifier.

3.3.2 Attaching with the debugger

The coprocessor-side piece of an application that has been downloaded to the coprocessor can be launched by the *init.sh* shell script incorporated into the JFFS2 image. If the debugger daemon was incorporated in and started by *init.sh*, then the application can be debugged using ICAT. If the application was not started by the shell script, the user can launch the coprocessor-side piece from the debugger by specifying the full path to the application.

To ensure a running application does not make too much progress before the debugger takes control, the developer should code an infinite loop early in the application and use the debugger to move the execution point past the loop after the application is quiesced. To ensure the loop does not starve other agents in the system, the loop should be coded in a manner similar to this:

```
{
  unsigned long i,j;
  i=j=0;
  for (;;)
  {
    sleep(1);
    i++;
    if (j == 28)
      /* Make sure optimizer doesn't remove all code after loop. */
      break;
  }
}
```

After attaching to the application with the debugger, set a breakpoint on the i++ statement and allow the application to run. When the breakpoint is hit, change the value of j to 28, and step out of the spin loop, or use the debugger's Jump to location function to move the execution point to the statement immediately following the loop. See the *ICAT Debugger Getting Started* manual for more complete instructions.

Note: This loop is not necessary for UDX applications because the coprocessor side UDX code will not be executed until a host-side request for the given verb has been sent from the host.

3.4 Compiling, assembling, and linking

This section lists options that must be specified when compiling, assembling, or linking to ensure that the coprocessor-side piece of a coprocessor application will run properly. Other options may also be specified as long as they do not conflict with the options listed in this section.

[1] Refer to the description of xcAttach in *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference* for details.

[2] The coprocessor-side debugger daemon (zdaemon) reserves the agent ID's 0xF0FF and 0xF1FF for PCI communication. CCA reserves the agent ID 0x4341 ("CA" in ASCII). These agent ID's should not be used for development purposes.

The Toolkit includes makefiles that specify the proper options for each sample. See “Overview of the development process” on page 40 for details on their use.

The C Runtime Library installed as part of the Linux OS on the coprocessor behaves in a similar fashion to that of a standard ANSI C installation with a few exceptions. Most notably, I/O routines do not have access to a system console, and cannot handle serial attachments.

3.4.1 Compiler options

Building applications with the Toolkit requires a gcc cross-compiler (which must be installed separately). A sample script for building the cross-compiler is in the Toolkit in *cctk/<version>/cross_compiler_scripts*. Please read the prologue of *build_4767_cross_compiler.sh* in that directory for details about building the cross-compiler.

Note: When using the cross-compile environment makefiles, make certain that the environment variables CROSS and GCC_NAME are defined so that the makefile chooses the gcc cross-compiler instead of the native gcc.

To make sure the gcc cross-compiler was used to build a card-side executable, run the file command on the executable. For example:

```
file ~/cctk/<version>/samples/toolkit/rte/card/gcc/sampleCardApp
```

and review the output, which should look something like this:

```
ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.18...
```

In contrast, a file built with the host compiler will look something like this:

```
ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.32...
```

Table 2 describes the various compiler switches and whether they are applicable to a given system.

Table 2 Compiler switches

Switch	Function	Coprocessor	Linux Host	Windows Host
-DDEBUG	Ensure test cases enter a spin loop prior to execution. Use only for debugging. Valid for toolkit samples only.	Yes if debugging, no otherwise	No	No
-DLINUX_ON_INTEL	Compile code for Linux.	No	Yes	No
-DLINUX	Compile code for Linux.	Yes	Yes	No
-DLINUX_ON_PPC	Compile code for the coprocessor, rather than the host.	Yes	No	No

-DHOST32BIT	Define 32-bit addressing for card side code.	Yes	No	No
-DHOST64BIT	Define 64-bit addressing for your host side application.	No	Yes	Yes
-DLOGMSGD	Enable card-side logging.	Yes	No	No
-m64	Force binaries built to be 64-bit executables.	No	Yes	No
-DHOSTLOG		No	No	Yes
-DNOBOOLEAN	Specify boolean support for Windows.	No	No	Yes
-DWINDOWS	Specify a Windows host build.	No	No	Yes
-DLITTLE_ENDIAN	Specify endianness for Windows.	No	No	Yes
-DNT_ON_I386	Legacy specifications for a Windows build.	No	No	Yes
-DWORDSIZE=64	Specify word size on Windows.	No	No	Yes
-gstabs+	Produce stabs debugging information. Needed for the ICAT debugger.	Yes if debugging, no otherwise	No	No
-Wall	List all warnings	Yes	Yes	No
-Werror	Treat warnings as errors.	Yes	Yes	No
-Wstrict-prototypes	Force prototype warnings	Yes	Yes	No
various optimization switches	Optimize compiled code.	No if debugging	N/A	N/A

Notes:
HOST32BIT should ONLY be defined for card-side applications. The name HOST32BIT may be confusing for card-side applications, but it should be interpreted to mean that the system that hosts the code is 32 bit. In this case, the code runs (is hosted) on the coprocessor which is a 32-bit environment. The use of the term HOST is overloaded.

HOST64BIT should ONLY be defined for host-side applications.
Technically the -Wall, -Werror, and -Wstrict-prototypes are not required to run applications, but they are strongly advised.

When in doubt about how to compile host or card side samples for a given operating system, please refer to the sample makefiles provided with the toolkit.

IBM provides standalone makefiles that need to be run from a command prompt. Use the Microsoft Visual Studio vcvars64.bat file, which is typically installed in “C:\Program Files (x86)\Microsoft Visual Studio <version>\VC\bin\amd64\vcvars64.bat”.

3.4.1.1 Include file directory search order

The appropriate compiler options pertaining to include search path order (typically compiler option -I (capital I) should be used when building either piece of a coprocessor application to ensure the following directories are searched for include files in the order shown:

1. *cctk/<version>/inc*
2. *cctk/<version>/inc/shared/include*
3. *cctk/<version>/inc/xcmanager*

Additional user-defined include search paths can be added to specify customer or sample specific includes. These includes should always be specified such that they come after the list specified above. Typically the sample makefiles allow for the definition of HOST_USER_INCLUDES and CARD_USER_INCLUDES for additional host and coprocessor side include search directories.

Note: Should customers writing standalone Toolkit applications wish to use the y4lib code, which abstracts much of the fibers implementation, the include files in cctk/<version>/samples/y4lib/inc should be specified LAST in the include file search order and the corresponding headers (y4_xc_api.h, y4_pthread.h, and y4_xc_oa.h) must be specified after all other includes. This is reflected in all toolkit samples.

The best practice for search order for includes is:

1. Linux includes, for example, string.h, stdlib.h, stdint.h
2. CCA/Toolkit includes from cctk/<version>/inc directory (and its children) as needed
3. Your custom includes (for your UDX or toolkit application)
4. If using the toolkit shim layer, toolkit shim includes (y4_xc_api.h, y4_pthread.h, etc)

For examples, see the makefiles in the samples directory (*cctk/<version>/makefiles)*.

3.4.1.2 Using the correct cross-compiler

When compiling a UDX or toolkit application, users must ensure that the proper cross-compiler is used. The cross-compilers for the IBM 4765 and IBM 4767 do not produce compatible output. If you cross-compile the UDX sample for the IBM 4767 using the cross-compiler for the IBM 4765 and attempt to load the resulting binary file on the coprocessor, your application will not run.

3.4.2 Linker options

The compiler used determines which linker must be used to create an executable file from the resulting shared object (.so) files or dynamic load libraries (.dll).

Library files that may be linked with a coprocessor-side application

cctk/<version>/lib/card/libxccmn.so	Common card-side functions for tasks such as key generation, encryption, decryption, hashing, signature generation and verification, etc.
cctk/<version>/lib/card/libxcoa.so	If the application will use Outbound Authentication functions.

Library files that may be linked with a host-side application

/usr/lib64/libcsulcca.so	Any host-side toolkit or UDX application on Linux. For convenience, a copy of the CCA host library is included in cctk/<version>/lib/host/linux. For MTM customers, a copy of libcsulcca.so is installed by the CCA installer, typically in /usr/lib64.
cctk/<version>/lib/host/linux/debug/libcsulcca.so	Any UDX application being debugged that needs to print out CPRB data for inspection. Note: in order to print out the CPRB data, the CSUDUMP environment variable must be set to “y” in the window from which the application is invoked and this library must be in the system's LD_LIBRARY_PATH such that it will be picked up BEFORE the normal libcsulcca.so library.
C:\Windows\System32\csuncca.dll *C:\Program Files\IBM\4767\lib\csuncca.lib*	Any host-side toolkit or UDX application on Windows. Note: This includes the host side device driver library (formerly ibm4765w.dll for the 4765).

3.5 Using the adapter’s Ethernet port

The IBM 4767 supports Ethernet communication via a USB-to-Ethernet dongle. There are many commercially available dongles that use one of these chipsets. If you have any questions about the chip sets or dongles to use with the USB port, please contact your Toolkit provider. Special consideration should be taken if your application chooses to use the ethernet port b/c doing so can impact the security of your application. For example, use of the ethernet port is not controlled by Miniboot and it opens your application up to the possibility of loading (or leaking) data (or code) that has not been sufficiently verified.

3.6 Building JFFS2 filesystem Images

The development process requires the creation of a JFFS2 (Journaling Flash Filesystem Version 2) filesystem image mirroring the expected /flashS3 filesystem on the card that can be loaded into the coprocessor using DRUID or can be signed using CRUSIGNR and optionally placed into a CLU file by CRUPKGR for subsequent download by CLU. MKFS (MaKe File System) for JFFS2 is a Linux utility and should be invoked from a Linux system. All JFFS2 images to be downloaded into the coprocessor should be built from a Linux host.

The preferred method to build JFFS2 filesystem images is to use the makefile provided in *cctk/<version>/build_seg3_image*. The makefile should be invoked from the *cctk/<version>/build_seg3_image* directory as follows:

```
cd ~cctk/<version>/build_seg3_image
make -f cctk.seg3.image.mak SAMPLE_NAME=samplename
     [ BUILD_TYPE={debug|prod} ]
```

where *samplename* is the name of the subdirectory under *cctk/<version>/samples/toolkit* that contains the coprocessor-side piece of the application (which must in turn be in the *card/<toolset>* subdirectory of the *samplename* directory, and must be named *sampleCardApp*), and the BUILD_TYPE constant determines which initialization script (debug or production) is incorporated into the JFFS2 image. The default BUILD_TYPE is prod. The *samplename* must be *wks* to build a workstation UDX.

This makefile gathers the files necessary to create a local copy of the flashS3 directory to be loaded onto the adapter (with DRUID or CLU after it has been appropriately signed and formatted into the appropriate Miniboot command using SIGNER) as a JFFS2 filesystem.

Once the files are gathered into the local copy of the flashS3 directory, their permissions and owners are modified to comply with expected values and a JFFS2 filesystem image is created from the local copy of the /flashS3 directory.

Listed below is an example of the output from a JFFS2 build. In this example, *cctk/<version>* has been substituted for the directory that was defined by CCTK_FS_ROOT.

```
user@server:cctk/<version>/build_seg3_image> make -f
     cctk.seg3.image.mak SAMPLE_NAME=rte BUILD_TYPE=prod
----------Start - Removing flashS3 directory-------------------
sudo rm -rf  ./flashS3
user's password:
----------End - Removing flashS3 directory---------------------
----------Start - Create segment-3 image directory-------------
sudo mkdir ./flashS3
----------End - Create segment-3 image directory---------------
----------Start - Copy binaries into flashS3-------------------
sudo cp /home/user/cctk/<version>/shells/cc.toolkit.prod.init.sh
./flashS3/init.sh
sudo cp /home/user/cctk/<version>/bin/card/cryptologkd ./flashS3/.
sudo cp /home/user/cctk/<version>/lib/card/libxccmn.so.1.2.0
./flashS3/.
sudo cp /home/user/cctk/<version>/lib/card/libxcoa.so.1.2.0  ./flashS3/.
cd ./flashS3;sudo ln -s  libxcoa.so.1.2.0 libxcoa.so
cd ./flashS3; sudo ln -s  -s libxcoa.so.1.2.0 libxcoa.so.1
cd ./flashS3; sudo ln -s  -s libxccmn.so.1.2.0 libxccmn.so
cd ./flashS3; sudo ln -s  -s libxccmn.so.1.2.0 libxccmn.so.1
sudo cp
/home/user/cctk/<version>/samples/toolkit/rte/card/gcc/sampleCardApp ./
flashS3/.
----------End - Copy binaries into flashS3-------------------
----------Start - Fixup permissions of flashS3---------------
sudo chmod -R 550 ./flashS3
sudo chmod 555 ./flashS3/init.sh
sudo chown -R 501.0 ./flashS3
sudo chmod a+rx ./flashS3
sudo chmod a+rx ./flashS3/*
----------End - Fixup permissions of flashS3-----------------
----------Start - Create JFFS2 image------------------
/home/user/bin/mkfs.jffs2 -e 128KiB -b -r ./flashS3 -o
cctk.rte.prod.`date "+%Y%m%d%H%M"`.bin
Segment-3 image cctk.rte.prod.201604051210.bin created.
----------End - Create JFFS2 image -------------------------
```

In this example, the resulting image file is named *cctk.rte.prod.<timestamp>.bin*.

The image file contains the contents of the *cctk/<version>/build_seg3_image/flashS3* directory as a JFFS2 filesystem. This *flashS3* directory's contents form the basis for what is loaded on the IBM coprocessor with DRUID. IBM's standard convention is to use the .bin extension for these images, but this is not strictly required. Typical contents of the /flashS3 directory, once loaded on the coprocessor, is shown below.

```
-r-xr-xr-x  1 501  501    2504 Oct 31 19:38 cducfg
-r-xr-xr-x  1 501  501   12888 Oct 31 19:38 cryptologkd
-r-xr-xr-x  1 501  501    3156 Oct 31 19:38 init.sh
lrwxrwxrwx  1 501  501      17 Oct 31 19:38 libxccmn.so -> libxccmn.so.1.2.0
lrwxrwxrwx  1 501  501      17 Oct 31 19:38 libxccmn.so.1 -> libxccmn.so.1.2.0
-r-xr-xr-x  1 501  501 2539052 Oct 31 19:38 libxccmn.so.1.2.0
lrwxrwxrwx  1 501  501      16 Oct 31 19:38 libxcoa.so -> libxcoa.so.1.2.0
lrwxrwxrwx  1 501  501      16 Oct 31 19:38 libxcoa.so.1 -> libxcoa.so.1.2.0
-r-xr-xr-x  1 501  501   81605 Oct 31 19:38 libxcoa.so.1.2.0
-r-xr-xr-x  1 501  501   36711 Oct 31 19:38 sampleCardApp
```

For more information on this process, please refer to the prologue and comments in the cctk.seg3.image.mak makefile and associated include files.

3.7 Downloading images to the coprocessor

Once a JFFS2 filesystem containing the of the application has been generated, the filesystem may be downloaded to the coprocessor using DRUID if the coprocessor has been prepared as a development environment as specified by "Preparing the development platform" on page 40.

DRUID does not affect any data in the nonvolatile memory (battery-backed RAM and flash) associated with the application. That is, druid writes to /flashS3 and then on the card after the REMBURN3 is complete, the normal procedure on the card is to transfer control to segment-3 by invoking /flashS3/init.sh. Typically this initialization script will not alter the contents of /bbram unless there is something specific in your version of this script or your code that would alter bbram. If the developer wants to clear state that has accumulated during prior debug sessions so that the application will start with a clean slate, the developer should first download *emergency_reload_seg3_OID6_r.j.m.clu* to the coprocessor using CLU.

The syntax of the command is

```
      druid [-l log_file] [-a adapter_number] [-d segment-3_image_filename]
```

where

- *image_filename* is the name of the file containing the JFFS2 filesystem image to download to the coprocessor. Path information must also be provided if the file is not in the current directory.
- *adapter_number* identifies the adapter to which the read-only disk image is downloaded. (More than one adapter may be installed in a host.) The default is 0.
- *log_file* is the name of the log file DRUID will use when loading the adapter.

The number assigned to a particular adapter 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.

DRUID displays a summary of the status of the coprocessor similar to that of a CLU ST command before it downloads the application. The summary includes:

- Coprocessor's serial number [3]
- Current boot count (see "Targeting arguments" on page 84 for details)
- Name, creation date, and size of the image file last downloaded to the coprocessor
- Name of the file containing the public key associated with the application currently loaded in the coprocessor [4]

Note: If you run DRUID to download an image when the coprocessor has CCA loaded or has nothing loaded, DRUID will fail because the adapter is not in the proper state. You may see an error similar to:

```
--ERROR-- Adapter is not in a druid-able state.
--ERROR-- OWNER2 Must be 3, but it is 2.
--ERROR-- OWNER3 Must be 6, but it is 2.
```

Note: DRUID can return messages that are informational only and do not indicate a "true" error condition.[5] For example:

```
Remburn3 0x85400080  on xcMBRequest
```

or

```
--ERROR-- Reload Segment-3  command failed.
--ERROR-- Return Status: 0x85400081  (return code 0x       0)
--ERROR-- The CDU daemon is not attached.
```

After DRUID completes, make sure you wait for the coprocessor to reboot and initialize. This normally takes 3-5 minutes. If you attempt to communicate with the coprocessor before it has initialized itself, you may receive a 0x80400013 error.

On Linux, if your application is CCA, a UDX, or Toolkit application which emits messages to the host through syslog calls, you can see that the application has started by using the following command:

```
sudo tail -f /var/log/messages
```

For example, the Reverse-then-echo sample will print a message to syslog similar to:

```
Dec  5 03:13:47 server: ibm4767: MESSAGE FROM YH10DV63N362 => v1, f1, Sev 6,
"Dec  5 08:14:36 sampleCardApp: CCTK rte sample started..."
```

On Windows, druid is named druid.exe. You can see messages sent to the host by copying and viewing the files in the 4767 message logs, which are in the logs directory of your IBM 4767 installation. Typically, this directory is "C:\Program Files\IBM\4767\logs".

Linux on the coprocessor loads and runs the initialization script file after the coprocessor is rebooted. See "How to reboot the coprocessor" on page 55 for a description of how to reboot the coprocessor.

[3] That is, the value *xcGetConfig* returns in *pConfigData->VPD.sn*. Refer to the *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference* for details.
[4] That is, the value of *pubkey_fn* supplied when DRUID last downloaded an application to the coprocessor.
[5] These errors can occur when a CDU operation does not complete. CDU may not complete if the application is not CDU-able (as is the case with all standalone toolkit applications). If CDU does not complete, DRUID will reset the adapter to finish the loading process.

Linux requires that the initialization script be a plain text file saved with UNIX style line feeds, rather than DOS style line feeds. When editing the initialization script to be built into the JFFS2 image, ensure that the file is saved with UNIX style line feeds. The easiest way to ensure this is to run the dos2unix command or equivalent on the file after editing or to use vim's ":set list!" and visually search for ^M characters.

3.7.1 Using syslog

Additionally, it is possible for a coprocessor-side application to log messages back to the host through syslog calls. These syslog calls will be intercepted by the crypto logging dameon (cryptologkd) on the coprocessor and then routed to syslog (or equivalent) on the host where the coprocessor resides.

Notes:

1. Do not put security relevant data (secrets) in the syslog.
2. Limit their number or you may flood the host with messages.
3. Best practice is to conditionally compile syslog messages in for debugging.
4. Customers may wish to use rsyslog instead of syslog-ng
5. Be careful when specifying the type (severity) of a syslog call.
6. Certain severity levels will be treated as a significant error and will reset the adapter. Examples include LOG_EMERG and LOG_ALERT as defined in syslog.h.

3.8 Debugging

After the application is running, it can be debugged using the ICAT debugger if the debugger daemon has been started on the coprocessor. Refer to *IBM 4767 PCIe Cryptographic Coprocessor Interactive Code Analysis Tool (ICAT) Getting Started* for details.

Important Notes:

1. The coprocessor-side debugger daemon (zdaemon) will run only in a development environment where the ownerID corresponding to segment 2 is 3, and where segment 3 ownerID is 6. Additionally, zdaemon should only be incorporated into development JFFS2 images, and should never be included in any image signed with the signer utility intended for a production environment.

2. All sample applications, including CCA applications, should display a startup message that is logged back to the host's */var/log/messages* file. The ICAT debugger daemon also logs a startup message to */var/log/messages*. When debugging, most startup messages are blocked by the debug spin loop, so if you are debugging, you should look for zdaemon's startup message instead of your application's startup message in */var/log/messages*.

3.9 Parsing a CLU file

CLU files can be parsed into human-readable (XML) format using the parseclu utility. This may prove useful during the signing process to verify the contents of signed images. This utility requires a CLU file type designation and CLU file as input. The XML output is best viewed in a Web browser such as Firefox.

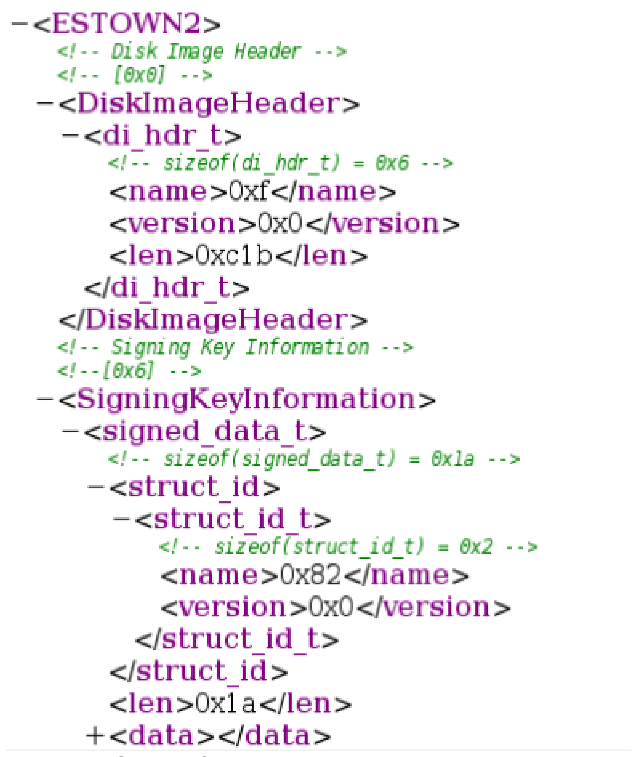
The syntax of the command is:

```
parseclu -t [type] -i [input file name] > [output file name]
```

where

- *type* is the type of the CLU file to be parsed: workstation CLU file, HUL file, meta data file, packaged clu file, CLU stripped file, zUDX package file, or Z master directory listing file
- *input file name* is the name of the CLU file to be processed. Path information must also be provided if the file is not in the current directory.
- *output file name* is the name of the file parseclu will produce when parsing the input file. Path information must also be provided if the file is not in the current directory.

Typical output from parseclu looks like this (this is only the first section; the entire file will be much larger).



```
-<ESTOWN2>
   <!-- Disk Image Header -->
   <!-- [0x0] -->
  -<DiskImageHeader>
    -<di_hdr_t>
       <!-- sizeof(di_hdr_t) = 0x6 -->
       <name>0xf</name>
       <version>0x0</version>
       <len>0xc1b</len>
     </di_hdr_t>
   </DiskImageHeader>
   <!-- Signing Key Information -->
   <!--[0x6] -->
  -<SigningKeyInformation>
    -<signed_data_t>
       <!-- sizeof(signed_data_t) = 0x1a -->
      -<struct_id>
        -<struct_id_t>
           <!-- sizeof(struct_id_t) = 0x2 -->
           <name>0x82</name>
           <version>0x0</version>
         </struct_id_t>
       </struct_id>
       <len>0x1a</len>
      +<data></data>
```

Figure 4 Output from parseclu

4 Packaging and releasing a coprocessor application

The design for the coprocessor was motivated by the need to satisfy simultaneously the following requirements:

1. Code must not be loaded into the coprocessor unless IBM or an agent IBM trusts has authorized the operation.
2. Once loaded into the coprocessor, code must not run or accumulate state unless the environment in which it runs is trustworthy.
3. Agents outside the coprocessor that interact with code running on the coprocessor must be able to verify that the code is legitimate and that the coprocessor is authentic and tampering with the coprocessor has not occurred.
4. Shipment and configuration of coprocessors and maintenance on and upgrades to code inside a coprocessor must not require trusted couriers or security officers.
5. IBM must not need to examine a developer's code or have any knowledge of a developer's private cryptographic keys in order to make it possible for customers to load the developer's code into a coprocessor and run it. This means that when IBM authorizes a customer to release code by providing enablement files and a segment owner ID, they are simply stating that IBM has authorized the given customer to release their code. IBM can not make ANY statement regarding the code (or security thereof) because IBM has not inspected the code in any way.

To meet these requirements, the design defines four "segments":

- Segment 0 is ROM and contains one portion of "Miniboot" and one portion of POST. Miniboot is the most privileged software in the coprocessor and among other things implements the security protocols described in this section. POST (Power On Self Test) is responsible for verifying various levels of hardware functionality. Segment-0 is loaded in a secure procedure at the manufacturing facility and because segment-0 is ROM, it can only be loaded once. After it is initially loaded at the manufacturing facility, segment-0 can not be updated. Because segment-0 can not be updated, the portions of POST and Miniboot residing in this segment contain a minimal subset of functionality.

- Segment 1 is flash and contains the other portion of "Miniboot". The division of Miniboot into a ROM portion and a flash portion preserves flexibility (the flash portion can be changed if necessary) while guaranteeing a basic level of security (implemented in the ROM portion). While segment 1 can be updated, typically such updates are rare. Every segment 1 updates creates a new entry in the adapter's internal certificate chain list. The space for this list is somewhat limited, so that only several hundred such certificates can be stored. By design, if the certificate space is exhausted, the adapter's segment 1 can no longer be updated since the adapter would not be able to trace back its certificate chain to the factory-fresh state. Typically, IBM updates segment 1 less than once per year, so there is no need to attempt to update segment 1 in between official IBM segment 1 releases.

- Segment 2 is flash and contains the coprocessor operating system, which is a Linux micro kernel.

- Segment 3 is flash and contains one or more coprocessor applications, such as CCA, a UDX, or a standalone Toolkit application.

The security protocols that enforce these design goals are based on ECC keypairs and a notion of who owns the code in each segment. IBM owns segments 0, 1, and 2 and issues an owner identifier to any party that is developing code to be loaded into segment 3. The coprocessor saves the identity of the owner of each segment and an ECC public key for each segment. The key is provided by the segment's owner.

The coprocessor does not accept a command that changes the contents of a segment unless the command is digitally signed with the private key that corresponds to the public key associated with the segment. The command must also correctly identify the owner of the segment. Commands that must change the contents of a segment that does not yet have a public key must be signed with the private key that corresponds to the public key associated with the segment's parent. For example, the command that initially sets the owner and public key for segment 3 must be signed with the private key for segment 2. For commands that require the segment's parents signature, IBM will provide these as part of the enablement process. For Toolkit and UDX customers, this will consist of an ESTOWN3 and ESIG3 command as well as a few public key certificate files.

The files shipped in the Toolkit are designed to make it easy for a developer to start work immediately but are also constructed in a way that does not threaten the security or integrity of an application deployed in the field or one that may be deployed in the future. During development, the developer uses a default ECC keypair (which makes development easy) that is tied to a generic owner identifier (which makes the generic keypair "harmless"). Specifically the development ownerID for segment-2 is 3 and segment-3 is 6. When the developer is ready to deploy an application in the field, the developer must obtain a unique developer identifier from IBM and must generate a new, unique ECC keypair. This is summarized in Table 3.

Table 3 Developer identifiers

Attribute	Development	Production
Owner	"Generic developer" Segment-2 Owner ID 3 Segment-3 Owner ID 6	Developer-unique identifier for Segment-3. Segment-2 Owner ID is 243 for workstation code.
Public Key	Generic (common) keys. For Segment-3, these keys are shipped in the *sample_keys* directory of the toolkit.	Developer-generated key.

Prior to deployment, a developer must restore the coprocessor used for development to a state suitable for use in production using *surrender_ownership_seg2_toolkit_OID3_r.j.m.clu* [1]:

```
<CLU> -c pl -l /logfile-directory/clu_log -d
~/cctk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID3_r.j.m.clu
```

The developer must then install the CCA application on the coprocessor and configure a CCA test node. High-level instructions on how to complete these steps are included in chapters 3, 4, and 5, respectively, of the *CCA Installation Manual*. A more specific example (your needs may vary) can be found in the signing directory of the Toolkit.

[1] The examples in this chapter assume the directory that contains the various utilities shipped with the toolkit is in the search path for executable files (that is, the PATH environment variable includes *cctk/<version>/bin/host/<platform>*).

Loading CCA onto the coprocessor and configuring CCA as a signing node prepares the coprocessor for use by the signer utility (CRUSIGNR) and (optionally) the packager utility (CRUPKGR).

The developer generates three ECC keypairs using CRUSIGNR's KEYGEN function:

- S3KDEVPU.KEY / S3KDEVPP.KEY – The developer's segment-3 public/private keypair. The public key of this key pair will be an input to SIGNER and will be loaded into the adapter as part of EMBURN3 or REMBURN3 commands created by Toolkit customers. The private key of this key pair will be used as input to CRUSIGNR to create the appropriately signed EMBURN3/REMBURN3 CLU file which can be loaded into the adapter via a CLU PL command.
- DEVSGNPU.KEY / DEVSGNPP.KEY – A key pair used to authorize and indicate that a the file being signed is being signed by a party IBM recognizes. These private and public key pairs (along with a certificate returned by IBM as part of the enablement process) will be used as input to SIGNER when creating EMBURN3 / REMBURN3 / SUROWN3 commands. These keys ARE NOT loaded onto the adapter.
- DEVPKGPU.KEY / DEVPKGPP.KEY – A key pair used to authorize and indicate that the file being packaged is being packaged by a party IBM recognizes. These private and public key pairs (along with a certificate returned by IBM as part of the enablement process) will be used as input to PACKAGER when creating a packaged CLU file. These keys ARE NOT loaded onto the adapter.

Please refer to the scripts in *signing/cru/signer/tasks/keygen* for an example of how to create keys using Signer. Per IBM policy, all three key pairs must be generated, even if customers do not wish to package their application into a single CLU file. This standardizes and simplifies enablement processing for IBM and customers.

When generating these keys, customers have the option to encrypt the private key portions under the adapter's master key, or return the private key portion unencrypted (in the clear). If customers choose to generate their keys encrypted under the master key, the appropriate actions should be taken to ensure the master key can be regenerated. IBM recommends generating keys encrypted under the master key if possible. Refer to the *CCA Installation Manual* for details.

Users who do not wish to generate the private keys in the clear may also wish to establish a profile that restricts the actions CRUSIGNR and CRUPKGR can perform. See “Using Signer and Packager” on page 56 for more information.

The first keypair supplies the key to be saved with the developer's application in segment 3. The second and third keypairs are used by CRUSIGNR and CRUPKGR, respectively, to generate digital signatures that CLU uses to verify that IBM has authorized its use.

The KEYGEN function creates two KEY files, one containing both the private and public keys (for example, S3KDEVPP.KEY) and the other containing just the public key (for example, S3KDEVPU.KEY). The KEYGEN function also creates a file containing the hash of the public key. The file has the same name as the file containing the public key and an extension of HSH (for example, S3KDEVPU.HSH). After an appropriate contract has been signed, the developer forwards each public key file to IBM (for example, as e-mail attachments or as a zip file).

The developer must also send the hash value of each public key file to IBM using two channels to ensure an adversary has not tampered with the keys. IBM provides directions for the exchange of keys and control information as a part of the contracted services. IMPORTANT: customers should only send the public keys and corresponding hashes. Do NOT, under any circumstances, send the private keys to IBM.

The developer should retain the files containing the private keys and keep them in a secure place. They should *not* be sent to IBM or to any other third party. Should the developer lose these private key files, there is NOTHING IBM can do to help them reconstitute or recover these keys. IBM only has access to the public portions of the keys (and the corresponding hash of the public keys since these were sent as part of the enablement request process). Loss of the private keys will require customers to regenerate a new set of private and public keys, and IBM will have to create a new set of enablement files and assign a new owner ID. Customers will then have to remove their old code with the old owner ID and completely reload using the newly assigned owner ID and enablement files.

The developer obtains the following:

1. Certificates for the CRUSIGNR and CRUPKGR public keys (DEVSGNPU.CRT and DEVPKGPU.CRT, respectively). The developer provides these certificates as input to CRUSIGNR and CRUPKGR, as appropriate. By convention, various CLU inputs and outputs, such as keys, establish ownership files, and certificates, are prefixed with “prod_” or “test_”. This is done to differentiate their intended uses in Production or in Test. Under no circumstances should test files be used in production, or vice-versa.

   These files are generated by IBM from the CRUSIGNR and CRUPKGR public keys provided by the developer.

2. The following files generated by CRUSIGNR[2] and included with the IBM Toolkit:

- *establish_ownership_seg2_toolkit_OID243_r.j.m.clu*, which establishes ownership of segment 2.[3] Segment 2 must be owned before an application or an operating system can be loaded into the coprocessor. This file is shipped with the Toolkit.
- *emergency_reload_seg2_toolkit_OID243_r.j.m.clu*, which loads the coprocessor operating system into segment 2. The operating system must be loaded before an application can be loaded into the coprocessor. This file is shipped with the Toolkit.
- *reload_seg2_toolkit_OID243_r.j.m.clu*, which replaces an existing coprocessor operating system in segment 2. This file is shipped with the Toolkit.
- *surrender_ownership_seg2_toolkit_OID243_r.j.m.clu*, which surrenders ownership of segment 2. This removes the operating system and any application that has been loaded into the coprocessor and also clears any information the application has saved in nonvolatile memory [4]. This file is shipped with the Toolkit.
- *prod_es3xser.clu* (previously called *estown3.clu*), which establishes ownership of segment 3. IBM assigns the developer [5] an owner identifier and *estown3.clu* saves that value in the coprocessor. Segment 3 must be owned before an application can be loaded into the coprocessor. This file is generated by IBM when the developer sends its public keys to IBM.
- An emergency signature file (*prod_es3xser.sig*, previously named *esig3dev.sig*) that incorporates the developer's owner identifier and segment 3 public key. The developer provides this file as input to the signer utility (CRUSIGNR) when creating a file containing an EMBURN3 command,

[2] See “Using Signer and Packager” on page 57 for details on the contents of these files.
[3] The owner identifier assigned to segment 2 (typically 243 [0xF3]).
[4] Use of a common owner identifier for segment 2 makes it easier for an end user to obtain updates to the system software in segment 2 because IBM need only create one file containing the updates, and anyone with a coprocessor containing a custom application can use the file to perform the update. But it also makes it easier for someone to remove accidentally or maliciously from a coprocessor a developer's application and any data it has saved in nonvolatile memory, since surrender_ownership_seg2_toolkit_OID243_r.j.m.clu removes any custom application installed on a coprocessor regardless of the application's origin.
[5] That is, an OEM or an organization within an OEM.

which loads the developer's application into the coprocessor. This file is generated by IBM from the segment 3 public key provided by the developer.

The developer must build a version of the application (and then package it into a filesystem built with JFFS2) suitable for release. This version of the application (as a JFFS2 image) can be used as input to the EMBURN3 or REMBURN3 command. The developer will probably want to build without debug information or debug code and may want to enable optimization, and remove or limit syslog messages to reflect the desired level of logging for a production environment.

The details surrounding preparation of the application for distribution depend heavily on whether the distributor wants to restrict use of the application in some way (for example, by specifying that it can only be installed in a particular set of coprocessors) and on the particular conditions under which the distributor expects the application to be installed (for example, does the distributor need to package the application in a way that enables users of an earlier version to upgrade, or is it enough to supply a file that can be loaded into a coprocessor fresh from the factory). The signer utility provides a great deal of flexibility and a discussion of its full potential is beyond the scope of this document. “Using Signer and Packager” may be of some assistance in this regard. Typically customers should use the default targeting options specified by the signing sample.

Note: When preparing the production JFFS2 load, the coprocessor-side debugger daemon (zdaemon) will run only in a development environment where the OA daemon has been started, and the ownerIDs corresponding to segment 2 and 3 are 3 and 6, respectively. Additionally, zdaemon should only be incorporated into development JFFS2 images, and should never be included in any image signed with the signer utility intended for a production environment.

After creating a suitable application release package (JFFS2 image), the developer uses CRUSIGNR to create an EMBURN3 command (CLU file) that incorporates the application, IBM's segment 2 ownerID (243), the developer's ownerID, and the developer's private key, and the emergency signature for segment-3 returned by IBM as part of the enablement process.

Please refer to the scripts in *signing/cru/signer/tasks/emburn3* for an example of how to create an EMBURN3 file using the 64-bit version of Signer.

EMBURN3 CLU files are typically downloaded immediately following the ESTOWN3 command which establishes ownership of segment-3 with the IBM-approved owner ID. A user can also use CLU to download the file generated by this process to a coprocessor that contains another version of the application. The EMBURN3 command clears any state information the earlier version of the application has saved in nonvolatile memory. This use of an EMBURN3 command is rare. Typically customers will only load an EMBURN3 once, then update their application using a REMBURN3 command which preserves such information.[6]

Please refer to the scripts in *signing/signer/tasks/remburn3* for an example of how to create an REMBURN3 file using Signer.

[6] The public key downloaded with the earlier version of the application must be the public key in S3KDEVPU.KEY. A new public key can be assigned when the updated version of the application is downloaded (the new public key is taken from S3KDEVPP.KEY), but the new public key cannot be loaded using an EMBURN3 command until IBM provides a certificate for the new public key.

5 Overview of the development process

This chapter describes the entire process from initial preparation of the coprocessor to the creation of a file containing a developer application that can be shipped to the developer's customers or end users [1].

5.1 Preparing the development platform

After the Toolkit and all prerequisites (see "Prerequisites" on page 2) have been installed, the developer can prepare the coprocessor for use as a development platform. The specific procedure depends on whether or not software has already been installed in the coprocessor and, if so, what software has been installed.

The instructions in this section assume the directory that contains the various utilities shipped with the toolkit is in the search path for executable files (that is, the PATH environment variable includes *cctk/<version>/bin/host/<platform>* and the CCA installation directory), and that the various system files (Linux device driver) have been installed.

5.1.1 Determining coprocessor status

CLU's ST command can be used to determine what software, if any, is loaded in the coprocessor. (CLU is shipped as part of the CCA installation package.) For example:

```
<CLU> -c ST -l /logfile-directory/clu.log
```

See Figure 5 on page 43 for a typical response to the CLU ST command.

ROM Status lines

If the ROM Status lines in the CLU ST output do not indicate segment 1 is in the INITIALIZED state or if page 1 is not certified, the coprocessor cannot be used as a development platform without additional assistance from IBM.

Warning

Do not update segment 1 unless absolutely necessary. Segment 1 can only be updated a limited number of times before the coprocessor runs out of memory in which to store the certificate chain connecting the segment 1 public key to the original key installed at the factory. Users should update the system software in a coprocessor as seldom as possible. Note that the segment 1 CLU file needs to be loaded only once per adapter. Also, the segment 1 revision number does not necessarily need to match the segment 2/segment 3 revision numbers for the coprocessor to function properly. Segments 2 and 3 can be updated as many times as desired. Please contact your Toolkit provider if you have any questions about loading segment 1.

If the SEG2 ROM Status line indicates that coprocessor segment 2 is UNOWNED (Owner ID 0), continue with "Segments 2 and 3 UNOWNED" on page 42.

[1] The examples in this chapter assume the directory that contains the various utilities shipped with the toolkit is in the search path for executable files (that is, the PATH environment variable includes cctk/<version>/bin/host/linux).

If the owner identifier associated with segment 2 is 2, continue with “Segments 2 and 3 RUNNABLE with OWNER 2/2” on page 45.

If the owner identifier associated with segment 2 is 3, continue with “Segments 2 and 3 RUNNABLE with OWNER 3/6” on page 46.

If the owner identifier associated with segment 2 is 243, continue with “Segment 2 OWNER 243” on page 46.
If the owner identifier associated with segment 2 is neither 2 nor 3, it is not possible to use the coprocessor for development without the assistance of the owner of segment 2. To be able to use it for development, the owner of segment 2 must supply a CLU file to surrender that segment 2 ownership.

Note: Unowned seg-2 with owner ID 0 is covered above; in that case, continue with “Segments 2 and 3 UNOWNED” on page 42.

For other situations, review the rest of this section for details.

5.1.2 Loading the coprocessor

Depending on the results of the CLU ST command, the developer loads the various segments with the appropriate contents. See Using CLU on page 53 for a complete description of how to use CLU to perform the PL functions. In particular, if more than one coprocessor is installed, the developer must insert the coprocessor number into the CLU PL command to ensure that the software is loaded into the appropriate coprocessor.

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/<clufilename>.clu
```

where *coprocessornumber* refers to the adapter to be loaded (0 based) and *<clufilename>.clu* refers to the particular CLU file to be loaded onto the adapter. See the following discussion for a description of the conditions for loading each CLU file.

Once the development CLU files have been loaded onto the coprocessor, it is ready for further development activities.

Note: Loading the coprocessor requires CLU to obtain exclusive access to the adapter. If other applications, such as CNM or any of the Toolkit or web samples are using the adapter, CLU will return with an error message indicating the adapter is busy (typically this is 0x80400010). On Linux a convenient way to determine if anything is using the coprocessor is to use the loadable modules command:

```
lsmod | grep ibm4767
```

Look at the third column of output. For example,

```
ibm4767              181940  2
```

This shows that two processes are using the coprocessor. If no process is using the coprocessor, the output will be:

```
ibm4767              181940  0
```

Alternatively, use the list open files command:

```
      lsof | grep ibm4767
```

If an application such as CNM is using the adapter, you should see output similar to:

```
      java       16998       user  50u  CHR  244,0  0t0  40904 /dev/ibm4767/0
      DestroyJa 16998 16999 user  50u  CHR  244,0  0t0  40904 /dev/ibm4767/0
      Signal     16998 17000 user  50u  CHR  244,0  0t0  40904 /dev/ibm4767/0
      JIT        16998 17001 user  50u  CHR  244,0  0t0  40904 /dev/ibm4767/0
      …
      AWT-Event 16998 17077 user  50u  CHR  244,0  0t0  40904 /dev/ibm4767/0
```

If no application is using the adapter, those entries will not be displayed.

5.1.2.1 Segments 2 and 3 UNOWNED

If the "ROM Status" line indicates segments 2 and 3 are UNOWNED, the contents of segment 1 (as specified in the "Segment 1 Image" line) dictate how to proceed. Possible contents include:

- Coprocessor in factory-fresh state
- Segment 1 downlevel
- Segment 1 current

These states are discussed in the following sections.

Coprocessor in factory-fresh state

Load Segment 1. If software has never been loaded into the coprocessor (for example, if the coprocessor has just been removed from a factory-sealed package), the segment 1 image name will indicate this.

Figure 5 shows a typical response from a CLU ST command for a factory fresh 4767. Note the “Factory” text in the Segment 1 Image:

```
----------------------------------------------------------------------
              Coprocessor Load Utility (CLU) version 5.2.21
----------------------------------------------------------------------
Invocation :  ./csulclu /home/user/logs/4767.log st 0
Log File   :  /home/user/logs/4767.log
Started    :  Tue Mar  8 15:03:36 2016
----------------------------------------------------------------------
Vital Product Data
  Part Number        : 00LV498
  Secure Part Number : 00LV498
  EC Number          : 0N37015
  Serial Number      : DV5CX338
  Description        : IBM 4767-002 PCI-e Cryptographic Coprocessor
  Manufacturing Site : 91
  POST-0 Version     : 1
  POST-0 Release     : 35
  MiniBoot-0 Version : 1
  MiniBoot-0 Release : 33
```

```
ROM Status
  Page 1 Certified   : YES
  Segment-1 State    : INITIALIZED
  Segment-2 State    : UNOWNED
  Segment-2 Owner ID : 0
  Segment-3 State    : UNOWNED
  Segment-3 Owner ID : 0
Segment-1 Information
  Segment-1 Image    : 5.2.19   P0123 M0121 P0123 F0D01
201601041015502A00001100000000000
  --INFO-- This appears to be a factory fresh card.
  --INFO-- The segment-1 key swap CLU file
(reload_seg1_xipz_factory_to_prod_keyswap_<version>.clu).
  --INFO-- must be loaded before loading segments 2 & 3.
  Segment-1 Revision : 0
  Segment-1 Hash     : 47DE D8EE BB79 CF98 2250 DDBB 1CE9 45C4 6CAB 4243 BD11
E4B0 D742 664C 978C 1702 C201 EF4E 4C97 A21A 73D1 F227 BAFD B5FE 5125 421C
EEBC A9C3 4A12 7E32 645F 1588
----------------------------------------------------------------------
  Obtain Status ended successfully at Tue Mar  8 15:04:08 2016
----------------------------------------------------------------------
  Finished    : Tue Mar  8 15:04:08 2016
----------------------------------------------------------------------
```

Figure 5 Factory-fresh CLU status

Note that a factory-fresh card has 0x11 in bytes 21 and 22 of the Segment 1 image.

A factory-fresh card must be reloaded with the production keyswap image. This reload is a one-time action that is needed to replace the factory-fresh image with the production keyswap image. The developer updates the system software in segment 1 by loading *reload_seg1_xipz_ecc_factory_keyswap_r.j.m.clu* into the coprocessor, for example:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
$IBM4767_INSTALL_DIR/clu/reload_seg1_xipz_ecc_factory_to_prod_keyswap_r.j.m.c
lu
```

If this command fails, further assistance from IBM is required. Contact Crypto at crypto@us.ibm.com.

If this command succeeds, the developer proceeds to load *establish_ownership_seg2_toolkit_OID3_r.j.m.clu* as indicated in "Segment 1 Current" below.

A Segment 1 image that is no longer factory-fresh (has been keyswapped) will have 0x22 in bytes 21 and 22 instead of 0x11 for a factory-fresh card.

Segment 1 downlevel

Update Segment 1. If segment 1 contains a downlevel version or revision of CCA segment 1, the developer updates the system software in segment 1 by loading *reload_seg1_xipz_ecc_r.j.m.clu* into the coprocessor, for example:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
$IBM4767_INSTALL_DIR/clu/reload_seg1_xipz_ecc_r.j.m.clu
```

If this command succeeds, the developer proceeds to load *establish_ownership_seg2_toolkit_OID3_r.j.m.clu* as indicated in “Segment 1 current (development load)” below.

Warning

> Do not update segment 1 unless absolutely necessary. Segment 1 can only be updated a limited number of times before the coprocessor runs out of memory in which to store the certificate chain connecting the segment 1 public key to the original key installed at the factory. Users should update the system software in a coprocessor as seldom as possible. Note that the segment 1 CLU file needs to be loaded only once per adapter. Also, the segment 1 revision number does not necessarily need to match the segment 2/segment 3 revision numbers for the coprocessor to function properly. Segments 2 and 3 can be updated as many times as desired. Please contact your Toolkit provider if you have any questions about loading segment 1.

Segment 1 current (development load)

If segment 1 contains the appropriate version and revision of segment 1, the developer performs these steps to prepare the adapter as a development environment:

1. Establish ownership of segment 2 as ownerID 3 which is the segment-2 development owner ID.
2. Set the public key for segment 2 and load segment 2 with the coprocessor operating system corresponding to ownerID 3.
   Note: The coprocessor operating system is identical for development and production loads. It is just signed with different keys and owner IDs to differentiate development versus production usage. This can be verified by loading each segment-2 and noting the hash values for the segments. They will be identical.
3. Establish ownership of segment 3 as ownerID 6 which is the segment-3 development owner ID.
4. Set the public key for segment 3 and load segment 3 with the reverse-then-echo sample application.

Together, these four steps are referred to in this section as “Development Load.” Here is a detailed description of Development Load:

The developer loads a production version of the coprocessor operating system into segment 2 by loading *establish_ownership_seg2_toolkit_OID3_r.j.m.clu*, followed by loading *emergency_reload_seg2_toolkit_OID3_r.j.m.clu.* For example:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/establish_ownership_seg2_toolkit_OID3_r.j.m.clu
```

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/emergency_reload_seg2_toolkit_OID3_r.j.m.clu
```

The developer then sets the owner identifier for segment 3 by loading *establish_ownership_seg3_toolkit_OID6_r.j.m.clu*, followed by loading *emergency_reload_seg3_toolkit_OID6_r.j.m.clu*, for example:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/establish_ownership_seg3_toolkit_OID6_r.j.m.clu
```

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/emergency_reload_seg3_toolkit_OID6_r.j.m.clu
```

The last file sets the public key associated with segment 3 and loads the "reverse-then-echo" sample application. If desired, the developer can confirm the software has been properly loaded by:

1. resetting the coprocessor to start the "reverse-then-echo" application (see “How to reboot the ” on page 55),
2. compiling and linking the host reverse-then-echo driver if necessary (see “Compiling, assembling, and linking” on page 25), and
3. running the host driver, for example:

```
~/cctk/<version>/samples/toolkit/rte/host/<toolset>/sampleHostApp
adapternumber text
```

The driver sends the text string to the reverse-then-echo application on the coprocessor identified by *adapternumber*, which reverses it and returns it to the driver. The driver then prints the text received. For example:

```
samples/toolkit/rte/host/gcc/sampleHostApp 0 ‘Go Big Blue!’
```

would display

```
‘!eulB giB oG’
```

This completes preparation of the coprocessor for use as a development platform. Continue with “Compiling, assembling, and linking” on page 49.

5.1.2.2 Segments 2 and 3 RUNNABLE with OWNER 2/2

If the "ROM Status" lines indicate segment 2 and 3 are both RUNNABLE and both have ownerID 2, the coprocessor cannot be used as a development platform until the owner of segment 2 supplies a CLU file to surrender that ownership. In this case, IBM provides the file needed to surrender ownership.

The developer loads *surrender_ownership_seg2_xipz_r.j.m.clu* into the coprocessor, for example:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/surrender_ownership_seg2_xipz_r.j.m.clu
```

This file surrenders ownership of segments 2 and 3. If this command succeeds, segments 2 and 3 become UNOWNED. The developer proceeds with the steps in “Segment 1 current (development load)” on page 44.

If this command fails, further assistance from IBM is required. (The failure may indicate the public key associated with segment 2 has not been set to the expected value.)

5.1.2.3 Segments 2 and 3 RUNNABLE with OWNER 3/6

If the "ROM Status" lines indicate segment 2 and 3 are both RUNNABLE, segment 2 has ownerID 3, and segment 3 has ownerID 6, the coprocessor cannot be used as a development platform until the owner of segment 2 supplies a CLU file to surrender ownership. In this case, IBM provides the file needed to surrender ownership.

The developer loads *surrender_ownership_seg2_toolkit_OID3_r.j.m.clu* into the coprocessor, for example:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID3_r.j.m.clu
```

This file surrenders ownership of segment 2 as owner 3 and clears the contents of BBRAM. If this command succeeds, segments 2 and 3 become UNOWNED. If the developer wants to reload the coprocessor, the developer continues with the steps in “Segment 1 current (development load)” on page 44.

If this command fails, further assistance from IBM is required.

5.1.2.4 Segment 2 OWNER 243

If the "ROM Status" lines indicate the segment 2 ownerID is 243, the coprocessor cannot be used as a development platform until segment 2’s ownership is relinquished. To do this, the developer loads *surrender_ownership_seg2_toolkit_OID243_r.j.m.clu* into the coprocessor, for example:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID3_r.j.m.clu
```

This file surrenders ownership of segment 2 as owner 243. Then, if the developer wants to reset and reload the coprocessor, the developer continues with the steps in “Segment 1 current (development load)” on page 44.

5.1.2.5 Segment 2 OWNED_BUT_UNRELIABLE

If the "ROM Status" lines indicate segment 2 is OWNED_BUT_UNRELIABLE, the coprocessor cannot be used as a development platform without additional assistance from the owner of segment 2.

If the segment 2 ownerID is 2, the developer loads *emergency_reload_seg2_xipz_r.j.m.clu* followed by *surrender_ownership_seg2_xipz_r.j.m.clu* into the coprocessor, for example:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/emergency_reload_seg2_xipz_r.j.m.clu
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/surrender_ownership_seg2_xipz_r.j.m.clu
```

These files reload segment 2 and then surrender ownership of segment 2 as owner 2. If the commands succeed, segments 2 and 3 become UNOWNED and the developer proceeds with the steps in “Segment 1 current (development load)” on page 44.

If the segment 2 ownerID is 3, the developer loads *emergency_reload_seg2_toolkit_OID3_r.j.m.clu* followed by *surrender_ownership_seg2_toolkit_OID3_r.j.m.clu* into the coprocessor, for example:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/emergency_reload_seg2_toolkit_OID3_r.j.m.clu
```

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID3_r.j.m.clu
```

These files reload segment 2 and then surrender ownership of segment 2 as owner 3. If the commands succeed, segments 2 and 3 become UNOWNED and the developer proceeds with the steps in “Segment 1 current (development load)” on page 44.

If the segment 2 ownerID is 243, the developer loads *emergency_reload_seg2_toolkit_OID243_r.j.m.clu* followed by *surrender_ownership_seg2_toolkit_OID243_r.j.m.clu* into the coprocessor, for example:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/emergency_reload_seg2_toolkit_OID243_r.j.m.clu
```

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID243_r.j.m.clu
```

These files reload segment 2 and then surrender ownership of segment 2 as owner 243. If the command succeeds, segments 2 and 3 become UNOWNED and the developer proceeds with the steps in “Segment 1 current (development load)” on page 44.

5.1.2.6 Segment 2 RUNNABLE, segment 3 OWNED BUT UNRELIABLE

If the "ROM Status" lines indicate segment 2 is RUNNABLE but segment 3 is OWNED_BUT_UNRELIABLE or RELIABLE_BUT_UNRUNNABLE, the coprocessor cannot be used as a development platform without additional assistance from the owner of segment 2 or segment 3.

If the segment 2 ownerID is 2, the developer loads *surrender_ownership_seg2_xipz_r.j.m.clu* into the coprocessor, for example:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/surrender_ownership_seg2_xipz_r.j.m.clu
```

This file surrenders ownership of segment 2. If the command succeeds, segments 2 and 3 become UNOWNED and the developer proceeds with the steps in “Segment 1 current (development load)” on page 44.

If the segment 2 ownerID is 3, the developer loads *surrender_ownership_seg2_toolkit_OID3_r.j.m.clu* into the coprocessor, for example:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID3_r.j.m.clu
```

This file surrenders ownership of segment 2. If the command succeeds, segments 2 and 3 become UNOWNED and the developer proceeds with the steps in “Segment 1 current (development load)” on page 44.

If the segment 2 ownerID is 243, the developer loads *surrender_ownership_seg2_toolkit_OID243_r.j.m.clu* into the coprocessor, for example:

```
<CLU> -l /logfile-directory/clu.log  -c PL [-a coprocessornumber] -d
~/cctk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID243_r.j.m.clu
```

This file surrenders ownership of segment 2. If the command succeeds, segments 2 and 3 become UNOWNED and the developer proceeds with the steps in “Segment 1 current (development load)” on page 44.

5.1.2.7 Development preparation summary

Figure 6 illustrates the steps involved in preparing a coprocessor for use as a development platform.

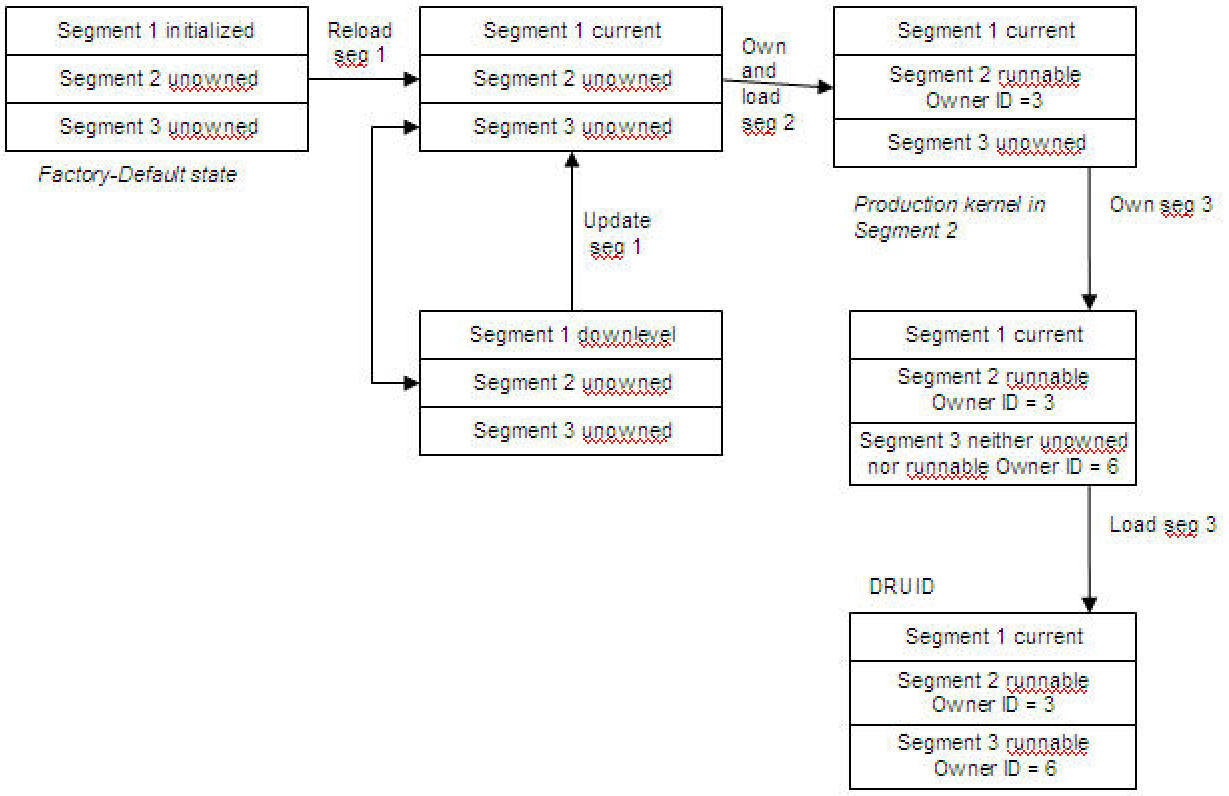


Figure 6 Development preparation process

- Reload seg 1 means to load *reload_seg1_xipz_ecc_factory_to_prod_keyswap_r.j.m.clu*, which loads IBM’s system software into segment 1.
- Update seg 1 means to load *reload_seg1_xipz_ecc_r.j.m.clu,* which updates IBM’s system software in segment 1 of the coprocessor.

- Own seg 2 means load *establish_ownership_seg2_toolkit_OID3_r.j.m.clu*, which establishes ownership of segment 2.
- Load seg 2 means load *emergency_reload_seg2_toolkit_OID3_r.j.m.clu*, which loads the embedded OS into the coprocessor.

- Own seg 3 means to load *establish_ownership_seg3_toolkit_OID6_r.j.m.clu*, which establishes ownership of segment 3.
- Load seg 3 means to load *emergency_reload_seg3_toolkit_OID6_r.j.m.clu*, which loads the JFFS2 image onto segment 3 of the coprocessor.

5.1.2.8 Compiling, assembling, and linking

Compile and link the application under development. Specify the appropriate options to ensure debugging information is incorporated into the executable file [2]. Refer to “Compiling, assembling, and linking” on page 25. For most host and card side makefiles, adding *DEBUG=y* to the makefile invocation will trigger the addition of the appropriate compiler / linker flags for debugging.

5.1.2.9 Building JFFS2 filesystem images

Refer to “Building JFFS2 filesystem Images” on page 29. Note: Should you wish to incorporate the ICAT card-side debugger daemon into your image for debugging, specify *BUILD_TYPE=debug* for debug builds.

5.1.2.10 Downloading and debugging

1. If desired, clear any state the application saved in nonvolatile memory during previous debug sessions:

   ```
   <CLU> -l /logfile-directory/clu.log -c PL -d
   ~/cctk/clufiles/emergency_reload_seg3_toolkit_OID6_r.j.m.clu
   ```

   Note: This is rarely necessary.

2. Download the JFFS2 image onto the coprocessor:

   ```
   druid [-a adapternumber] -l log_file -d <segment-3 JFFS2 Image>
   ```

3. Wait for the coprocessor to reboot and start the application. Note: It is useful to watch /var/log/messages and look for messages coming from the ibm4767 device driver to verify the application has started:

   ```
   sudo tail -f /var/log/messages
   ```

4. Set all appropriate ICAT-related environment variables. Note: When building samples with *build_seg3_image/cctk.seg3.image.mak*, the appropriate environment variables will be written to a *seticat_<sample name>.sh* shell script. This script assumes adapter 0 is the desired adapter. Invoke this script as *source seticat_<sample name>.sh* to export the environment variables in the current terminal session), then start the debugger and attach to the application:

   ```
   icatpzx
   ```

   Refer to the *IBM 4767 PCIe Cryptographic Coprocessor Interactive Code Analysis Tool (ICAT) Getting Started* manual for more information.

[2] The developer is free to choose the executable file name.

If changes to the application prove necessary, make them and then continue with "Compiling, assembling, and linking" on page 49.

5.1.2.11 Testing a coprocessor application in a production environment

At some point it will be necessary to test the application in a production environment. To do so, remove the debugger and any debugging code from the application and then rebuild the application by compiling, assembling, linking, and building a JFFS2 image. Do not specify the options that incorporate debugging information in the executable.

1. Load a production-level copy of the mkfs image into the coprocessor (using a CLU file or DRUID). Sign the JFFS2 image using keys generated by development along with certificates and other files provided by IBM.

   Note: This requires CCA to be loaded in the adapter.

2. Prepare the adapter for the production environment by surrendering ownership of segment 2.

3. Load the production CLU files:

   - *establish_ownership_seg2_toolkit_OID243_r.j.m.clu*
   - *emergency_reload_seg2_toolkit_OID243_r.j.m.clu*
   - *reload_seg2_toolkit_OID243_r.j.m.clu*
   - *ESTOWN3.clu* (file provided by IBM as part of the signing / enablement process)
   - Your segment 3 CLU file for *EMBURN3* (file name depends on you)

4. Wait for the coprocessor to reboot and start the application. Reminder: *sudo tail -f /var/log/messages* is useful here.

5. Test the application.

6. If changes to the application prove necessary, make them and then re-sign and reload the application into the adapter.

Development process example

This section provides a sample set of steps needed to compile, link, load, and run the reverse-then-echo (RTE) sample provided in *cctk/<version>/samples/toolkit/rte*. This sample assumes that the Linux development environment has been set up as described in "Installation and setup" on page 9 and "Loading the coprocessor" on page 41.

Follow these steps. Refer back to the previous sections in this chapter for details. These steps are assumed to be invoked from a command line terminal (such as an xterm on Linux or a command shell on Windows) where applicable.

Customers may find that it is preferable to open multiple terminal sessions: one for card-side builds, one for host-side builds, and one for JFFS2 image builds. This is not required, but it provides logical separation of duties and may help clarify the host ↔ card interactions and boundaries.

Run the following commands from a terminal window.

1. Set environment variables for host-side builds.
   For example, on Linux:
   ```
   export CCTK_FS_ROOT=/home/user/cctk/<version>
   ```

   For example, on Windows:
   ```
   set CCTK_FS_ROOT=/home/user/cctk/<version>
   ```

2. Set environment variables for card-side builds on Linux in addition to the ones for host-side builds. For example:
   ```
   export CROSS=/home/user/<generic reference to your cross compiler base>
   export GCC_NAME=ppc476-ibm-linux-gnu-<generic reference to your cross
   compiler GCC_NAME>
   ```

3. Set environment variables for JFFS2 image builds on Linux:
   ```
   export CCTK_JFFS2_DIR=/usr/sbin (this may differ in your installation)
   ```

   Note: It is assumed that the cross-compiler is built and installed following instructions from your toolkit provider or through the scripts found in the cross_compiler_scripts directory.

4. Make the host side application:
   Linux:
   ```
   cd <CCTK_FS_ROOT>/samples/toolkit/rte/host/gcc
   make –f host.mak
   ```

   Windows:
   ```
   cd <CCTK_FS_ROOT>\samples\toolkit\rte\host\msvc
   nmake –f host.mak
   ```

   Note: before running nmake, you must run the appropriate bat file for your Microsoft Visual Studio installation. IBM provides standalone makefiles that need to be run from a command prompt. You can use the Microsoft Visual Studio vcvars64.bat file, which is typically installed in "C:\Program Files (x86)\Microsoft Visual Studio <version>\VC\bin\amd64\vcvars64.bat" with the IBM makefiles.

5. Make the coprocessor side (Linux only):
   ```
   cd <CCTK_FS_ROOT>/samples/toolkit/rte/card/gcc
   make –f card.mak
   ```

6. Build the JFFS2 image (Linux only):
   ```
   cd <CCTK_FS_ROOT>/build_seg3_image
   make –f cctk.seg3.image.mak SAMPLE_NAME=rte
   ```

7. Load the image onto the adapter:
   Linux:
   ```
   druid -a 0 -l sample.log -d cctk.rte.prod.<timestamp>.bin
   ```

   Windows:
   ```
   druid.exe -a 0 -l sample.log -d cctk.rte.prod.<timestamp>.bin
   ```

8. Wait for the adapter to initialize (3 to 5 minutes).

Note: On Linux, this command:

```
sudo tail -f /var/log/messages
```

will display useful messages from the adapter, including a message that the application has started.

9. Run the host-side application. On Linux:

```
cd <CCTK_FS_ROOT>
samples/toolkit/rte/host/gcc/sampleHostApp 0 ‘Go Big Blue!’
```

On Windows:

```
cd <CCTK_FS_ROOT>
samples/toolkit/rte/host/gcc/sampleHostApp.exe 0 ‘Go Big Blue!’
```

The RTE sample runs and returns the text ‘!eulB giB oG’.

6 Using CLU

The Coprocessor Load Utility (CLU) interacts with the coprocessor's ROM-based system software to update software in flash. The Coprocessor Load Utility can also obtain information about the coprocessor, reset the coprocessor, or validate the software in the coprocessor.

Note: CLU is always provided with the CCA Support Program and is not a direct part of the Toolkit. However, it is needed to load the coprocessor, and as such is documented here for convenience. Official CLU documentation can be found in the *CCA Installation Manual*, available on the Library page.

6.1 Syntax

CLU -c command [-l log_file] [-a adapter_number] **`[-d data_file] [-v]`**

where command is one of the following:

EP	Load EP11 Images	Load CLU stripped images for EP11 only.
PL	Program load	Load firmware onto the specified adapter.
RS	Reset adapter	Reset the specified adapter.
SS	System status	Check status of all adapters.
ST	Status	Check status of specified adapter.
VA	Validate Adapter	Validate the specified adapter's certificate chain.
VF	Verify File	Verify the specified CLU file was signed by IBM.
HC	CCA Load from HUL	Load CCA onto card with UNOWNED Seg 2
RC	CCA Reload from HUL	Reload CCA onto card with CCA currently loaded
HE	EP11 Load From HUL	Load EP11 onto card with UNOWNED Seg2
RE	EP11 Reload from HUL	Reload EP11 onto card with EP11 currently loaded

If no log_file is specified, CLU will read the serial number of the adapter by the adapter_number parameter (0 by default) and create a log file named <serial number>.log.

The data_file option is valid only with the following commands:

PL	The data file provides the signed image to load.
VA	The data file provides the adapter validation CLU file.
VF	The data file provides the CLU file to verify.
HC/RC/HE/RE	The data file specifies the HUL file to use.

-v	Verbose output	Enables extended output on certain commands.

Note: Adapter numbers are zero-based, so the first adapter in the system is adapter 0.

CLU -h Help	Invokes this help menu.

Deprecated CLU Invocation:

CLU log_file command [adapter_number] [data_file] [-Q]
    -Q Ignored for backward compatibility.

6.2 Return codes

When the utility finishes processing, it returns a value that can be tested in a script file or in a command file. The returned values are:

Value	Description
0	OK.
Nonzero	Command failed. Check the log file for more information.

7 How to reboot the coprocessor

An IBM 4767 can be rebooted in any of several ways:

1. Using CLU's RS command, for example:[1]

```
<CLU> -c RS -l /logfile-directory/clu.log
```

2. By stopping the device driver and restarting it. This has the additional benefit of resynchronizing the device driver. On Linux, this can be accomplished by:

   a. Physically unloading the driver (as root or via sudo):

      ```
      <full path to>/cctk/<version>/bin/driver/ibm4767_unload
      ```

   b. Subsequently reloading the driver (as root or via sudo):

      ```
      <full path to>/cctk/<version>/bin/driver/ibm4767_load
      ```

   On Windows, this can be accomplished by using the Windows Device Manager.

   When a request is issued from the host to the coprocessor, the device driver keeps track of how long it takes the coprocessor to reply to the request. During normal operations, this is not an issue. However, when debugging an application using ICAT, this default timeout will be inadequate. As a result, if the timeout value is exceeded, the device driver will forcibly reboot the coprocessor, thus terminating the application while it is being debugged. To avoid this situation, the default timeout can be extended by following the procedure in “How to change the host device driver timeout”.

3. The coprocessor reboots at the conclusion of a CLU command or after DRUID downloads an application.

7.1 How to change the host device driver timeout

7.1.1 Linux

On some Linux distros, the file */etc/modprobe.d/ibm4767.conf* must be amended to specify the timeout options passed to the host device driver.

Note: To see the current value of the device driver timeout values, issue the following command:

```
cat /proc/driver/ibm4767/timeout
```

Before making any changes to /etc/modprobe.d/ibm4767.conf, MAKE A BACKUP COPY OF THIS FILE AND STORE IT IN A SAFE LOCATION!

[1] The examples in this chapter assume the directory that contains the various utilities shipped with the toolkit is in the search path for executable files (that is, the PATH environment variable includes cctk/bin/host/linux).

Then, as root, edit */etc/modprobe.d/ibm4767.conf* to indicate that the “install ibm4767” line specifies the desired timeout values. Initially, this line should look like:

```
install ibm4767 /sbin/modprobe --ignore-install ibm4767  $CMDLINE_OPTS;
  /sbin/ibm4767_mknod
```

Edit this line to specify the desired timeout values in seconds. For example, if you want the timeout value set to 1200 seconds, specify the following line in */etc/modprobe.d/ibm4767.conf*:

```
install ibm4767 /sbin/modprobe --ignore-install ibm4767
  timeout_mcpu=1200,1200  $CMDLINE_OPTS; /sbin/ibm4767_mknod
```

Once this file has been modified to reflect the desired timeout, you need to unload and reload the device driver (as root or via sudo). This can be done with the ibm4767_unload and ibm4767_load scripts in the toolkit.

To verify the timeout value has been changed, issue this command:

```
cat /proc/driver/ibm4767/timeout
```

7.1.2 Windows

On Windows, you can change the host device driver temporarily (until the next system reboot) or permanently. You will need administrator privileges to make these changes.

To see the current device driver timeout values, navigate to the directory where the host software is installed (typically, this is “*C:\Program Files\IBM\4767*”), and then to the utils directory. Issue the following command:

```
IBM4767_driver_util.exe -status
```

The timeouts for MCPU windows 0 and 1 will be displayed near the bottom of the status listing.

To change a timeout temporarily (until the next system reboot), issue the following command:

```
IBM4767_driver_util.exe -to <window> <timeout>
where:
window = the window to be changed (0 or 1)
timeout = the timeout in milliseconds (e.g., 1200)
```

Confirm that the timeout values have been changed by reviewing the status again.

To change the timeout values permanently, edit the registry. Using *regedit*, navigate to the registry value for *Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\ibm4767*. Find the entries for *TimeoutMCPU0* and *TimeoutMCPU1*. Change those entries to the time values you prefer. For example, you may want to change both values to 1200 (decimal).

After making the registry changes, reboot the server to make the timeout changes permanent. After rebooting, you can use *regedit* to make sure the timeout changes were done correctly.

8 Using Signer and Packager

This chapter describes the use of the Signer and Packager utilities and explains why the design of the coprocessor makes these utilities necessary. Note: Signer and Packager use CCA Passphrase profiles and do not support Passphrase2 profiles.

8.1 Coprocessor memory segments and security

The design for the coprocessor was motivated by the need to satisfy simultaneously the following requirements:

1. Code must not be loaded into the coprocessor unless IBM or an agent IBM trusts has authorized the operation.
2. Once loaded into the coprocessor, code must not run or accumulate state unless the environment in which it runs is trustworthy.
3. Agents outside the coprocessor that interact with code running on the coprocessor must be able to verify that the code is legitimate and that the coprocessor is authentic and has not been tampered with.
4. Shipment and configuration of coprocessors, and maintenance on and upgrades to code inside a coprocessor, must not require trusted couriers or security officers.
5. IBM must not need to examine a developer's code or have any knowledge of a developer's private cryptographic keys in order to make it possible for customers to load the developer's code into a coprocessor and run it. [1]

Toward these ends, the design defines four "segments":

- Segment 0 is ROM and contains one portion of Miniboot. Miniboot is the most privileged software in the coprocessor and among other things implements the protocols described in this section.
- Segment 1 is flash and contains the other portion of Miniboot. The division of Miniboot into a ROM portion and a flash portion preserves flexibility (the flash portion can be changed if necessary) while guaranteeing a basic level of security (implemented in the ROM portion).
- Segment 2 is flash and usually contains the coprocessor operating system.
- Segment 3 is flash and usually contains one or more coprocessor applications.

Segment 0 obviously cannot be changed. Segment 1 can be changed, but should this prove necessary, IBM will provide a file that can be downloaded using CLU to effect the change. A developer need not create commands that affect segment 1. Unless a developer wants to modify the Linux operating system, they need not create commands that affect segment 2. Therefore, the remainder of this chapter deals with changes to segment 3.

There are seven pieces of information associated with each segment:

1. The identity of the owner of the segment, that is, the party responsible for the software that is to be loaded into the segment. Owner identifiers are two bytes long.[2] IBM owns segment 1 and issues an owner identifier to any party that is developing code to be loaded into segment 2. An

[1] Notice in particular that neither the EMBURN3 nor the REMBURN3 command requires IBM to have a copy of the code in segment 3 or the private key corresponding to the public key associated with segment 3.

[2] An owner identifier of all zeros is reserved and means "no owner". A developer's owner identifier is not necessarily the same as the "Developer Identifier" the developer uses when registering coprocessor applications as described in the *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference*.

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).
2. The public key for the owner of the segment.
3. The contents of the segment (that is, the operating system or coprocessor application).
4. Data stored in nonvolatile memory by the code in the segment.
5. The name of the segment (for example, the name of the coprocessor application).
6. The revision level of the contents of the segment (for example, the version number of the coprocessor application).
7. A flag indicating whether or not data stored in BBRAM by the code in the segment is to be cleared if the contents of a more privileged segment change.

Segment 2 and segment 3 can be in one of the following states, depending on how much of the information associated with the segment has been verified:

- UNOWNED - None of the information associated with the segment has been set (that is, it is all unreliable).
- OWNED_BUT_UNRELIABLE - The segment has an owner but the rest of the information associated with the segment is unreliable.
- RELIABLE_BUT_UNRUNNABLE - All of the information associated with the segment is reliable, but the code in the segment should not be allowed to run.
- RUNNABLE - All of the information associated with the segment is reliable, and the code in the segment may be allowed to run.

Miniboot enforces the following rules:

- If segment 2's state changes to UNOWNED for any reason, segment 3's state is also changed to UNOWNED.
- If segment 2's state is not RUNNABLE, segment 3's state cannot be RUNNABLE. If segment 2's state changes from RUNNABLE to OWNED_BUT_UNRELIABLE or to RELIABLE_BUT_UNRUNNABLE, segment 3's state is changed to RELIABLE_BUT_UNRUNNABLE. If segment 2's state changes from RUNNABLE to UNOWNED, segment 3's state is also changed to UNOWNED in accordance with the first rule.
- If a segment is not RUNNABLE, the areas of BBRAM controlled by the segment are cleared (that is, any information an application in the segment may have saved in BBRAM is lost).

The rules can be expressed in this manner:

1. A segment cannot be owned if its parent segment is not owned, and
2. A segment cannot be RUNNABLE if its parent is not RUNNABLE.

If the coprocessor's tamper-detection circuitry detects an attempt to compromise the physical security of the coprocessor, all data in BBRAM is cleared and Miniboot changes segment 2's state to UNOWNED. Certain unusual errors affecting segment 1 or segment 2 can also cause segment 2's state to change to UNOWNED, OWNED_BUT_UNRELIABLE, or RELIABLE_BUT_UNRUNNABLE.

Miniboot will not transfer control to segment 2 after the coprocessor is rebooted unless segment 2's state is RUNNABLE. The code in segment 2 should not transfer control to an application in segment 3 unless segment 3's state is RUNNABLE.[3]

[3] Segment 3's state is maintained in BBRAM.

Miniboot changes the state of a segment in response to certain commands Miniboot receives from the host. Figure 8 on page 60 shows the state transitions for segment 2, and Figure 9 on page 61 shows the state transitions for segment 3.

A file that is downloaded to the coprocessor using CLU essentially contains one or more of the pieces of information associated with a segment and one or more Miniboot commands. The Signer utility generates a file containing a single Miniboot command and the corresponding segment information and digitally signs it so CLU can verify the command was produced by an authorized agent.

The Packager utility combines signed commands into a single file so that a single download can perform several Miniboot commands. A developer who makes a change to an application during development must use the Signer and Packager utilities to create a file that contains the revised application and the necessary commands to load it into segment 3 [4] and make that segment RUNNABLE. This may entail replacing an existing copy of the application or loading the application into an empty segment. In like manner, prior to shipment of the completed application, one or more files must be created to allow the end user to load the application and run it no matter what state segment 3 is in to begin with.

8.1.1 State transitions

This section describes the various states for each of the coprocessor's loadable segments (segments 1, 2, and 3). It also shows the CLU commands used to change each segment's state. The actions that can be done include preparing the segments for code loading, loading code, and removing code. In the following diagrams, these colors are used to distinguish the types of state transitions:

Blue	Prepares the coprocessor for code with the specified ownerID
Green	Loads code on the coprocessor
Red	Deletes code from the coprocessor

8.1.1.1 Segment 1 state transitions

Segment 1's state changes rarely. When a new coprocessor is purchased, segment 1 is in a factory-fresh state. The user loads segment 1 with the factory to prod keyswap file. This file can be loaded only once. Then, the user reloads segment 1 with the current file version. Typically, this file must be loaded only once.

From this point on, the only time segment 1 needs to be reloaded is if IBM releases an update to segment 1. Typically, IBM releases segment 1 updates less than once per year.

[4] Or segment 2 if the developer is writing an operating system for the coprocessor.

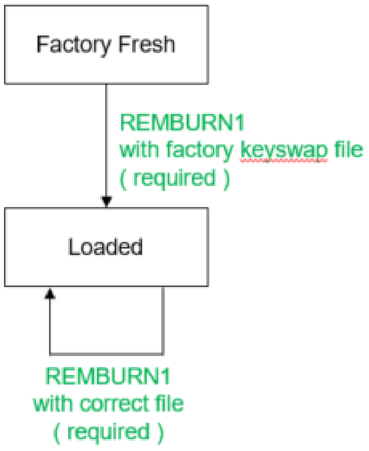


Figure 7 State transitions for segment 1

8.1.1.2 Segment 2 state transitions

The segment 2 state can change when switching from development to production or when you want to clear the card by surrendering ownership of segment 2.

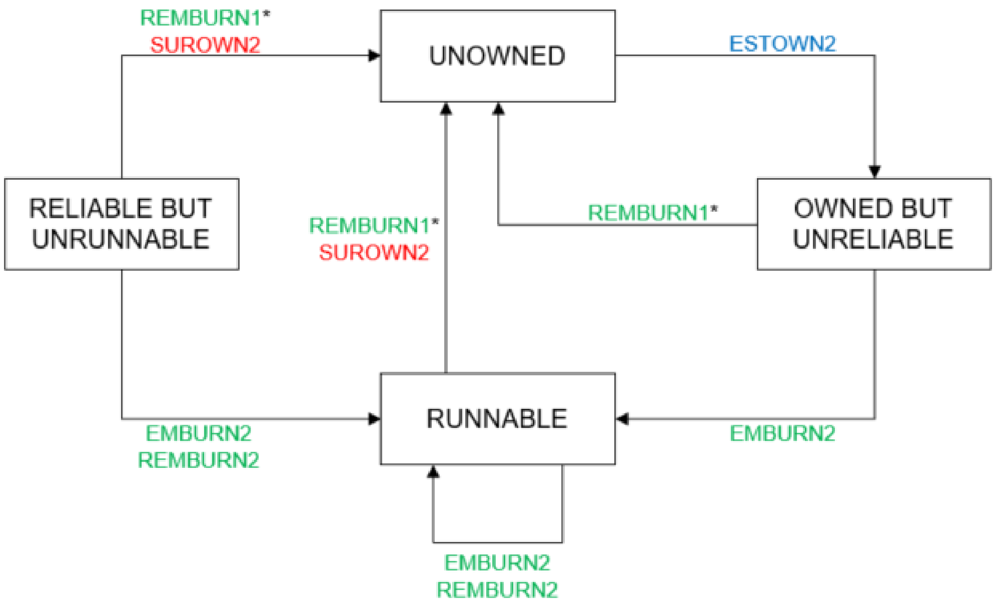


Figure 8 State transitions for segment 2

8.1.1.3 Segment 3 state transitions

Segment 3's state changes whenever new code is loaded into the coprocessor, for example, when you load CCA or the Toolkit into segment 3.

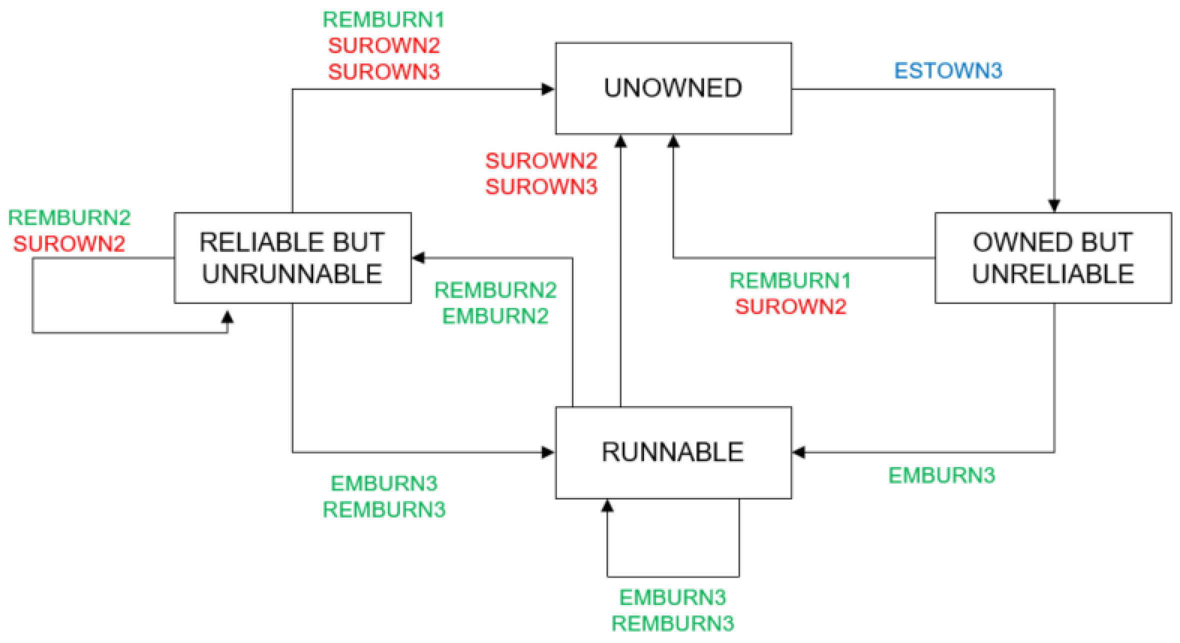


Figure 9 State transitions for segment 3

8.2 Signing station security considerations

Special consideration should be taken to ensure the security of the signing station. This includes access to the signing station per your company's security policy. Examples of security considerations include, but are not limited to:

- using appropriate password security,
- ensuring timely installation of patches and security fixes,
- separating administrator accounts and operator accounts,
- breaking the master keys into parts and distributing the parts to at least two separate individuals,
- ensuring that the master key parts are loaded into the crypto card by those individuals each time signing is performed and then are removed immediately from the card after signing is performed,
- keeping the keys used in the signing process on portable storage such as USB memory sticks,
- storing the portable storage containing the signing keys in locked cabinets and using them only during the signing process,
- using appropriate multi-party authentication and separation of duties,
- restricting physical access to the facility where the signing station is housed,
- further limiting physical access to the signing station itself, and
- isolating the signing station from networks.

IBM advises precautions such as these because poor security regarding the signing station and signing keys can negate the tremendous amount of security provided by the IBM adapter and the cryptographic components that protect your solution.

Consult with your security architect to ensure that you have the appropriate security guidelines in place. Please contact IBM if you have questions.

8.3 Disaster recovery

It is extremely important to design and implement a viable disaster recovery plan. This plan should include, but should not be limited to, plans for recovery from:

- physical loss of the signing station due to fire, water, or other physical disaster,
- inadvertent loss of master keys and other key data, and
- tamper, intrusion, or compromise of the signing station, which could include a rogue's actions on the signing station itself.

Disaster recovery plans should provide for timely complete backups of the signing station. These backups should be kept in an off-site storage facility. The master keys should be kept in secure locations far enough away from the signing station's physical location so that disaster does not affect both. Additional considerations should be included based on your organization's security and general disaster recovery policy.

Consult with your security architect to ensure that you have the appropriate disaster recovery plans in place. Please contact IBM if you have questions.

8.4 Signer and Packager utilities

These utilities, which are provided for Linux, provide the ability to sign and package a card-side image using the development test keys associated with segment-2 owner ID 2 and segment-3 owner ID 6. The signed files created by the sample Signer and Packager Toolkit files are intended ONLY as an example of how to create signed images that can be loaded via CLU into a development environment.

Signer and Packager use parameter=value pairs for inputs read from a configuration file specified when invoking the utility. For example, to specify the REMBURN3 segment-2 and segment-3 owner ID parameters for a sample REMBURN3 command, the following lines can be used:

```
REMBURN3_SEGMENT2_OWNERID=3
REMBURN3_SEGMENT3_OWNERID=6
```

Additionally, comments are allowed in the parameter files to allow developers room to document their procedures. Comments start with a # and continue to the end of the line.

For examples of how to use Signer and Packager, please review the signing directory in the Toolkit. Scripts to process key generation and signing requests are provided as examples. Please refer to the internal documentation contained in these scripts for more information.

Under no circumstances should files signed with the keys provided with this sample be used in a production environment.

8.4.1 Initializing CCA for use with Signer

Before running Signer / Packager, CCA or a UDX must be loaded on the coprocessor and the coprocessor must be properly initialized as a signing station. For development and testing purposes, the cca_test_init program provided with CCA can be used to initialize the coprocessor as a signing station. For production signing, additional configuration of the coprocessor may be required and will depend on your specific requirements. Traditionally, this would include loading a specific master key and incorporating more restrictive CCA roles and profiles than the ones initialized by cca_test_init.

The CCA role used when signing must enable certain operations. See CCA roles for Signer and Packager on page 92 for additional information.

8.4.2 Roles and Profiles

Before using Signer, you will need to define a set of roles and associated profiles that provide the proper framework for toolkit development and signing. In general, it is a good idea for each role to contain only the access control points they need and no more.

This section defines a sample set of roles and profiles. IBM cannot endorse a specific setup, but can provide this sample set as a starting point. Using your organization's security policy, make changes that are appropriate for your specific environment.

The term “SO” stands for Security Officer. This example identifies four SOs. Using multiple SOs helps ensure appropriate separation of duties so that one individual does not have too much power.

Role	Corresponding Profile	Description
DEFAULT	<NONE>	Mandatory CCA default role. This role should allow only for minimal functionality. This role is not associated with any profile.
mkadm1	mkadm1	Role/Profile to be used by **SO#1** and **SO#3** to: Load the first and third master key parts.
mkadm2	mkadm2	Role/Profile to be used by **SO#2** to: 1. Load the second and last master key parts. 2. Set the master key.
admin	admin	Perform administrative tasks such as changing a user's password, setting the adapter's clock, etc.
keygen	keygen	Generate initial keys for the coprocessor.
crutasks	crutasks	Role/Profile to be used for signing.

Please note that none of the administrators can load and set the master key on their own.

8.4.2.1 *mkadm1* role access control points and setup

The *mkadm1* role is used by **SO#1** to load the first master key part and used by **SO#3** to load the third master key part.

The following access control points (ACP's) must be **enabled** in the *mkadm1* role. All other ACP's must be **restricted**.

***mkadm1* Access Control Points**

ACP	Description in CNM	Justification
x'001D'	Compute Verification Pattern	Necessary to compute MKVP of any keys stored in key storage.
x'0107'	One-Way Hash, SHA-1	Required to log on to a profile.
x'0116'	Read Public Access Control Information	Required by CNM to read public access control information such as the allowed ACP lists for various roles.
x'0320'	Load First APKA Master Key Part	Required to load the first APKA master key part.
x'0321'	Combine Intermediate APKA Master Key Parts	Required to combine intermediate APKA master key parts.

SO#1 creates and loads the *mkadm1* role using the following steps:

1. From CNM's main menu, select *Access Control → Roles → New*.
2. For *Role ID,* enter *mkadm1.*
3. For *Valid Days,* make sure all days of the week are checked.
4. For *Valid Times in GMT (Start-End),* enter *00:00* to *23:59.*
5. For each ACP in the mkadm1 *Access Control Points* table listed above, search through the list of *Restricted Operations* to find the corresponding ACP. Select the ACP then click *Permit*.
6. Click *Load,* then click *OK*, and then click *Cancel* to return to CNM's main menu.

SO#2 must verify that the *mkadm1* role contains only the ACP's listed above.

8.4.2.2 *mkadm2* role access control points and setup

The *mkadm2* role is used by **SO#2** to load the second master key part, and used by **SO#4** to load the fourth (last) master key part and set the master key.

The following access control points (ACP's) must be **enabled** in the *mkadm2* role. All other ACP's must be **restricted**.

***mkadm2* Access Control Points**

ACP	Description in CNM	Justification
x'001D'	Compute Verification Pattern	Necessary to compute MKVP of any keys stored in key storage.
x'0107'	One-Way Hash, SHA-1	Required to log on in CNM.
x'0116'	Read Public Access Control Information	Required by CNM to read public access control information such as the allowed ACP lists for various roles.
x'0119'	Load Function-Control Vector	Required to load a function control vector.
x'011A'	Clear Function-Control Vector	Required to clear a function control vector.
x'0321'	Combine Intermediate APKA Master Key Parts	Required to combine intermediate APKA Master Key parts.
x'0322'	Activate New APKA Master Key (SET)	Required to set the APKA Master Key.

SO#2 should create the *mkadm2* role using the following steps

1. From CNM's main menu, select *Access Control* → *Roles* → *New*.
2. For *Role ID,* enter *mkadm2.*
3. For *Valid Days,* make sure all days of the week are checked.
4. For *Valid Times in GMT (Start-End),* enter *00:00* to *23:59.*
5. For each ACP in the *mkadm2 Access Control Points* table listed above, search through the list of *Restricted Operations* to find the corresponding ACP. Select the ACP then click *Permit*.
6. Click *Load,* then click *OK*, and then click *Cancel* to return to CNM's main menu.

SO#1 must verify that the *mkadm2* role contains only the ACP's listed above.

8.4.2.3 *admin* role setup

A separate administrative role needs to be created that allows an administrator to perform various administrative-related tasks on the adapter.

The following access control points (ACP's) must be **enabled** in the *admin* role. All other ACP's must be **restricted**.

***admin* Access Control Points**

x'0107'	One-Way Hash, SHA-1	Needed to compute hash for log on.
x'010F'	Reset Intrusion Latch	Needed to reset the intrusion latch.
x'0110'	Set Clock	Required for administrative tasks.
x'0111'	Reinitialize Device	Required for administrative tasks.
x'0112'	Initialize Access Control System	Required by CNM for almost all access control related tasks.
x'0113'	Change User Profile Expiration Date	Required by CNM to change a profile expiration date. This would fall under the category of a normal administrative task.
x'0114'	Change User Profile Authentication Data	Required by CNM to change the user profile password. This would fall under the category of a normal administrative task.
x'0115'	Reset User Profile Logon-Attempt-Failure Count	Required by CNM to reset the logon failure count if a user forgets his password and attempts to log on too many times. This would fall under the category of a normal administrative task.
x'0116'	Read Public Access Control Information	Required by CNM to read public access control information such as the allowed ACP lists for various roles.
x'0117'	Delete user Profile	Required to delete a user profile. This would fall under the category of a normal administrative task.
x'0118'	Delete Role	Required to delete a role. This would fall under the category of a normal administrative task.
x'0119'	Load Function-Control Vector	Required to load a function control vector.
x'011A'	Clear Function-Control Vector	Required to clear a function control vector.
x'011B'	Force User Logoff	Required to force a user logoff if someone else is logged on.
x'011C'	Set EID	Required to set the environment ID.

SO#1 or **SO#2** creates the *admin* role using the following steps:

1. From CNM's main menu, select *Access Control → Roles → New*.
2. For *Role ID,* enter *admin.*
3. For *Valid Days,* make sure all days of the week are checked.
4. For *Valid Times in GMT (Start-End),* enter *00:00* to *23:59.*
5. For each ACP in the *admin Access Control Points* table listed above, search through the list of *Restricted Operations* to find the corresponding ACP. Select the ACP then click *Permit*.

6. Click *Load,* then click *OK*, and then click *Cancel* to return to CNM's main menu.

Both SOs must verify that the *admin* role contains only the ACP's listed above.

8.4.2.4 *keygen* role setup (Signing Station Only)

A separate key generation role needs to be created to allow for keys to be generated. This role will be needed ONLY on adapters you use to generate keys.

***keygen* Access Control Points**

ACP	Description in CNM	Justification
x'001D'	Compute Verification Pattern	Necessary to compute MKVP of any keys stored in key storage.
x'0100'	PKA96 Digital Signature Generate	Required to generate a digital signature.
x'0101'	PKA96 Digital Signature Verify	Required to verify a digital signature.
x'0103'	PKA 96 Key Generate	Required to generate ECC keys.
x'0104'	PKA96 Key Import	Required to import keys so public key portions can be extracted.
x'0107'	One-Way Hash, SHA-1	Required to hash logon PassPhrase data.
x'011B'	Force User Logoff	Suggested by *CCA Basic Services Reference* as necessary to log on to or log off of CCA.
x'0136'	One-Way Hash, SHA-256	Required for backwards compatibility.
x'0138'	One-Way Hash, SHA-512	Required for backwards compatibility.
x'0326'	Generate ECC Keys in the Clear	Required for Toolkit Keys used with DRUID.

SO#1 or **SO#2** creates the *keygen* role using the following steps:

1. From CNM's main menu, select *Access Control → Roles → New*.
2. For *Role ID,* enter *keygen*.
3. For *Valid Days,* make sure all days of the week are checked.
4. For *Valid Times in GMT (Start-End),* enter *00:00* to *23:59.*
5. For each ACP in the *keygen Access Control Points* table listed above, search through the list of *Restricted Operations* to find the corresponding ACP. Select the ACP then click *Permit*.
6. Click *Load,* then click *OK*, and then click *Cancel* to return to CNM's main menu.

Both SOs must verify that the *keygen* role contains only the ACP's listed above.

8.4.2.5 *crutasks* role access control points and setup (Signing Station only)

A separate *crutasks* role needs to be created to allow Signer to perform its duties. This role will be needed ONLY on adapters you use as Signing Station.

***crutasks* Access Control Points**

ACP	Description in CNM	Justification
x'001D'	Compute Verification Pattern	Necessary to compute MKVP of any keys stored in key storage.
x'0100'	PKA96 Digital Signature Generate	Required by CSNDDSG call in Signer.
x'0101'	PKA96 Digital Signature Verify	Required by CSNDDSV call in Signer.
x'0104'	PKA96 Key Import	Required to import keys so we can extract public key portions.
x'0107'	One-Way Hash, SHA-1	Required by CSNBOWH for older versions of Signer (or to hash logon pass phrase data).
x'011B'	Force User Logoff	Suggested by CCA Basic Services Guide as necessary when CSUALCT is called to log on to or log off of CCA.
x'0136'	One-Way Hash, SHA-256	Required by CSNBOWH call for older versions of Signer.
x'0138'	One-Way Hash, SHA-512	Required by CSNBOWH call in Signer.

SO#1 or **SO#2** creates the *crutasks* role using the following steps:

1. From CNM's main menu, select *Access Control → Roles → New*
2. Enter *crutasks* for the *Role ID*.
3. The *Comment* field can be left blank. (It can be blank for all roles.)
4. Verify that the *Required authentication strength* is 0. It can be blank for all roles.
5. Verify that the role is valid on all days of the week from 00:00 – 23:59. Note: Each weekday is <u>not selected</u> by default.
6. For each ACP in the *crutasks Access Control Points* table listed above, search through the list of *Restricted Operations* to find the corresponding ACP. Select the ACP then click *Permit*.
7. Click *Load,* then click *OK*, and then click *Cancel* to return to CNM's main menu.

Both SOs must verify that the *crutasks* role contains only the ACP's listed above.

8.4.2.6 PassPhrase setup

Each role will require a corresponding profile. For each profile (mkadm1, mkadm2, admin, keygen, and crutasks), create a strong passphrase that conforms to the following rules:

1. Be at least twelve characters in length but no longer than 64 characters in length.
2. Contain at least one upper case letter.
3. Contain at least one “special” character (such as an exclamation point).
4. Contain at least one number.
5. Be verified by by an appropriate password strength-checking tool such as cracklib-check.
    a) cracklib-check is located in the /usr/sbin directory on SLES systems. It can be used by invoking the command and then entering passwords at the prompt. The utility then determines the strength of the password. Passwords should be at least twelve characters in length contain at least one numeric character, contain at least one upper case letter, and one

non alpha-numeric character.

b) This command should only be performed directly on the secure signing station. Do not pipe the password to be input to the cracklib-check tool, as this stores the password in your command history.

8.4.2.7 *mkadm1* profile setup

At this point, all of the roles necessary have been created. The corresponding profiles must be created at this time.

SO#1 creates the *mkadm1* profile which is used for loading the first and third master key parts. This profile can be set up using by performing the following tasks:

1. On CNM's main menu, select *Access Control* → *Profiles* → *New.*
2. Select *PassPhrase* from the pop up menu.
3. Specify *mkadm1* in the *User ID* field.
4. Specify today's date the *Activation Date* field*.*
5. Specify today's date plus six years in the *Expiration Date* field.
6. Select *mkadm1* as the *Role* for this profile.
7. Specify a passphrase as described in section 8.4.2.6.
8. Specify a date six years in the future in the *Passphrase Expiration Date* field.
9. Click *Load,* then click *OK*, and then click *Cancel* to return to CNM's main menu.
10. **Remember the passphrase!** You will distribute it to SO#3.

SO#2 must verify that the *mkadm1* profile contains only the ACP's listed above.

8.4.2.8 *mkadm2* profile setup

SO#2 creates the *mkadm2* profile which is used for loading the second and fourth master key parts and setting the master key register. This profile can be set up using by performing the following tasks:

1. From CNM's main menu, select *Access Control* → *Profiles* → *New.*
2. Select *PassPhrase* from the pop up menu.
3. Specify *mkadm2* in the *User ID* field.
4. Specify today's date in the *Activation Date* field*.*
5. Specify today's date plus six years in the *Expiration Date* field**.**
6. Select *mkadm2* as the *Role* for this profile.
7. Specify a passphrase as described in section 8.4.2.6.
8. Specify a date six years in the future in the *Passphrase Expiration Date* field.
9. Click *Load,* then click *OK*, and then click *Cancel* to return to CNM's main menu.
10. **Remember the passphrase!** You will distribute it to SO#4.

SO#1 must verify that the *mkadm2* profile corresponds to the *mkadm2* role.

8.4.2.9 *admin* profile setup

The *admin* profile is used to perform administrative tasks using the *admin* role. It can be created by either **SO#1** or **SO#2** using CNM by performing the following steps:

1. On CNM's main menu, select Access Control → Profiles → *New.*
2. Select *PassPhrase* from the pop up menu.
3. Specify *admin* in the User ID field.
4. Specify today's date in the Activation Date field.
5. Specify a today's date plus six years in the Expiration Date field.
6. Select *admin* as the Role for this profile.
7. Specify a passphrase as described in section 8.4.2.6.
8. Specify a date six years in the future in the *Passphrase Expiration Date* field.
9. Click *Load,* then click *OK*, and then click *Cancel* to return to CNM's main menu.
10. **Remember the passphrase!** You will distribute it to the admin user.

Both **SO**s must verify that the *admin* profile corresponds to the *admin* role.

8.4.2.10 *keygen* profile setup (Signing Station Only)

The *keygen* profile is used to perform key generation. It is needed only on any adapters you plan to use to generate keys. It can be created by either **SO#1** or **SO#2** using CNM by performing the following steps:

1. From CNM's main menu, select *Access Control → Profiles → New.*
2. Select *PassPhrase* from the pop up menu.
3. Specify *keygen* in the *User ID* field.
4. Specify today's date in the *Activation Date* field.
5. Specify a today's date plus six years in the *Expiration Date* field.
6. Select *keygen* as the *Role* for this profile.
7. Specify a passphrase as described in section 8.4.2.6.
8. Specify a date six years in the future in the *Passphrase Expiration Date* field.
9. Click *Load,* then click *OK*, and then click *Cancel* to return to CNM's main menu.
10. **Remember the passphrase!** You will distribute it to the user who generates keys.

Both **SO**s must verify that the *keygen* profile corresponds to the *keygen* role.

8.4.2.11 *crutasks* profile setup

The *crutasks* profile is used to perform Signer initialization using the *crutasks* role. It can be created by either **IBM SO#1** or **IBM SO#2** using CNM by performing the following steps:

1. From CNM's main menu, select *Access Control → Profiles → New.*
2. Select pass phrase from the pop up menu.
3. Specify *crutasks* in the *User ID* field.

4. Specify today's date in the *Activation Date* field.
5. Specify a today's date plus six years in the *Expiration Date* field.
6. Select *crutasks* as the *Role* for this profile.
7. Specify a pass phrase as described in section 8.4.2.6.
8. Specify a date six years in the future in the *Passphrase Expiration Date* field.
9. Click *Load,* then click *OK*, and then click *Cancel* to return to CNM's main menu.
10. **Remember the pass phrase!** You will distribute it to the Signer user.

Both **IBM SO**'s must verify that the *crutasks* profile corresponds to the *crutasks* role.

8.4.2.12 *DEFAULT* role finalization

At this point, all of the roles and profiles necessary have been created. You must now restrict the *DEFAULT* role so that it can only be used for minimal functionality.

Warning: Do not change the DEFAULT role until you are sure you have created the other roles and profiles with the proper authority. Once the DEFAULT role is restricted as shown below, it can no longer be used to complete the steps described in the above sections.

The following ACP's must be **enabled** in the *DEFAULT* role. All other ACP's must be **restricted**.

ACP	Description in CNM	Justification
x'001D'	Compute Verification Pattern	Necessary to compute MKVP of any keys stored in key storage.
x'0107'	One-Way Hash, SHA-1	Required to log in.
x'0110'	Set Clock	Set clock in case of time drift.
x'0116'	Read Public Access-Control Information	Allows a user to see expiration dates of roles without logging in. This is public information, and should not be considered sensitive.

SO#1 should restrict the *DEFAULT* role with the following steps:

1. From CNM's main menu, select *Access Control → Roles*.
2. Select the *DEFAULT* role and click *Edit*.
3. Click *Restrict All*. Then select each ACP listed in the table above and click *Permit*.
4. Click *Load,* click *OK*, and then click *Cancel* to return to the main menu.
5. Verify the contents of the role as listed above. Also verify that the role is valid on all days from *00:00 – 23:59*.

SO#2 must verify that the *DEFAULT* role contains only the ACP's listed above. Select the *DEFAULT* role and click *Edit* to review and verify the role.

8.4.3 Summary of roles and profiles

This table summarizes the roles and profiles involved in this sample setup.

Table 4: CCA Role and Profile Configuration

User	Corresponding CCA Profile	Associated CCA Role	Permitted Operations ACP	Description
SO#1 SO#3	mkadm1	mkadm1	x'001D'	Compute Verification Pattern
			x'0107'	One-Way Hash, SHA-1
			x'0116'	Read Public Access Control Information
			x'0320'	Load First APKA Master Key Part
			x'0321'	Combine Intermediate APKA Master Key Parts
SO#2 SO#4	mkadm2	mkadm2	x'001D'	Compute Verification Pattern
			x'0107'	One-Way Hash, SHA-1
			x'0116'	Read Public Access Control Information
			x'0119'	Load Function-Control Vector
			x'011A'	Clear Function-Control Vector
			x'0321'	Combine Intermediate APKA Master Key Parts
			x'0322'	Activate New APKA Master Key (SET)
admin	admin	admin	x'0107'	One-Way Hash, SHA-1
			x'010F'	Reset Intrusion Latch
			x'0110'	Set Clock
			x'0111'	Reinitialize Device
			x'0112'	Initialize Access Control System
			x'0113'	Change User Profile Expiration Date
			x'0114'	Change User Profile Authentication Data
			x'0115'	Reset User Profile Logon-Attempt-Failure Count
			x'0116'	Read Public Access Control Information
			x'0117'	Delete user Profile
			x'0118'	Delete Role
			x'0119'	Load Function-Control Vector

User	Corresponding CCA Profile	Associated CCA Role	Permitted Operations	
			ACP	Description
			x'011A'	Clear Function-Control Vector
			x'011B'	Force User Logoff
			x'011C'	Set EID
Signer user	crutasks	crutasks	x'001D'	Compute Verification Pattern
			x'0100'	PKA96 Digital Signature Generate
			x'0101'	PKA96 Digital Signature Verify
			x'0104'	PKA96 Key Import
			x'0107'	One-Way Hash, SHA-1
			x'011B'	Force User Logoff
			x'0136'	One-Way Hash, SHA-256
			x'0138'	One-Way Hash, SHA-512
all other users	< NONE >	DEFAULT	x'001D' x'0107' x'0110' x'0116'	Compute Verification Pattern One-Way Hash, SHA-1 Set Clock Read Public Access-Control Information

8.4.4 Using Signer

On Linux, the Signer and Packager utilities are in the cctk/<version>/bin/host/linux directory in the Toolkit. Scripts that demonstrate their usage are found in the signing directory. This directory's structure looks like this:

Directory / File	Contents
signing	Top-level directory for Signing, Packaging, and Key Generation scripts. Contains script and configuration files needed to sign and/or package custom images for test and/or production.
signing/images	Directory to store images to be signed by sample scripts.
signing/keys	Contains sample test keys used with signing and packaging samples. Under no circumstances should these keys be used during a production signing.
signing/packager	Contains sample scripts demonstrating how to package signed outputs.

The Toolkit includes scripts and include files that make it easy to configure and run the Signer and Packager utilities. Due to the large number of input parameters, we suggest that you use these scripts instead of trying to invoke Signer or Packager manually.

After configuring a coprocessor with CCA (or a UDX) as a signing station, as discussed in Initializing CCA for use with Signer on page 63, you can use the Signer utility.

Note: the CCA profile used for this exercise must have the proper authority to run Signer. For example, a DEFAULT role that has not been changed after the coprocessor has beenconfigured as a signing station can be used. Alternatively, the *crutasks* profile discussed in crutasks role access control points and setup (Signing Station only) on page 67 and crutasks profile setup on page 70 can be used for this sample exercise.

Perform the following tasks to create sample outputs:

```
export CCTK_FS_ROOT=<fully-qualified path>/cctk/<current Toolkit version>
cd $CCTK_FS_ROOT/signing
source setenv.sh
./toolkit_signing_sample.sh -b <fully qualified path to segment-3 JFFS2
image> -c test [ -l <CCA profile with proper authority> -p <password for that
CCA profile> ]
```

Note: the -l and -p options are needed if the DEFAULT role has been edited to reduce its allowed actions.

The *setenv.sh* script sets up the environment so that Signer and Packager can run. The *toolkit_signing_sample.sh* script creates EMBURN3, REMBURN3, and SUROWN3 sample CLU files.

8.4.5 Using Packager

On Linux, the Packager utility can be used as a convenience to incorporate multiple signed outputs into one single CLU file which can be loaded onto the coprocessor using CLU. CLU will then decode this packaged file and load the individually signed files in the order specified when packaging.

Sample packaging scripts are located in signing/packager. These scripts are called from within the *toolkit_signing_sample.sh* script described above.

8.5 The Signer utility

On Linux, the Signer utility (CRUSIGNR) generates a file containing a single Miniboot command and digitally signs it so CLU can verify the command was produced by an authorized agent. The Signer utility also performs certain cryptographic functions. This section describes the syntax of the CRUSIGNR command and explains the function of the various CRUSIGNR options.[5]

The signer utility used to develop a UDX that will be run on a PCIe Cryptographic Coprocessor installed in an IBM server is named crusignr. On Linux, crusignr is in the *cctk/<version>/bin/host/linux* directory.

Syntax

CRUSIGNR [-S *profile*] *function* -F *parm_file_name*

CRUSIGNR ignores the case of its options (for example, -F and -f are equivalent).

The -*S* option directs CRUSIGNR to logon to CCA under the profile specified by *profile*.

CRUSIGNR will prompt the user to enter the password for the profile. This allows a development organization to limit the operations for which the DEFAULT role is authorized. See “Summary of steps to package and release an application” on page 89 for a list of operations CRUSIGNR must perform and

[5] The syntax diagrams in this section assume the directory that contains the various utilities shipped with the toolkit is in the search path for executable files (that is, the PATH environment variable includes cctk/bin/host/linux).

Chapter 6, "Secured Code-Signing Node" of the *CCA Installation Manual* for roles and profiles that may be of interest.

The first form of invocation causes CRUSIGNR to read arguments from the file named *parm_file_name*. Path information must also be provided if the file is not in the current directory. Each argument in the file must appear on a separate line. Once the file is exhausted, CRUSIGNR issues a prompt for each additional argument required and reads the argument from stdin.

CRUSIGNR writes all messages to stdout or stderr, where they can be captured in a log file.

CRUSIGNR uses the C runtime library to parse the arguments it reads. Numeric arguments with a leading zero are therefore treated as octal numbers rather than decimal numbers. For example, 023 is decimal 19, not decimal 23.

8.5.1 Signer operations

The first argument to CRUSIGNR specifies the Miniboot command CRUSIGNR is to generate or the cryptographic function CRUSIGNR is to perform. For Toolkit users, this command may be one of the following functions, which are grouped according to function type.

8.5.1.1 Signer cryptographic functions

Command	**Action**
KEYGEN	Generate an RSA key pair.
HASH_GEN	Generate the hash for a file using the SHA1 algorithm.
HASH_VER	Verify the hash of a file using the SHA1 algorithm.

8.5.1.2 Signer miniboot command functions

Command	**Action**
EMBURN3	Load software into segment 3.
REMBURN3	Replace the software in segment 3.
SUROWN3	Surrender ownership of segment 3.

8.5.1.3 Signer miscellaneous functions

Command	**Action**
HELP	Display instructions about how to use the program.

8.5.1.4 Signer IBM-specific functions

The following functions are used by IBM to initialize and configure the coprocessor and prepare specific CLU files for developers. Developers writing operating systems or applications for the coprocessor should not need to use these functions (although developers may need to supply as input to the packager files supplied by IBM that direct Miniboot to perform certain of these commands).

Command	
DATACERT	IBM-only function.
EMBURN2	Load software into segment 2.
ESIG2	Build an emergency signature for segment 2.
ESTOWN2	Establish ownership of segment 2.
FCVCERT	IBM-only function to certify an FCV file.
KEYCERT	IBM-only function to certify a key.
REMBURN1	IBM-only function to replace the software in segment 1.
REMBURN2	IBM-only function to replace the software in segment 2.
SIGNFILE	IBM-only function to sign a file.
SUROWN2	IBM-only function to surrender ownership of segment 2.

CRUSIGNR ignores the case of its first argument (for example, KEYGEN, keygen, and KeyGen are equivalent).

The remainder of this section describes each Signer function, including the arguments it takes if applicable (not all commands can be performed by customers), and briefly discusses how it is used during the development process. Signer functions are listed alphabetically.

8.5.2 EMBURN2 - Load software into segment 2

EMBURN2 creates a file that can be downloaded into coprocessor segment 2, which normally contains the coprocessor operating system. The file includes the public key to be associated with segment 2 and the code to load into segment 2. A developer only needs to use this command if the developer is writing an operating system for the coprocessor.

Segment 2 must be owned before an EMBURN2 command can be issued. The file the EMBURN2 command causes CRUSIGNR to create will often be packaged with commands to ensure the proper agent owns segment 2 (for example, SUROWN2 followed by ESTOWN2). The EMBURN2 command causes the coprocessor to clear data previously stored in BBRAM by code in segment 2 and/or segment 3.

8.5.3 EMBURN3 - Load software into segment 3

Syntax

The following parameters must be specified in the *parm_file_name* parameter file or on the command line:

```
EMBURN3_OUTPUT_FILENAME=<value>
EMBURN3_PART_NUMBER=<value>
EMBURN3_EC_NUMBER=<value>
EMBURN3_DESCRIPTIVE_TEXT=<value>
EMBURN3_SIGNING_KEY_CERTIFICATE_FILENAME=<value>
EMBURN3_PRIVATE_SIGNING_KEY_FILENAME=<value>
EMBURN3_SEGMENT2_OWNERID=<value>
EMBURN3_SEGMENT3_OWNERID=<value>
EMBURN3_SEGMENT_IMAGE_TO_LOAD_FILENAME=<value>
EMBURN3_SEGMENT_PUBLIC_KEY_FILENAME=<value>
EMBURN3_EMERGENCY_SIGNATURE_FILENAME=<value>
EMBURN3_ADAPTER_FAMILY_TARGET=<value>
EMBURN3_SEGMENT_IMAGE_TITLE=<value>
EMBURN3_SEGMENT_REVISION_NUMBER=<value>
<optional targeting arguments>
```

EMBURN3 creates a file that can be downloaded into coprocessor segment 3, which normally contains a read-only disk image of a coprocessor application. The file includes the public key to be associated with segment 3 and the disk image to load into segment 3.

Segment 3 must be owned before an EMBURN3 command can be issued. The file the EMBURN3 command causes CRUSIGNR to create will often be packaged with commands to ensure the proper agent owns segment 3 (for example, SUROWN3 followed by ESTOWN3). The EMBURN3 command causes the coprocessor to clear data previously stored in BBRAM by code in segment 3.

This command takes the following arguments:

- EMBURN3_OUTPUT_FILENAME is the name of the file CRUSIGNR generates to hold the EMBURN3 command. Path information must also be provided if the file is not in the current directory. By convention, the file extension is .clu.
- EMBURN3_PART_NUMBER, EMBURN3_EC_NUMBER, and EMBURN3_DESCRIPTIVE_TEXT provide certain descriptive information that is incorporated into the output file. See "File description arguments" on page 83 for details.
- EMBURN3_SIGNING_KEY_CERTIFICATE_FILENAME and
- EMBURN3_PRIVATE_SIGNING_KEY_FILENAME specify the ECC private key that CRUSIGNR will use to sign the output file and the certificate provided by IBM for the corresponding ECC public key. See "Signature key arguments" on page 83 for details.
- EMBURN3_ADAPTER_FAMILY_TARGET, EMBURN3_SEGMENT_IMAGE_TITLE, and EMBURN3_SEGMENT_REVISION_NUMBER specify the name of the file that is to be loaded into segment 3 (for example, the file that contains the read-only disk image) and provide certain descriptive information about the image that is also downloaded to the coprocessor. See "Image file arguments" on page 84 for details.
- EMBURN3_SEGMENT_PUBLIC_KEY_FILENAME – note this should actually point to the entire keypair, both private and public – is the name of a file that contains an ECCkeypair. Path information must also be provided if the file is not in the current directory. The public key in this file is the new public key to be associated with segment 3.[6] This key is downloaded to the coprocessor and is used

[6] If desired, the new public key may be the same as the public key currently associated with segment 3, if there is one.

to authenticate subsequent commands that affect segment 3. The key must be the same as the public key contained in the emergency signature information in the EMBURN3_EMERGENCY_SIGNATURE_FILENAME file.

CRUSIGNR includes in the output file a hash of the file enciphered using the private key in the *privkey_fn* file. The coprocessor uses the public key in the emergency signature information in the *esig_fn* file to validate the hash and rejects the EMBURN3 command if the validation fails.

- EMBURN3_EMERGENCY_SIGNATURE_FILENAME is the name of the file that contains emergency signature information provided by IBM. Path information must also be provided if the file is not in the current directory. It includes the public key from the EMBURN3_SEGMENT_PUBLIC_KEY_FILENAME file and includes a hash of the emergency signature information enciphered using the private key corresponding to the public key associated with segment 2. The coprocessor uses the public key associated with segment 2 to validate the hash and rejects the EMBURN3 command if the validation fails.
- EMBURN3_SEGMENT2_OWNERID is the owner identifier associated with segment 2. The coprocessor compares this value to the owner identifier stored in the coprocessor and rejects the EMBURN3 command if the two identifiers are not equal.
- EMBURN3_SEGMENT3_OWNERID is the owner identifier associated with segment 3. The coprocessor compares this value to the owner identifier stored in the coprocessor and rejects the EMBURN3 command if the two identifiers are not equal.

8.5.4 ESIG3 - Build emergency signature for segment 3

ESIG3 creates a file containing an "emergency signature" that can be provided as an argument to the EMBURN3 command. As this command requires the segment-2 private key, only IBM can create these files. Customers will receive an ESIG3 file as part of the enablement request sent to IBM when creating a production image.

8.5.5 ESTOWN3 - Establish ownership of segment 3

ESTOWN3 creates a file that directs Miniboot to establish ownership of segment 3, that is, to change segment 3's state from UNOWNED to OWNED_BUT_UNRELIABLE. The file includes the owner identifier of the new owner, which is saved in the coprocessor. A developer will only need to use this command if the developer is writing an operating system for the coprocessor: the developer owns segment 2 and uses the ESTOWN3 command to assign ownership of segment 3 to an agent developing a segment 3 application to run on top of the operating system.

Segment 3 must be unowned before an ESTOWN3 command can be issued. The file this command causes CRUSIGNR to create will often be packaged with commands to surrender ownership of segment 3 and load software into segment 3 after the new owner is established (for example, SUROWN3 and EMBURN3).

As this command requires the segment-2 private key, only IBM can create these files. Customers will receive an ESTOWN3 file as part of the enablement request sent to IBM when creating a production image.

8.5.6 HASH_GEN - Generate hash for file

Syntax

The following parameters must be specified in the *parm_file_name* parameter file or on the command line:

```
HASHGEN_INPUT_FILENAME_TO_HASH=<value>
HASHGEN_OUTPUT_FILENAME=<value>
HASHGEN_HASH_ALGORITHM="SHA-512"
```

HASH_GEN uses the SHA512algorithm to generate a hash for the file HASHGEN_INPUT_FILENAME_TO_HASH and writes the result to the file HASHGEN_OUTPUT_FILENAME. The output file consists of groups of four characters representing hexadecimal digits separated by blanks (for example, 03A2 8989 BD90 FFED 0078).

HASHGEN_INPUT_FILENAME_TO_HASH and HASHGEN_OUTPUT_FILENAME must include path information if either file is not in the current directory. HASHGEN_HASH_ALGORITHM must be "SHA-512 " (note the extra space at the end).

8.5.7 HASH_VER - Verify hash of file

Syntax

The following parameters must be specified in the *parm_file_name* parameter file or on the command line:

```
HASHVER_INPUT_HASH_TO_VERIFY_FILENAME=<value>
HASHVER_INPUT_FILENAME_TO_VERIFY=<value>
HASHVER_HASH_ALGORITHM="SHA-512 "
```

HASH_VER verifies that the hash in the file HASHVER_INPUT_HASH_TO_VERIFY_FILENAME matches the hash the HASH_GEN function would generate given HASHVER_INPUT_FILENAME_TO_VERIFY as input and issues a message indicating the result The HASHVER_INPUT_HASH_TO_VERIFY_FILENAME file has the same format as the HASHGEN_OUTPUT_FILENAME file generated by the HASH_GEN function. HASHVER_HASH_ALGORITHM must be "SHA-512 " (note the extra space at the end).

HASHVER_INPUT_HASH_TO_VERIFY_FILENAME and HASHVER_INPUT_FILENAME_TO_VERIFY must include path information if either file is not in the current directory.

8.5.8 KEYGEN - Generate ECC key pair

Syntax

The following parameters must be specified in the *parm_file_name* parameter file or on the command line:

```
KEYGEN_PRIVATE_KEY_ENCRYPTION_MECHANISM=<value>
KEYGEN_PRIVATE_KEY_OUTPUT_FILENAME=<value>
KEYGEN_PUBLIC_KEY_OUTPUT_FILENAME=<value>
KEYGEN_SKELETON_KEYTOKEN_FILENAME=<value>
[KEYGEN_CAN_GENERATED_KEY_BE_CLONED=<value>]
[KEYGEN_TRANSPORT_KEYTOKEN_FILENAME=<value>]
```

KEYGEN generates an ECC keypair and saves it in the file KEYGEN_PRIVATE_KEY_OUTPUT_FILENAME. The public key is also saved in the file KEYGEN_PUBLIC_KEY_OUTPUT_FILENAME and the hash of the public key[7] is saved in a file with the same name as KEYGEN_PUBLIC_KEY_OUTPUT_FILENAME and extension .hsh.

The file KEYGEN_SKELETON_KEYTOKEN_FILENAME determines certain characteristics of the keypair, including the key length (that is, the number of bits in the modulus) and the public key exponent. One or more standard skeletons are provided with the Toolkit. A developer can also generate customized skeleton files.

The file KEYGEN_TRANSPORT_KEYTOKEN_FILENAME contains a DES IMPORTER or DES EXPORTER key-encrypting key. This parameter will typically not be used by customers.

A filename must include path information if the file is not in the current directory.

CRUSIGNR uses the PKA_Key_Generate CCA verb to generate the keypair. The KEYGEN_PRIVATE_KEY_ENCRYPTION_MECHANISM argument determines the *rule_array* parameter passed with the PKA_Key_Generate verb as follows:

- 0 - Use MASTER for the *rule_array* parameter. This causes the coprocessor to encrypt the ECCkeypair in **KEYGEN_PRIVATE_KEY_OUTPUT_FILENAME** with the coprocessor CCA master key before returning the keypair.
- 2 - Use CLEAR for the *rule_array* parameter. This causes the coprocessor to return the RSA keypair in **KEYGEN_PRIVATE_KEY_OUTPUT_FILENAME** "in the clear" (that is, the private key parts in the file are not encrypted).

Refer to the *CCA Basic Services Reference* for details on the format of skeleton files and the CCA PKA_Key_Generate verb. For convenience a sample skeleton key token for ECC p-521 keys is provided in the file signing/keys/skeleton_key_tokens/ecc_p521_4767_default_skeleton_key_20130531-1315.tok.

8.5.9 REMBURN2 - Replace software in segment 2

REMBURN2 creates a file that can be downloaded into coprocessor segment 2, which normally contains the coprocessor operating system. The file includes the public key to be associated with segment 2 and the code to load into segment 2. A developer will only need to use this command if the developer is writing an operating system for the coprocessor.

Segment 2 must already be occupied (that is, segment 2's state must be RUNNABLE or RUNNABLE_BUT_UNRELIABLE) before a REMBURN2 command can be issued.

8.5.10 REMBURN3 - Replace software in segment 3

Syntax

The following parameters must be specified in the *parm_file_name* parameter file or on the command line:

```
REMBURN3_OUTPUT_FILENAME=<value>
REMBURN3_PART_NUMBER=<value>
```

[7] The KEYGEN command computes the hash in the same manner and stores it in the same format as the HASH_GEN command.

```
REMBURN3_EC_NUMBER=<value>
REMBURN3_DESCRIPTIVE_TEXT=<value>
REMBURN3_SIGNING_KEY_CERTIFICATE_FILENAME=<value>
REMBURN3_PRIVATE_SIGNING_KEY_FILENAME=<value>
REMBURN3_SEGMENT_IMAGE_TO_LOAD_FILENAME=<value>
REMBURN3_ADAPTER_FAMILY_TARGET=<value>
REMBURN3_SEGMENT_IMAGE_TITLE=<value>
REMBURN3_SEGMENT_REVISION_NUMBER=<value>
REMBURN3_SEGMENT_PUBLIC_KEY_FILENAME=<value>
REMBURN3_SEGMENT_PRIVATE_KEY_FILENAME=<value>
REMBURN3_SEGMENT2_OWNERID=<value>
REMBURN3_SEGMENT3_OWNERID=<value>
<optional targeting arguments>
```

REMBURN3 creates a file that can be downloaded into coprocessor segment 3, which normally contains a read-only disk image of a coprocessor application. The file includes the public key to be associated with segment 3 and the disk image to load into segment 3.

Segment 3 must already be occupied (that is, segment 3's state must be RUNNABLE or RUNNABLE_BUT_UNRELIABLE) before a REMBURN3 command can be issued.
This command takes the following arguments:

- REMBURN3_OUTPUT_FILENAME is the name of the file CRUSIGNR generates to hold the REMBURN3 command. Path information must also be provided if the file is not in the current directory. By convention, the file extension is .clu.
- REMBURN3_EC_NUMBER, REMBURN3_PART_NUMBER, REMBURN3_DESCRIPTIVE_TEXT provides certain descriptive information that is incorporated into the output file. See “File description arguments” on page 83 for details.
- REMBURN3_PRIVATE_SIGNING_KEY_FILENAME, REMBURN3_SIGNING_KEY_CERTIFICATE_FILENAME specifies the ECC private key that CRUSIGNR will use to sign the output file and the certificate provided by IBM for the corresponding ECC public key. See “Signature key arguments” on page 83 for details.
- REMBURN3_SEGMENT_IMAGE_TO_LOAD_FILENAME, REMBURN3_ADAPTER_FAMILY_TARGET, REMBURN3_SEGMENT_IMAGE_TITLE, REMBURN3_SEGMENT_REVISION_NUMBER specifies the name of the file that is to be loaded into segment 3 and provides certain descriptive information about the code that is also downloaded to the coprocessor. See “Image file arguments” on page 84 for details.
- REMBURN3_SEGMENT_PUBLIC_KEY_FILENAME is the name of the file that contains the public key to be associated with segment 3.[8] Path information must also be provided if the file is not in the current directory. This key is downloaded to the coprocessor (replacing the key that is already there) and is used to authenticate subsequent commands that affect segment 3.
- REMBURN3_SEGMENT_PRIVATE_KEY_FILENAME is the name of a file that contains an ECCkeypair. Path information must also be provided if the file is not in the current directory. The public key in this file must be the public key associated with segment 3. CRUSIGNR includes in the output file a hash of the file enciphered using the private key from the **REMBURN3_PRIVATE_SIGNING_KEY_FILENAME** file. The coprocessor uses the public key associated with segment 3 to validate the hash and rejects the REMBURN3 command if the validation fails.

[8] If desired, the new public key may be the same as the public key currently associated with the segment.

- REMBURN3_SEGMENT2_OWNERID is the owner identifier associated with segment 2. The coprocessor compares this value to the owner identifier stored in the coprocessor and rejects the REMBURN3 command if the two identifiers are not equal.
- REMBURN3_SEGMENT3_OWNERID is the owner identifier associated with segment 3. The coprocessor compares this value to the owner identifier stored in the coprocessor and rejects the REMBURN3 command if the two identifiers are not equal.

8.5.11 SUROWN2 - Surrender ownership of segment 2

SUROWN2 creates a file that directs Miniboot to surrender ownership of segment 2, that is, to change segment 2's state to UNOWNED.[9] A developer will only need to use this command if the developer is writing an operating system for the coprocessor.

Segment 2 must be owned before a SUROWN2 command can be issued. The file this command causes CRUSIGNR to create will often be packaged with commands to grant ownership of segment 2 to another agent and load software into segment 2 (for example, ESTOWN2 followed by EMBURN2).

8.5.12 SUROWN3 - Surrender Ownership of Segment 3

Syntax

The following parameters must be specified in the *parm_file_name* parameter file or on the command line:

```
SUROWN3_OUTPUT_FILENAME=<value>
SUROWN3_PART_NUMBER=<value>
SUROWN3_EC_NUMBER=<value>
SUROWN3_DESCRIPTIVE_TEXT=<value>
SUROWN3_SIGNING_KEY_CERTIFICATE_FILENAME=<value>
SUROWN3_PRIVATE_SIGNING_KEY_FILENAME=<value>
SUROWN3_SEGMENT_PRIVATE_KEY_FILENAME=<value>
SUROWN3_SEGMENT2_OWNERID=<value>
SUROWN3_SEGMENT3_OWNERID=<value>
```

SUROWN3 creates a file that directs Miniboot to surrender ownership of segment 3, that is, to change segment 3's state to UNOWNED.

Segment 3 must be owned before a SUROWN3 command can be issued. The file this command causes CRUSIGNR to create will often be packaged with commands to grant ownership of segment 3 to another agent and load software into segment 3 (for example, ESTOWN3 followed by EMBURN3).

This command takes the following arguments:

- SUROWN3_OUTPUT_FILENAME is the name of the file CRUSIGNR generates to hold the SUROWN3 command. Path information must also be provided if the file is not in the current directory. By convention, the file extension is .clu
- SUROWN3_PART_NUMBER, SUROWN3_EC_NUMBER, SUROWN3_DESCRIPTIVE_TEXT provides certain descriptive information that is incorporated into the output file. See “File description arguments” on page 83 for details.

[9] This also changes segment 3’s state to UNOWNED.

- SUROWN3_PRIVATE_SIGNING_KEY_FILENAME, SUROWN3_SIGNING_KEY_CERTIFICATE_FILENAME specifies the RSA private key that CRUSIGNR will use to sign the output file and the certificate provided by IBM for the corresponding RSA public key. See “Signature key arguments” on page 83 for details.
- SUROWN3_SEGMENT_PRIVATE_KEY_FILENAME is the name of a file that contains an RSA keypair. Path information must also be provided if the file is not in the current directory. The public key in this file must be the public key associated with segment 3. CRUSIGNR includes in the output file a hash of the file enciphered using the private key from the SUROWN3_PRIVATE_SIGNING_KEY_FILENAME file. The coprocessor uses the public key associated with segment 3 to validate the hash and rejects the SUROWN3 command if the validation fails.
- SUROWN3_SEGMENT2_OWNERID is the contains the owner identifier. The coprocessor compares this value to the owner identifier stored in the coprocessor and rejects the REMBURN3 command if the two identifiers are not equal.
- SUROWN3_SEGMENT3_OWNERID is the owner identifier associated with segment 3. The coprocessor compares this value to the owner identifier stored in the coprocessor and rejects the REMBURN3 command if the two identifiers are not equal.

8.5.13 File description arguments

CRUPKGR and many CRUSIGNR functions take as arguments certain descriptive information that is incorporated into the files CRUPKGR and CRUSIGNR generate. The format of these arguments is as follows:

partnumber ECnumber description

where

- *partnumber (<task>_PART_NUMBER)* is a string containing up to eight characters. The string is padded with blanks to the full eight characters before it is incorporated into the output file.
- *ECnumber (<task>_EC_NUMBER)* is a string containing up to eight characters. The string is padded with blanks to the full eight characters before it is incorporated into the output file.
- *description (<task>_DESCRIPTIVE_TEXT)* is a string containing up to 80 characters. The string is padded with blanks to the full 80 characters before it is incorporated into the output file.

8.5.14 Signature key arguments

CRUSIGNR and CRUPKGR incorporate a digital signature in files they generate that are destined to be input to CLU. This allows CLU to verify that the file was generated by an agent authorized to do so by IBM (or by an authority IBM has so authorized).[10] The format of these arguments is

sigkey_cert_fn sigkey_fn

where

- *sigkey_cert_fn* is the name of the certificate file for the key to be used to sign the output file. Path information must also be provided if the file is not in the current directory.
- *sigkey_fn* is the name of the file containing the ECCprivate key to be used to sign the output file. Path information must also be provided if the file is not in the current directory.

[10] The signature key arguments are for the purposes of administrative control. Core security is provided by verification of other signatures and is performed inside the coprocessor.

When CRUSIGNR creates an output file containing a Miniboot command, CRUSIGNR incorporates the certificate from the *sigkey_cert_fn* file, computes a hash of the output file, encrypts the hash with the private key in the *sigkey_fn* file, and appends the encrypted hash to the output file. When CLU processes the file, CLU computes the hash of the relevant portions of the file, extracts the public key from the certificate using the public key corresponding to the private key used to create the certificate[11], uses the extracted key to decrypt the hash, and verifies that the two hash values match.

8.5.15 Image file arguments

Many CRUSIGNR functions incorporate an image file (for example, the code that is to be loaded into a segment) into the file CRUSIGNR generates. The format of the arguments that apply to an image file is as follows:

image_fn family title revision

where

- *image_fn* is the name of the file to incorporate in the output file. Path information must also be provided if the file is not in the current directory.
- *family* indicates on which models of the cryptographic coprocessor the code is intended to execute. Recognized values supported with the coprocessor are:
  5 for code that targets the IBM PCIe Cryptographic Coprocessor.
- *title* is a string containing up to 80 characters. The string is padded with blanks to the full 80 characters before it is incorporated into the output file.
- *revision* is a number between 0 and 65535, inclusive.

revision and the last 32 bytes of *title* can be referenced in targeting information. See “Targeting arguments” on page 84 for details.

8.5.16 Targeting arguments

The CRUSIGNR functions that generate Miniboot commands (EMBURN3, REMBURN3, and SUROWN3) incorporate information that specifies certain conditions that must be met before the coprocessor will accept and process the command. Because this information can be used to restrict a command so that it can only be used with coprocessors that already contain certain software or even with a specific individual coprocessor, it is called "targeting information." The format of the arguments that specify targeting information is

RTCid RTCid_mask VPDserno VPDserno_mask VPDpartno VPDpartno_mask VPDecno VPDecno_mask VPDflags VPDflags_mask bootcount_fl [bootcount_left[bootcount_right]] seg1_info [seg2_info[seg3_info]]

where

- *RTCid* and *RTCid_mask* specify a range of permitted values for the serial number incorporated in the coprocessor chip that implements the real-time clock and the battery-backed RAM.[12] Each of these arguments is a string and may contain as many as eight characters. The arguments should have the same length.

[11] The public key is compiled into CLU.

[12] That is, the value xcGetConfig returns in pInfo->AdapterID. Refer to the *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference* for details.

Each character in *RTCid_mask* must be either ASCII 0 or ASCII 1. CRUSIGNR uses *RTCid_mask* to construct an 8-byte hexadecimal number. Each byte in the hexadecimal number is set to 0xFF if the corresponding character in *RTCid_mask* is ASCII 1 and is set to 0x00 otherwise.

CRUSIGNR logically ANDs *RTCid* with the hexadecimal number derived from *RTCid_mask* and passes the result to the coprocessor. The coprocessor logically ANDs the serial number incorporated in the coprocessor's real-time clock chip with the hexadecimal number derived from *RTCid_mask* and compares the result to the value generated by CRUSIGNR. If they are not equal, the Miniboot command that incorporates the targeting information is rejected.

If a command is intended to apply to all possible coprocessors, specify an arbitrary character for *RTCid* and 0 for *RTCid_mask*.

- *VPDserno* and *VPDserno_mask* specify a range of permitted values for the coprocessor's IBM serial number.[13] Each of these arguments is a string and may contain as many as eight characters. The arguments should have the same length.

  Each character in *VPDserno_mask* must be either ASCII 0 or ASCII 1. CRUSIGNR uses *VPDserno_mask* to construct an 8-byte hexadecimal number. Each byte in the hexadecimal number is set to 0xFF if the corresponding character in *VPDserno_mask* is ASCII 1 and is set to 0x00 otherwise.

  CRUSIGNR logically ANDs *VPDserno* with the hexadecimal number derived from *VPDserno_mask* and passes the result to the coprocessor. The coprocessor logically ANDs the coprocessor's IBM serial number with the hexadecimal number derived from *VPDserno_mask* and compares the result to the value generated by CRUSIGNR. If they are not equal, the Miniboot command that incorporates the targeting information is rejected.

  If a command is intended to apply to all possible coprocessors, specify an arbitrary character for *VPDserno* and 0 for for *VPDserno_mask*.

- *VPDpartno* and *VPDpartno_mask* specify a range of permitted values for the coprocessor's IBM part number. Each of these arguments is a string and may contain as many as seven characters. The arguments should have the same length.

  Each character in *VPDpartno_mask* must be either ASCII 0 or ASCII 1. CRUSIGNR uses *VPDpartno_mask* to construct a 7-byte hexadecimal number. Each byte in the hexadecimal number is set to 0xFF if the corresponding character in *VPDpartno_mask* is ASCII 1 and is set to 0x00 otherwise.

  CRUSIGNR logically ANDs *VPDpartno* with the hexadecimal number derived from *VPDpartno_mask* and passes the result to the coprocessor. The coprocessor logically ANDs the coprocessor's IBM part number with the hexadecimal number derived from *VPDpartno_mask* and compares the result to the value generated by CRUSIGNR. If they are not equal, the Miniboot command that incorporates the targeting information is rejected.

[13] That is, the value xcGetConfig returns in pInfo->VPD.sn. Refer to the *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference* for details.

If a command is intended to apply to all possible coprocessors, specify an arbitrary character for VPDpartno and 0 for VPDpartno_mask.

- *VPDecno* and *VPDecno_mask* specify a range of permitted values for the coprocessor's IBM engineering change level.[14] Each of these arguments is a string and may contain as many as seven characters. The arguments should have the same length.

  Each character in *VPDecno_mask* must be either ASCII 0 or ASCII 1. CRUSIGNR uses *VPDecno_mask* to construct a 7-byte hexadecimal number. Each byte in the hexadecimal number is set to 0xFF if the corresponding character in *VPDecno_mask* is ASCII 1 and is set to 0x00 otherwise.

  CRUSIGNR logically ANDs *VPDecno* with the hexadecimal number derived from *VPDecno_mask* and passes the result to the coprocessor. The coprocessor logically ANDs the coprocessor's IBM engineering change level with the hexadecimal number derived from *VPDecno_mask* and compares the result to the value generated by CRUSIGNR. If they are not equal, the Miniboot command that incorporates the targeting information is rejected.

  If a command is intended to apply to all possible coprocessors, specify an arbitrary character for *VPDecno* and 0 for *VPDecno_mask*.

- *VPDflags* and *VPDflags_mask* specify a range of permitted values for the coprocessor's VPD flags.[15] Each of these arguments is a string and may contain as many as 32 characters. The arguments should have the same length.

  Each character in *VPDflags_mask* must be either ASCII 0 or ASCII 1. CRUSIGNR uses *VPDflags_mask* to construct a 32-byte hexadecimal number. Each byte in the hexadecimal number is set to 0xFF if the corresponding character in *VPDflags_mask* is ASCII 1 and is set to 0x00 otherwise.

  CRUSIGNR logically ANDs *VPDflags* with the hexadecimal number derived from *VPDflags_mask* and passes the result to the coprocessor. The coprocessor logically ANDs the last 32 bytes of the coprocessor's Vital Product Data record with the hexadecimal number derived from *VPDflags_mask* and compares the result to the value generated by CRUSIGNR. If they are not equal, the Miniboot command that incorporates the targeting information is rejected.

  If a command is intended to apply to all possible coprocessors, specify an arbitrary character for *VPDflags* and 0 for *VPDflags_mask*.

- *bootcount_fl* and *bootcount_right* are used as follows: each time the coprocessor boots, it increments one of two counters. The "right count" is a 32-bit number that is zero when the coprocessor leaves the factory and is incremented each time the coprocessor is booted in a nonzeroized state. It is set to zero if the coprocessor detects an attempt to compromise the coprocessor's security.[16] *bootcount_fl* and *bootcount_right* specify a range of permitted values for the left and right counts.

  bootcount_fl may be 0, 1, or 2. If *bootcount_fl* is 0 and *bootcount_right* does not appear and the Miniboot command that incorporates the targeting information is accepted regardless of the right count.

[14] That is, the value xcGetConfig returns in pInfo->VPD.pn. Refer to the *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference* for details.
[15] That is, the value xcGetConfig returns in the last sixteen bytes of pInfo->VPD.reserved. Refer to the *IBM 4767 PCIe Cryptographic Coprocessor Custom Software Interface Reference* for details.
[16] The DRUID utility displays the current l right count each time it is run.

If a command is intended to apply to all possible coprocessors, specify 0 for *bootcount_fl* and omit *bootcount_right*.

- *seg1_info*, *seg2_info*, and *seg3_info* specify a range of permitted values for certain of the information associated with segment 1, segment 2, and segment 3, respectively. The format of *seg1_info*, *seg2_info*, and *seg3_info* is

  segflags segflags_mask revision_min revision_max hash_fl [hash]

  where:

  - *segflags* and *segflags_mask* specify a range of permitted values for the last 32 bytes of the segment's name or title (as specified in the EMBURN or REMBURN command that loaded the segment into the coprocessor - see “Image file arguments” on page 84 for details). By convention, this portion of the name is used to hold information that specifies the version of the code loaded into the segment. Each of these arguments is a string and may contain as many as 32 characters. The arguments should have the same length.

    Each character in *segflags_mask* must be either ASCII 0 or ASCII 1. CRUSIGNR uses *segflags_mask* to construct a 32-byte hexadecimal number. Each byte in the hexadecimal number is set to 0xFF if the corresponding character in *segflags_mask* is ASCII 1 and is set to 0x00 otherwise.

  - The coprocessor logically ANDs *segflags* with the 32-byte hexadecimal number derived from *segflags_mask*. Both quantities are first extended on the right with binary zeros to a length of 80 bytes if necessary. It then logically ANDs the last 32 bytes of the name associated with the segment (as stored in the coprocessor) with the hexadecimal number derived from *segflags_mask* and compares the two results. If they are not equal, the Miniboot command that incorporates the targeting information is rejected.

    If a command is intended to apply to all possible coprocessors, specify an arbitrary character for *segflags* and 0 for *segflags_mask*.

  - *revision_min* and *revision_max* specify a range of permitted values for the segment's revision level (as specified in the EMBURN or REMBURN command that loaded the segment into the coprocessor - see “Image file arguments” on page 84 for details). Each of these arguments is a number between 0 and 65535, inclusive. *revision_max* must be greater than or equal to *revision_min*.

    The coprocessor compares the revision level associated with the segment (as stored in the coprocessor) with *revision_min* and *revision_max*. If the revision level is less than *revision_min* or greater than *revision_max*, the Miniboot command that incorporates the targeting information is rejected.

    If a command is intended to apply to all possible coprocessors, specify 0 for *revision_min* and 65535 for *revision_max*.

  - *hash_fl* and *hash* specify the segment's contents (that is, the code in the segment). *hash_fl* may be 0 or 1 or N or Y and *hash* is a string containing 20 characters.

If *hash_fl* is specified on the command line, it must be N or Y. If *hash_fl* is specified as part of the parameter file identified by CRUSIGNR's -F option, it must be 0 or 1.

If *hash_fl* is 1 or Y, hash must be a string containing 20 characters. Each character must be a hexadecimal digit (that is, ASCII 0 through 9, a through f, or A through F) and *hash* is interpreted as a 10-byte hexadecimal number (for example, 0F1E2D3C4B5A69788796 is taken to mean 0x0F1E2D3C4B5A69788796). The coprocessor computes the hash value of the contents of the segment using the SHA1 algorithm and compares the hash to the value specified by *hash*. If the two values are not equal, the Miniboot command that incorporates the targeting information is rejected.

If *hash_fl* is 0 or N, *hash* is omitted. The Miniboot command is accepted regardless of the contents of the segment.

If a command is intended to apply to all possible coprocessors, specify 0 or N for *hash_fl* and omit *hash*.

Only *seg1_info* appears in "type 1" targeting information. The EMBURN2 command incorporates type 1 targeting information.

seg1_info and *seg2_info* appear in "type 2" targeting information. The EMBURN3 command incorporates type 2 targeting information.

seg1_info, *seg2_info*, and *seg3_info* appear in "type 3" targeting information. The REMBURN3 and SUROWN3 commands incorporate type 3 targeting information.

8.6 The Packager utility

On Linux, the Packager utility (CRUPKGR) generates a file containing one or more Miniboot commands (each generated by CRUSIGNR) and digitally signs it so CLU can verify the command was produced by an authorized agent. This section describes the syntax of the CRUPKGR command and explains the function of the various CRUPKGR options.[17]

Syntax

```
CRUPKGR [-S profile] -F parm_file_name
```

CRUPKGR ignores the case of its options (for example, **-S** and **-s** are equivalent).

The **-S** option directs CRUPKGR to logon to CCA under the profile specified by profile. CRUPKGR will prompt the user to enter the password for the profile. This allows a development organization to limit the operations for which the DEFAULT role is authorized. See "CCA roles for Signer and Packager" on page 89 for a list of operations CRUPKGR must perform and Chapter 6, "Secured Code-Signing Node" of the *CCA Installation Manual* for roles and profiles that may be of interest.

CRUPKGR uses the C runtime library to parse the arguments it reads. Numeric arguments with a leading zero are therefore treated as octal numbers rather than decimal numbers. For example, 023 is decimal 19, not decimal 23.

[17] The syntax diagrams in this section assume the directory that contains the various utilities shipped with the toolkit is in the search path for executable files (that is, the PATH environment variable includes cctk/<version>/bin/host/linux).

CRUPKGR takes the following arguments as specified in the input parameter file:

- PACKAGER_OBJECT_SIGNING_KEY_PAIR, PACKAGER_OBJECT_SIGNING_KEY_CERTIFICATE specifies the ECC private key that CRUPKGR will use to sign the output file and the certificate provided by IBM for the corresponding ECCpublic key. See “Signature key arguments” on page 83 for details.
- PACKAGER_NUMBER_OF_SIGNED_FILES_TO_PACKAGE specifies the number of files (each containing a single Miniboot command) CRUPKGR is to combine into a single image. *num_files* must be greater than zero.
- PACKAGER_SIGNED_FILENAME1, PACKAGER_SIGNED_FILENAME2,... is a list containing the name of each file CRUPKGR is to combine into a single image. Path information must also be provided if the file is not in the current directory. The files are added to the image in the order in which they appear in the list.
- PACKAGER_PACKAGED_OUTPUT_FILENAME is the name of the file CRUPKGR generates to hold the combined input files. Path information must also be provided if the file is not in the current directory. By convention, the file extension is .clu. The default is *fn.clu*, where *fn* is the name of the last file in *in_fn_list*.
- PACKAGER_DISK_IMAGE_ID specifies how the output file is intended to be used. Recognized values are as follows:

  - 4 for segment 3
  - 9 for any other image
  - 12 for reload segment 3 (REMBURN3)
  - 14 for reload segment 3 (EMBURN3)
  - 16 for establish ownership of segment 3 (ESTOWN3)
  - 18 for surrender ownership of segment 3 (SUROWN3)

  Most values of PACKAGER_DISK_IMAGE_ID are associated with a single CRUSIGNR command, which is shown in parenthesis following the description of the value. For example, specify 12 to package a single CRUSIGNR file containing a REMBURN3 command. Specify 9 if the output file will contain more than one Miniboot command.

- PACKAGER_PART_NUMBER, PACKAGER_EC_NUMBER, PACKAGER_DESCRIPTION provides certain descriptive information that is incorporated into the output file. See “File description arguments” on page 83 for details.

8.7 Summary of steps to package and release an application

This section describes the steps needed to use Signer and Packager to prepare an application for release.

1. If not already installed, install the CCA Support Program on the host, install the CCA application in the coprocessor, and configure the coprocessor as a CCA test node following the instructions in chapters 3, 4, and 5 of the *CCA Installation Manual*.

2. Generate three ECC keypairs using CRUSIGNR's KEYGEN function [18]:

   ```
   export CCTK_FS_ROOT=<full path to>/cctk/<version>
   cd $CCTK_FS_ROOT/signing
   source setenv.sh
   cd signer/tasks/keygen
   ./create_4767_toolkit_signing_keys.sh test devsgnpu.inc
   ./create_4767_toolkit_signing_keys.sh test devpkgpu.inc
   ./create_4767_toolkit_signing_keys.sh test s3kdevpu.inc
   ```

   The public key files are s3kdevpu.key, devsgnpu.key, and devpkgpu.key. The corresponding hash files are s3kdevpu.hsh, devsgnpu.hsh, and devpkgpu.hsh.
   The first keypair supplies the key to be saved with the developer's application in segment 3. The second and third keypairs are used by CRUSIGNR and CRUPKGR, respectively, to generate digital signatures that CLU uses to verify that IBM has authorized its use.

3. Forward each public key generated in step 3 above to IBM. Communicate the hash value of each public key (the hash value is also generated by the KEYGEN commands) to IBM by way of a separate channel [19] to ensure an adversary has not replaced the developer's public key file with another.

4. The developer obtains the following from IBM or from the Toolkit:

   a. Certificates for the CRUSIGNR and CRUPKGR public keys (*DEVSGNPU.CRT* and *DEVPKGPU.CRT*, respectively). The developer provides these certificates as input to CRUSIGNR and CRUPKGR, as appropriate.

      These files are generated by IBM from the CRUSIGNR and CRUPKGR public keys provided by the developer. The certificates, ESTOWN3, and ESIG3 files come back from IBM as part of the signing process.

   b. The following files from the Toolkit.[20] Note: *r.j.m* in these filenames refers to release *r* major revision *j* minor revision *m* of the files.

      ▪ *establish_ownership_seg2_toolkit_OID243_r.j.m.clu*, which establishes ownership of segment 2 [21]. Segment 2 must be owned before an application or an operating system can be loaded into the coprocessor. This file is shipped with the Toolkit.

      ▪ *emergency_reload_seg2_toolkit_OID243_r.j.m.clu*, which loads the coprocessor operating system into segment 2. The operating system must be loaded before an application can be loaded into the coprocessor. This file is shipped with the Toolkit.

[18] This version of the KEYGEN command does not encrypt the private keys in the *PP.KEY files, which may not provide the degree of security required. To encrypt the private keys with the CCA master key, specify 0 rather than 2 for the KEYGEN_PRIVATE_KEY_ENCRYPTION_MECHANISM argument.
The appropriate actions should be taken to ensure the master key can be regenerated should the need arise. Refer to the *CCA Installation Manual* for details.
[19] IBM typically provides a form for this purpose that can be returned by way of fax or email.
[20] See Using Signer and Packager for details on the contents of these files.
[21] The owner identifier assigned to segment 2 (typically 243 [0xF3]).

  - *reload_seg2_toolkit_OID243_r.j.m.clu*, which replaces an existing coprocessor operating system in segment 2. This file is shipped with the Toolkit.

  - *surrender_ownership_seg2_toolkit_OID243_r.j.m.clu*, which surrenders ownership of segment 2. This removes the operating system and any application that has been loaded into the coprocessor and also clears any information the application has saved in nonvolatile memory [22]. This file is shipped with the Toolkit.

  - estown3.clu, which establishes ownership of segment 3. This file is provided by IBM as part of the signing process. IBM assigns the developer [23] an owner identifier and estown3.clu saves that value in the coprocessor. Segment 3 must be owned before an application can be loaded into the coprocessor.

    This file is generated by IBM when the developer provides its public keys.

c. An emergency signature file (esig3dev.sig) that incorporates the developer's owner identifier and segment 3 public key. The developer provides this file as input to the signer utility (CRUSIGNR) when creating a file containing an EMBURN3 command, which loads the developer's application into the coprocessor.

   This file is generated by IBM from the segment 3 public key provided by the developer.

IBM typically supplies the files listed in step 4a above in zipped form.

5. Build a version of the application for release. For example, build without debugging information or debug code, change the value of **`pAgentID->DeveloperID`** in any calls to xcAttach, and change the value of **`pRequestBlock->AgentID->DeveloperID`** in any calls to xcRequest to the owner identifier assigned by IBM.

6. Create an EMBURN3 command that incorporates the application, IBM's segment 2 ownerID, the developer's ownerID, and the developer's unique keys.

   Note: Automated examples for the creation of an EMBURN3 image using Signer can be found in the scripts in cctk/<version>/signing/signer/tasks/emburn3. Please refer to the script prologues in each directory for more information.

   A user can use CLU to download the file generated by this process to a coprocessor that contains an earlier version of the application. The EMBURN3 command clears any state information the earlier version of the application has saved in nonvolatile memory. To preserve such information, create a REMBURN3 command instead.[24]

[22] Use of a common owner identifier for segment 2 makes it easier for an end user to obtain updates to the system software in segment 2 because IBM need only create one file containing the updates and anyone with a coprocessor containing a custom application can use the file to perform the update. But it also makes it easier for someone to remove accidentally or maliciously from a coprocessor a developer’s application and any data it has saved in nonvolatile memory.
[23] That is, an OEM or organization within an OEM.
[24] The public key downloaded with the earlier version of the application must be the public key in S3KDEVPU.KEY. A new public key can be assigned when the updated version of the application is downloaded (the new public key is taken from S3KDEVPP.KEY) but the new public key cannot be loaded using an EMBURN3 command until IBM provides a certificate for the new public key.

Note: Automated examples for the creation of an REMBURN3 image using Signer can be found in the makefiles in cctk/<version>/signing/signer/tasks/remburn3. Please refer to the script prologues in each directory for more information.

8.8 CCA roles for Signer and Packager

CRUSIGNR and CRUPKGR use CCA verbs for certain operations and consequently require that certain permissions be enabled in the DEFAULT role or in the role associated with the profile under which the utilities logon to CCA:

SIGNER EMBURN/REMBURN/SUROWN

ACP	Description in CNM	Justification
x'001D'	Compute Verification Pattern	Needed to compute MKVP of any keys stored in key storage.
x'0100'	PKA96 Digital Signature Generate	Required by CSNDDSG call in SIGNER.
x'0101'	PKA96 Digital Signature Verify	Required by CSNDDSV call in SIGNER.
x'0104'	PKA96 Key Import	Required to import keys so SIGNER can extract public key portions.
x'0107'	One-Way Hash, SHA-1	Required to hash logon passphrase data.
x'011B'	Force User Logoff	Listed in *CCA Basic Services Reference* as necessary when CSUALCT is called to log on to or log off of CCA.
x'0138'	One-Way Hash, SHA-512	Required by CSNBOWH call in SIGNER.

SIGNER KEYGEN

ACP	Description in CNM	Justification
x'0326'	Generate ECC clear keys	Needed to generate ECC keys for signing operations.

9 Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectable rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, 10594, USA.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

9.1 Copying and distributing softcopy files

For online versions of this document, 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.

9.2 Trademarks

IBM and AIX are registered trademarks of the IBM Corporation in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

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

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

10 List of abbreviations

ACP	Access Control Point
AES	Advanced Encryption Standard
ANSI	American National Standards Institute
API	Application Program Interface
ASCII	American National Standard Code for Information Exchange
BBRAM	battery-backed random access memory
CCA	Common Cryptographic Architecture
CLU	Coprocessor Load Utility
CNM	Cryptographic Node Management
DES	Data Encryption Standard
FCV	Function Control Vector
HSM	Hardware Security Module
IBM	International Business Machines
ICAT	Interactive Code Analysis Tool
MD5	Message digest 5 (hashing algorithm)
MK	Master Key
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PCI-X	Peripheral Component Interconnect eXtended
PDF	Portable Document Format
PKA	Public Key Architecture
RNG	Random Number Generator
RSA	Rivest-Shamir-Adleman (algorithm)
SHA	Secure Hash Algorithm
SRDI	Security relevant data item
UDX	user-defined extension
VPD	Vital product data

11 Glossary

A

access control. Ensuring that the resources of a computer system can be accessed only by authorized users and in authorized ways.

access control point (ACP). A command that ensures that a certain resource of the cryptographic adapter can be accessed properly.

adapter. An electronic circuit board (expansion card) that a user can plug into an expansion slot to add memory or special features to a computer.

agent. (1) An application that runs within the IBM 4767 PCIe Cryptographic Coprocessor. (2) Synonym for *secure cryptographic coprocessor application*.

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. In computer security, a process used to verify the user of an information system or protected resource.

authorize. In computer security, to permit 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 coprocessor uses BBRAM to store persistent data for coprocessor applications, as well as the coprocessor device key.

C

ciphertext. Data that has been altered by any cryptographic process.

cleartext. Data that has not been altered by any cryptographic process.

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 CCA functions. The CCA API is described in the *CCA Basic Services Reference*.

coprocessor. (1) A supplementary processor that performs operations in conjunction with another processor. (2) A microprocessor on an adapter that extends the address range of the processor in the host system, or adds specialized instructions to handle a particular category of operations; for example, an I/O coprocessor, math coprocessor, or a network coprocessor.

Coprocessor Load Utility (CLU). A program used to load validated code into the IBM 4767 PCIe Cryptographic Coprocessor.

Cryptographic Coprocessor (IBM 4767). An adapter that provides a comprehensive set of cryptographic functions to a workstation.

cryptographic node. A coprocessor that provides cryptographic services such as key generation and digital signature support.

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

D

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

decipher. (1) To convert enciphered data into clear data. (2) Contrast with encipher.

device driver. (1) A file that contains the code needed to use an attached device. (2) A program that enables a computer to communicate with a specific peripheral device; for example, a printer, videodisc player, or a CD drive.

E

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.

F

feature. A part of an IBM product that can be ordered separately from the essential components of the product.

flash memory. A specialized version of erasable programmable read-only memory (EPROM) commonly used to store code in small computers.

H

host. As regards to the coprocessor, the workstation into which the coprocessor is installed.

I

Interactive Code Analysis Tool (ICAT). A remote debugger used to debug applications running within the coprocessor.

intrusion latch. A software-monitored bit. The intrusion latch does not trigger the destruction of data stored within the coprocessor.

J

jumper. A wire that joins two unconnected circuits.

K

key. In computer security, a sequence of symbols used with an algorithm to encipher or decipher data.

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

M

master key. In computer security, the top-level key in a hierarchy of key-encrypting keys (KEKs).

miniboot. Software within the coprocessor designed to initialize the operating system and to control updates to flash memory.

P

passphrase. In computer security, a string of characters known to the computer system and to a user; the user must specify it to gain full or limited access to the system and to the data stored therein.

Peripheral Component Interconnect (PCI). A 32-bit parallel computer expansion card standard.

Peripheral Component Interconnect Express (PCIe). A high-speed serial connection computer expansion card standard that replaces the PCI and PCI-X standards which were utilized in the IBM 4758 and 4764 Cryptographic Adapters.

Peripheral Component Interconnect eXtended (PCI-x). A 64-bit version of the PCI, utilized in the IBM 4764 Cryptographic Adapter.

private key. (1) In computer security, a key that is known only to the owner and used with a public key algorithm to decipher data. Data is enciphered using the related public key. (2) Contrast with public key.

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.

R

RSA algorithm. A public key encryption algorithm developed by R. Rivest, A. Shamir, and L. Adleman.

S

Security Relevant Data Item (SRDI). Data that is securely stored by an IBM Cryptographic Adapter.

V

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

vital product data (VPD). A structured description of a device or program that is recorded at the manufacturing site.

W

workstation. A terminal or microcomputer, usually one that is connected to a mainframe or a network, and from which a user can perform applications.

Numerics

4764. IBM 4764 PCI-X Cryptographic Coprocessor.

4765. IBM 4765 PCIe Cryptographic Coprocessor.

4767. IBM 4767 PCIe Cryptographic Coprocessor.

12 Index