IBM 4765 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 84.

Sixth Edition (September, 2015)
This and other publications related to the IBM 4765 PCIe Cryptographic Coprocessor can be obtained in PDF format from the product Web site. Click on the PCIe Cryptographic Coprocessor link at http://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 2011, 2015.
Note to U. S. Government Users—Documentation related to restricted rights Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.
# Contents

About this document.................................................................vii
Prerequisite knowledge......................................................................vii
Organization of this document.....................................................vii
Typographic conventions..............................................................vii
Syntax diagrams..............................................................................vii
Related publications........................................................................vii
Summary of changes........................................................................vii

1 Introduction.................................................................................1
  1.1 Available documentation.......................................................1
  1.2 Prerequisites..........................................................................2
  1.3 Development overview..........................................................3
  1.4 Development components......................................................4
  1.5 Release components.............................................................6

2 Installation and setup...............................................................8
  2.1 Installing the toolkit...............................................................8
    2.1.1 Directories and files.........................................................8
  2.2 Installing the IBM 4765 device driver......................................18

3 Developing and debugging a 4765 application..........................19
  3.1 Environment variables..........................................................19
  3.2 Coprocessor-side development process road map.....................20
  3.3 Special coding requirements during development...................21
    3.3.1 Developer identifiers for non-UDX toolkit applications.....21
    3.3.2 Attaching with the debugger............................................21
  3.4 Compiling, assembling, and linking........................................22
    3.4.1 Compiler options............................................................22
    3.4.2 Linker options...............................................................24
  3.5 Using the adapter’s Ethernet port............................................24
  3.6 Building JFFS2 filesystem Images..........................................26
  3.7 Downloading and debugging................................................27

4 Packaging and releasing a 4765 application.............................30

5 Overview of the development process......................................36
  5.1 Preparing the development platform.....................................36
    5.1.1 Determining coprocessor status.........................................36
    5.1.2 Loading the coprocessor.................................................37

6 Using CLU.................................................................................49

7 How to reboot the IBM 4765....................................................51
  7.1 How to change the host device driver timeout........................51

8 Coprocessor-side operating system files..................................53

9 Using Signer and Packager......................................................55
  9.1 Coprocessor memory segments and security.........................55
  9.2 Signing station security considerations..................................58
  9.3 Disaster recovery...............................................................59
  9.4 The 64-bit Signer and Packager utilities.................................59
    9.4.1 Initializing CCA for use with Signer.................................60
    9.4.2 Using Signer.................................................................60
    9.4.3 Using Packager..............................................................61
  9.5 The Signer utility.................................................................61
    9.5.1 Signer operations..........................................................62
    9.5.2 EMBURN2 - Load software into segment 2.......................63
    9.5.3 EMBURN3 - Load software into segment 3.......................64
    9.5.4 ESIG3 - Build emergency signature for segment 3.............65
    9.5.5 ESTOWN3 - Establish ownership of segment 3.................65
    9.5.6 HASH_GEN - Generate hash for file...............................66
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.5.7</td>
<td>HASH_VER - Verify hash of file.</td>
</tr>
<tr>
<td>9.5.8</td>
<td>KEYGEN - Generate RSA key pair.</td>
</tr>
<tr>
<td>9.5.9</td>
<td>REMBURN2 - Replace software in segment 2.</td>
</tr>
<tr>
<td>9.5.10</td>
<td>REMBURN3 - Replace software in segment 3.</td>
</tr>
<tr>
<td>9.5.11</td>
<td>SUROWN2 - Surrender ownership of segment 2.</td>
</tr>
<tr>
<td>9.5.12</td>
<td>SUROWN3 - Surrender Ownership of Segment 3.</td>
</tr>
<tr>
<td>9.5.13</td>
<td>File description arguments.</td>
</tr>
<tr>
<td>9.5.14</td>
<td>Signature key arguments.</td>
</tr>
<tr>
<td>9.5.15</td>
<td>Image file arguments.</td>
</tr>
<tr>
<td>9.5.16</td>
<td>Trust and countersignature arguments.</td>
</tr>
<tr>
<td>9.5.17</td>
<td>Targeting arguments.</td>
</tr>
<tr>
<td>9.6</td>
<td>The Packager utility.</td>
</tr>
<tr>
<td>9.7</td>
<td>Summary of steps to package and release a 4765 application.</td>
</tr>
<tr>
<td>9.8</td>
<td>CCA roles for Signer and Packager.</td>
</tr>
<tr>
<td>10</td>
<td>Notices.</td>
</tr>
<tr>
<td>10.1</td>
<td>Copying and distributing softcopy files.</td>
</tr>
<tr>
<td>10.2</td>
<td>Trademarks.</td>
</tr>
<tr>
<td>11</td>
<td>List of abbreviations and acronyms.</td>
</tr>
<tr>
<td>12</td>
<td>Glossary.</td>
</tr>
<tr>
<td>13</td>
<td>Index.</td>
</tr>
</tbody>
</table>
Figures

Figure 1 Development process overview ......................................................................................................3
Figure 2 Toolkit directory structure.............................................................................................................9
Figure 3 Development process road map ....................................................................................................20
Figure 4 Shaved cable placement ..............................................................................................................25
Figure 5 Difference in shaved / standard cables .........................................................................................25
Figure 6 Gap between shaved / standard cables ........................................................................................25
Figure 7 Factory-fresh CLU status ...............................................................................................................38
Figure 8 Development preparation process .................................................................................................45
Figure 9 State transitions for segment 2 ....................................................................................................57
Figure 10 State transitions for segment 3 ...................................................................................................58
Tables
Table 1 CLU file names comparison............................................................................................................13
Table 2 Toolkit sample programs.................................................................................................................16
Table 3 Compiler switches...........................................................................................................................22
Table 4 Developer identifiers.......................................................................................................................31
About this document

The IBM 4765 PCIe Cryptographic Coprocessor Custom Software Developer's Toolkit Guide describes the Developer's Toolkit and its components, including the tools that enable developers to:

- build applications for the IBM 4765 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 may also be 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 Developer's Toolkit and how to prepare an IBM 4765 PCIe Cryptographic Coprocessor for use as a development platform.

Chapter 3, “Developing and debugging a 4765 application” discusses in detail the use of each of the tools used during development of a coprocessor application.

Chapter 4, “Packaging and releasing a 4765 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 IBM 4765” 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, “Coprocessor-side operating system files” describes the files in the IBM 4765’s operating system segment 2.

Chapter 9, “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.
Chapter 10, “Notices” includes product and publication notices.

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

**Typographic conventions**

This publication uses the following typographic conventions:

- Commands that you enter verbatim onto the command line are presented in **monospace** type.
- Variable information and parameters, such as file names, are presented in *italic* type.
- 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.

**Syntax diagrams**

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

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

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

<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**


**Summary of changes**

This edition of the *IBM 4765 PCIe Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide* contains product information that is current with the IBM 4765 PCIe Cryptographic Coprocessor announcements.
1 Introduction

The Developer’s 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 4765 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 4765 PCIe Cryptographic Coprocessor:

- The Developer’s Toolkit.
- A machine from the IBM ServerProven™ list running an operating system that is supported for the current release of the toolkit. See the System x section of the Software download page on the IBM PCIe Cryptographic Coprocessor Web site for details: http://www-03.ibm.com/security/cryptocards/pciecc/ordersoftware.shtml
- A C compiler and linker that can cross-compile to the target PowerPC Linux. ¹
- A C compiler and linker that can compile applications for the chosen host platform.

IBM’s CCA Support Program feature is required in order to create and debug toolkit applications.

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 4765 PCIe Cryptographic Coprocessor CCA Support Program Installation Manual, which describes how to install, load, and begin to use CCA.
- The IBM 4765 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 4765 PCIe Cryptographic Coprocessor CCA User Defined Extensions Reference and Guide.

¹ Instructions for obtaining the cross-compiler can be obtained from your IBM 4765 Toolkit provider.
### 1.2 Prerequisites

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


   The IBM 4765 PCIe should be installed in a host following the instructions in the *IBM 4765 PCIe Cryptographic Coprocessor Installation Manual*, which also lists the hardware and software requirements for the host. For application development, the host must be a machine from the IBM ServerProven™ list running a supported operating system: [http://www-03.ibm.com/security/cryptocards/pciecc/ordersoftware.shtml](http://www-03.ibm.com/security/cryptocards/pciecc/ordersoftware.shtml)

2. Developers also need a copy of the IBM CCA Installer, which is available from the following link on the IBM 4765 Web site: [http://www.ibm.com/security/cryptocards/pciecc/ordersoftware.shtml](http://www.ibm.com/security/cryptocards/pciecc/ordersoftware.shtml). This installer will include the device driver for the IBM 4765 coprocessor, as well as certain utilities (such as CLU) needed for development.

3. 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 4765 Toolkit provider. Also, a sample script for building the cross-compiler is in the 4765 Toolkit in `y4tk/<version>/crossCompiler_scripts`. Please read the prologue of `build_4765_cross_compiler.sh` in that directory for details about building the cross-compiler.

4. A compiler for code running on the host system, such as gcc. For application development, the host must be a machine from the IBM ServerProven™ list running a supported operating system: [http://www-03.ibm.com/security/cryptocards/pciecc/ordersoftware.shtml](http://www-03.ibm.com/security/cryptocards/pciecc/ordersoftware.shtml). Only the compiler and linker need to be installed; other components (visual build environments and so on) are not required.

5. The IBM 4765 PCIe Application Program Development Toolkit (the Developer's Toolkit), also referred to as the y4tk, 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 4765 (ICATPYX) as well as the IBM 4765 CCA UDX Application Program Development Toolkit Extension (the UDX Toolkit).

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://sources.redhat.com/jffs2](http://sources.redhat.com/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.

Note: Refer to the Custom Programming page of the Web site located at [http://www.ibm.com/security/cryptocards](http://www.ibm.com/security/cryptocards) for more information about the toolkits. To contact IBM concerning availability of either toolkit, submit a request using the Comments and Questions form located on either the Custom Programming page or the Support page of the product Web site, or contact your Toolkit provider.
1.3 Development overview

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 is compiled using a cross-compiler.

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 Developer's Toolkit headers as necessary.  
   Note: The sample applications in `y4tk/<version>/samples` provide many examples of how to write, compile, and link both pieces of a coprocessor application.
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 `y4tk/<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 Developer's Toolkit:

**Compiler and linker**

Use the appropriate compiler (gcc) and linker for the host-side code. These are not shipped with the Developer's Toolkit. YaST can be used to install gcc for the host code.

Use a gcc-based cross-compiler for the coprocessor-side code. Instructions for obtaining the cross-compiler can be obtained from your Toolkit provider. A sample script for building the cross-compiler is in the 4765 Toolkit in `y4tk/<version>/cross_compiler_scripts`. Please read the prologue of `build_4765_cross_compiler.sh` in for details about building the cross-compiler.

**Libraries and include files**

Special stub libraries are provided in `y4tk/<version>/lib32/card/gcc/stubs` which can be used to link the coprocessor-side code against the embedded coprocessor-side shared object libraries which provide various cryptographic functionality on the coprocessor. These files allow developers to create coprocessor-side executables without having an exact copy of the coprocessor-side libraries.

When developing an application to be run on the IBM 4765, use the Developer's Toolkit include files and the UDX Toolkit include files in addition to the include files shipped 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 26 for a sample invocation of the JFFS2 utility to create an image.

  Note that the process used to build JFFS2 images has changed from the IBM 4764 to the IBM 4765. On the 4764, `mkfs.jffs2` is used to create a directory structure which will reside in `/user0` when loaded.
onto the coprocessor using DRUID or CLU.

On the 4765, 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 the init.sh used with the 4765, files may be copied into /lib and into /ramS3.

It is not possible to use a mkfs.jffs2 image created for the 4764 on a 4765 coprocessor.

mkfs.jffs2 is not shipped with the toolkit. It can be downloaded from http://sources.redhat.com/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<arch>/<platform>` subdirectory of the toolkit. DRUID should never be used to load production images onto a coprocessor.

• **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 (e.g., it is csulclu on Linux and csufclu on AIX). On Linux, CLU will typically be located in `/opt/IBM/4765/clu`.

Be careful not to confuse CLU for the 4764 with CLU for the 4765. 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 UDXs. An appropriate script must be incorporated in the JFFS2 image loaded onto the coprocessor.

The initialization script must set any environment variables that the coprocessor-side piece of your 4765 application needs to establish, as well as environment variables that the debugger daemon [ydaemon] needs, if applicable. The script must then launch the coprocessor-side piece of your 4765 application, first launching ydaemon, if applicable. The various scripts available have been constructed so that most customers will not need to alter them.

• **Segment-3 Image Creation Makefile:** The `y4tk/<version>/build_seg3_image` subdirectory includes a makefile for automated creation of a JFFS2 image to be loaded onto the 4765 using DRUID. This makefile also creates helper scripts that export environment variables used when debugging with ICAT.

**CLU input files**
The Developer's 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.
Debugger
The Developer's Toolkit includes the IBM Interactive Code Analysis Tool (ICAT), a program (icatpyx on Linux) that controls and debugs the coprocessor-side piece of a coprocessor application. For more information on ICAT, refer to the IBM 4765 PCIe Cryptographic Coprocessor ICAT Debugger Getting Started document.

ICAT also includes ydaemon, 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.

Coprocessor operating system
The Developer's Toolkit includes two copies of the embedded Linux operating system, 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.

1.5 Release components
The software required to prepare an application for release to end users, most of which is contained in the Developer's Toolkit, is as follows.

Utilities
Use the following utilities to prepare the coprocessor-side piece of a coprocessor application for release:

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

- **Packager**\(^2\), 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 (y4load/y4unload) that properly load and unload the device driver for the coprocessor. Note: the procedure for setting the default host device driver timeout has changed for the IBM 4765. See How to change the host device driver timeout on page 51 for details.

\(^2\)Signer and Packager are provided as both 32- and 64-bit executables. The 32-bit versions of Signer and Packager are unchanged from previous releases. The 64-bit versions of these utilities have been rewritten as part of an ongoing development effort to support current and future adapters on 64-bit systems. This rewrite included additional changes to Signer and Packager to clarify the input gathering procedures, improve logging, improve error messages and error handling, simplify usage, and other miscellaneous improvements.
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 http://www-03.ibm.com/security/cryptocards/pciecc/ordersoftware.shtml. 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 Developer's 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 Developer's Toolkit and the UDX Toolkit, discusses the toolkit's directory structure and contents, and lists many of the files used during development.

2.1 Installing the toolkit

The Developer's Toolkit is shipped as a zipped tarball and a corresponding md5sum file. To install it:

- Verify the md5sum of the zipped 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.

- Untar and unzip the file (tar -xvzf) to a location of your choosing. Typically, users will store a master copy of the Toolkit tarball on their machine, then extract the contents to their home directory for development.

2.1.1 Directories and files

The Developer's 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.
Each card directory contains a directory that identifies the tool set to which the contents of the directory refer. Thus, for example, `y4tk/<version>/samples/toolkit/lib32/card/gcc` (on Linux) or `.../xlc` (on AIX) contains libraries to be used when building the coprocessor-side piece of a coprocessor application using gcc targeted for a 32-bit executable. Such a directory will be denoted `<toolset>` in the discussion that follows.

Each host directory contains a directory that identifies the compiler, architecture, and platform on which development takes place. For example, `y4tk/<version>/bin/host<arch>/linux` contains host utilities to use during development in a Linux environment for a given architecture. Such a directory will be denoted `<platform>` in the discussion that follows.

### 2.1.1.1 Card directory

The `y4tk/<version>/bin/card` directory contains various daemons that a developer may wish to include in the JFFS2 image downloaded to the coprocessor.
• *cryptologkd* allows kernel messages to be passed back to the host device driver, which on Linux emits them to /var/log/messages.

• *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*.

• *xcoad* starts the Outbound Authentication (OA) daemon. For UDX applications, *startcdud* will launch *xcoad*. Toolkit applications must start *xcoad* manually in init.sh if it is required.

Note: The card-side debugger probe ydaemon will not start unless the OA daemon is running. When debugging any application, OA must be started. This is because ydaemon should not be run in a production environment, and OA enables checking the owner IDs of segment 2 and segment 3 to determine if the software is in production or development mode.

### 2.1.1.2 Platform directory

The *y4tk/<version>/bin/host<arch>/<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.

• The signer utility (CRUSIGNR), used when preparing the production version of a coprocessor application for release. See “The Signer utility” on page 61 for details.

• The packager utility (CRUPKGR), also used when preparing the production version of a coprocessor application for release. See “The Packager utility” on page 77 for details.

### 2.1.1.3 Host Device Driver Scripts Directory

• Scripts (*y4unload/y4load*) used to load and unload the device driver. These scripts must be run as root or via sudo.

  *y4load* loads the device driver:

  ```
  sudo y4load
  ```

  Note: the procedure for using *y4load* to set the default host device driver timeout has changed for the IBM 4765. See How to change the host device driver timeout on page 51 for details.

A user may want to add *y4tk/<version>/bin/host<arch>/* and *y4tk/<version>/bin/driver* to the PATH so that these scripts and utilities such as DRUID are easily accessible.
2.1.1.4 Image directory
The y4tk/<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 image is built.

Note: For the IBM 4765, 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 y4tk/<version>/clufiles directory contains files to be used as input to CLU, including those listed below. See “Using CLU” on page 49 for a complete description of how to use CLU.

Note: Certain toolkit files are no longer packaged together to form a combined CLU file containing more than one ESTOWN/EMBURN/REMBURN 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. Comparisons with filenames on xCrypto are listed below and are also in Table 1 on page 13.

The CLU file naming convention is functionname_rjm.clu, where r signifies release, j signifies major revision, and m signifies minor revision.

- **reload_seg1_xipz_ecc_factory_keyswap_rjm.clu** which loads IBM’s system software into a coprocessor¹. This file can only be loaded into a coprocessor in the factory-fresh state². Note: on xCrypto, this file is cr1RRRSS.clu. Note: this file is part of the CCA installation and is no longer included in the toolkit.

- **reload_seg1_xipz_ecc_rjm.clu**, 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. Note: on xCrypto, this file is ce1RRRSS.clu. Note: this file is part of the CCA installation and is no longer included in the Toolkit.

**Warning**

reload_seg1_xipz_ecc_rjm.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_ecc_rjm.clu need be loaded only once and that segment 1 revision does not necessarily need to match the segment 2/segment 3 revision for the coprocessor to function properly. Please contact your Toolkit provider if you have any questions about loading segment 1.

- **establish_ownership_seg2_toolkit_OIDnnn_rjm.clu**, which prepares a coprocessor for use with segment 2 ownerID nnn³. Note: on xCrypto, this file is the ESTOWN2 portion of tdvRRRSS.clu.

---
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.
³ This action sets the public key and owner identifier associated with segment 2. Currently, the owner identifier
The version of this file with ownerID 3 prepares the coprocessor for use during development. The version with ownerID 243 can be incorporated in CLU files a developer prepares for release to customers.

**Warning**

Under no circumstances should a production environment use the CLU file with ownerID 3.

establish_ownership_seg2_toolkit_OIDnnp_rjm.clu can only be loaded into a coprocessor after reload_seg1_xipz_ecc_rjm.clu has been loaded, and once loaded cannot be reloaded until the corresponding surrender_ownership_seg2_toolkit_OIDnnp_rjm.clu file has been loaded.

- emergency_reload_seg2_toolkit_OIDnnp_rjm.clu, which loads the embedded OS into the coprocessor. Note: on xCrypto, this file is the EMBURN2 portion of ttdRRRSS.clu.

This file can only be loaded into a coprocessor after the corresponding establish_ownership_seg2_toolkit_OIDnnp_rjm.clu file has been loaded. Loading this file clears BBRAM. The version of this file with ownerID 3 prepares the coprocessor for use during development. The version with ownerID 243 can be incorporated in CLU files a developer prepares for release to customers.

Under no circumstances should a production environment use the CLU file with ownerID 3.

- reload_seg2_toolkit_OIDnnp_rjm.clu, which reloads the embedded OS into the coprocessor. Note: there is no xCrypto equivalent for this file.

This file can only be loaded into a coprocessor after the corresponding emergency_reload_seg2_toolkit_OIDnnp_rjm.clu file has been loaded. Loading this file does not clear BBRAM.

- establish_ownership_seg3_toolkit_OID6_rjm.clu, which prepares a coprocessor for use in development. Note: on xCrypto, this file is te3RRRSS.clu.

This file can only be loaded into a coprocessor after the emergency_reload_seg2_toolkit_OID3_rjm.clu file has been loaded, and once loaded cannot be reloaded until the surrender_ownership_seg2_toolkit_OID3_rjm.clu file or the surrender_ownership_seg3_toolkit_OID6_rjm.clu has been loaded.

- emergency_reload_seg3_toolkit_OID6_rjm.clu, which loads the initial development environment JFFS2 image onto the coprocessor (a prerequisite for use of the coprocessor in development). Note: on xCrypto, this file is tl3RRRSS.clu.

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_ecc_factory_keyswap_rjm.clu or reload_seg1_xipz_ecc_rjm.clu. Loading CCA also causes the key associated with segment 1 to be set to the proper value.
This file can only be loaded into a coprocessor after the 
establish_ownership_seg3_toolkit_OID6_rjm.clu file has been loaded. Loading this file clears BBRAM.

- **reload_seg3_toolkit_OID6_rjm.clu**, which reloads the initial development environment JFFS2 image onto the coprocessor. Note: on xCrypto, this file is tr3RRRSS.clu.

  This file can only be loaded into a coprocessor after the emergency_reload_seg3_toolkit_OID6_rjm.clu file has been loaded. Loading this file does not clear BBRAM.

- **surrender_ownership_seg2_toolkit_OIDnnn_rjm.clu**, 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). Loading this file places the coprocessor in the same state as immediately after the reload_seg1_xipz_ecc_rjm.clu has been loaded for the first time. Note: on xCrypto, this file is trsRRRSS.clu.

  This file can only be loaded into a coprocessor after the corresponding emergency_reload_seg2_toolkit_OIDnnn_rjm.clu file has been loaded. The ownerID nnn must correspond to the currently loaded segment 2 ownerID to use this file. The version of this file with ownerID 3 prepares the coprocessor for use during development. The version with ownerID 243 can be incorporated in CLU files a developer prepares for release to customers.

  Under no circumstances should a production environment use the CLU file with ownerID 3.

- **surrender_ownership_seg2_xipz_rjm.clu**, 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_ecc_rjm.clu has been loaded for the first time. Note: on xCrypto, this file is crsRRRSS.clu.

  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**, 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_rjm.clu file is loaded.

  This file can only be loaded into a coprocessor after the emergency_reload_seg3_toolkit_OID6_rjm.clu file has been loaded.

### 2.1.1.6 CLU file names

The file names and file functions the developer loads onto the coprocessor for the IBM 4764 and IBM 4765 are different. Table 1 compares the two sets of files.

<p>| Table 1 CLU file names comparison |</p>
<table>
<thead>
<tr>
<th>Function</th>
<th>IBM 4765 file</th>
<th>IBM 4764 file</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reload segment 1</strong></td>
<td><code>reload_seg1_xipz_ecc_factory_keyswap_rjm.clu</code> <em>(in CCA installation - $IBM4765_INSTALL_DIR/clu)</em></td>
<td><code>cr1RRRSS.clu</code></td>
</tr>
<tr>
<td><strong>Update segment 1</strong></td>
<td><code>reload_seg1_xipz_ecc_rjm.clu</code> <em>(in CCA installation - $IBM4765_INSTALL_DIR/clu)</em></td>
<td><code>ce1RRRSS.clu</code></td>
</tr>
<tr>
<td><strong>Establish segment 2 ownership</strong></td>
<td><code>establish_ownership_seg2_toolkit_OID3_rjm.clu</code></td>
<td>ESTOWN2 portion of <code>tdvRRRSS.clu</code></td>
</tr>
<tr>
<td><strong>Load segment 2 for Toolkit development</strong></td>
<td><code>emergency_reload_seg2_toolkit_OID3_rjm.clu</code></td>
<td>EMBURN2 portion of <code>tdvRRRSS.clu</code></td>
</tr>
<tr>
<td><strong>Reload segment 2 for Toolkit development</strong></td>
<td><code>reload_seg2_toolkit_OID3_rjm.clu</code></td>
<td>No xCrypto equivalent</td>
</tr>
<tr>
<td><strong>Establish segment 3 ownership for Toolkit development</strong></td>
<td><code>establish_ownership_seg3_toolkit_OID6_rjm.clu</code></td>
<td><code>te3RRRSS.clu</code></td>
</tr>
<tr>
<td><strong>Load segment 3 for Toolkit development</strong></td>
<td><code>emergency_reload_seg3_toolkit_OID6_rjm.clu</code></td>
<td><code>tl3RRRSS.clu</code></td>
</tr>
<tr>
<td><strong>Reload segment 3 for Toolkit development</strong></td>
<td><code>reload_seg3_toolkit_OID6_rjm.clu</code></td>
<td><code>tr3RRRSS.clu</code></td>
</tr>
<tr>
<td><strong>Surrender segment 2 ownership for Toolkit development</strong></td>
<td><code>surrender_ownership_seg2_toolkit_OID6_rjm.clu</code></td>
<td><code>trsRRRSS.clu</code></td>
</tr>
<tr>
<td><strong>Surrender segment 2 for CCA</strong></td>
<td><code>surrender_ownership_seg2_xipz_rjm.clu</code></td>
<td><code>crsRRRSSS.clu</code></td>
</tr>
<tr>
<td><strong>Surrender segment 3 ownership for Toolkit development</strong></td>
<td><code>surrender_ownership_seg3_toolkit_OID6_rjm.clu</code></td>
<td>No xCrypto equivalent</td>
</tr>
<tr>
<td><strong>Establish segment 2 ownership for production</strong></td>
<td><code>establish_ownership_seg2_toolkit_OID243_rjm.clu</code></td>
<td>ESTOWN2 portion of <code>tdvRRRSS.clu</code></td>
</tr>
<tr>
<td><strong>Load segment 2 for production</strong></td>
<td><code>emergency_reload_seg2_toolkit_OID243_rjm.clu</code></td>
<td>EMBURN2 portion of <code>tdvRRRSS.clu</code></td>
</tr>
<tr>
<td><strong>Surrender segment 2 ownership for production</strong></td>
<td><code>surrender_ownership_seg2_toolkit_OID243_rjm.clu</code></td>
<td><code>trsRRRSS.clu</code></td>
</tr>
</tbody>
</table>

See “Overview of the development process” on page 36 for a complete discussion of which CLU files to load in each situation.

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 36 for details.

Note: The “emergency” part of the names of several of the CLU filenames 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.
2.1.1.7 Cross-compiler directory

The y4tk/<version>/cross-compiler directory contains a sample build script and associated files that show how to download and build the cross-compiler. The prologue of the script contains additional information about building the cross-compiler.

2.1.1.8 Debuggers directory

The y4tk/<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 versions of the debugger in y4tk/<version>/debuggers/linux. The 32-bit version is in …/icatpyx-nnn and the 64-bit version is in …/icatpyx64-nnn, where nnn is the version of the debugger.

See the IBM 4765 PCIe Cryptographic Coprocessor ICAT Debugger Getting Started document, located in the y4tk/<version>/debuggers/linux/icatpyx<arch>-nnn/docs directory, for information on how to use ICAT.

2.1.1.9 Docs directory

The y4tk/<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 4765 PCIe Cryptographic Coprocessor Custom Software Interface Reference, and the IBM 4765 PCIe Cryptographic Coprocessor CCA User Defined Extensions Reference and Guide.

2.1.1.10 Inc directory

The y4tk/<version>/inc directories contain include files (.h) that are required to build coprocessor applications. Note: For the IBM 4765, 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.

2.1.1.11 lib<arch>directory

The y4tk/<version>/lib<arch> directories contain shared object (.so) and other library files that are required to build host and card side coprocessor applications. Please note that there are 32 and 64 bit versions of all host-side libraries if your Toolkit was built with 64-bit support. 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 y4tk/<version>/lib<arch>/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 `y4tk/<version>/lib/card/<toolset>`; there are separate subdirectories for base (non-UDX) applications (`../stubs`) and UDX applications (`../udxlibs`).

Note: The device driver library, which is `libcryp.so` on xCrypto, has been built into the CCA shared object `libcsulcca.so` for the IBM 4765. This library, `libcsulcca.so`, is NOT shipped with the IBM 4765 Toolkit. It is a part of the CCA Installation package, and it is installed in `/usr/lib` by default. This installation package can be downloaded from http://www-03.ibm.com/security/cryptocards/pciecc/ordersoftware.shtml.

### 2.1.1.12 Samples directories

The `y4tk/<version>/samples` directories contain the source for several sample coprocessor applications (both host-side and coprocessor-side pieces).

- `y4tk/<version>/samples/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.

- `y4tk/<version>/samples/toolkit` contains samples for base (non-UDX) applications.

- `y4tk/<version>/samples/udx` contains samples for UDX applications.

#### Base toolkit samples

The toolkit sample directory contains the sample programs shown in Table 2.

<table>
<thead>
<tr>
<th>Sample name</th>
<th>Description / purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Counts the number of installed adapters.</td>
</tr>
<tr>
<td>diagnostics</td>
<td>Allows a user to run commands on the adapter and have the output displayed to the host.</td>
</tr>
<tr>
<td>drbg</td>
<td>Exercises the Deterministic Random Bit Generator (DRBG) random number generator (RNG) functions.</td>
</tr>
<tr>
<td>ethernet</td>
<td>Demonstrates how to use Ethernet / TCP/IP communications over the USB port on the 4765.</td>
</tr>
<tr>
<td>getvpdata</td>
<td>Queries coprocessor Vital Product Data.</td>
</tr>
<tr>
<td>icatprnt</td>
<td>Demonstrates how to use printf with ICAT so that the printfs will be redirected to the messages window in ICAT.</td>
</tr>
<tr>
<td>java</td>
<td>Demonstrates how to use the RTE sample with the Java JNI layer.</td>
</tr>
<tr>
<td>logging</td>
<td>Demonstrates how to log messages from the coprocessor into <code>/var/log/messages</code> on the host machine.</td>
</tr>
<tr>
<td>oa</td>
<td>Uses Outbound Authentication to display OA certificate information.</td>
</tr>
<tr>
<td>oem</td>
<td>Demonstrates iterative host to coprocessor communication.</td>
</tr>
<tr>
<td>rte</td>
<td>“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.</td>
</tr>
<tr>
<td>serial</td>
<td>Demonstrates how to use serial communications over the USB port on the 4765.</td>
</tr>
<tr>
<td>skeleton</td>
<td>Multithreaded program that demonstrates various API functionality, including:</td>
</tr>
</tbody>
</table>
• DES Encryption, Decryption, and MAC functions
• SHA-1 Hash functions
• Large Integer Math functions
• PKA functions
• Random Number Generator functions
• Reverse Then Echo (RTE) text from host
• AES Encryption, Decryption, and MAC functions

This sample provides many working examples of how to call various coprocessor-side API functions.

| stat     | Queries coprocessor status information. |
| time     | Gets and sets the time on the coprocessor. |

### UDX sample

The UDX sample directory contains a sample program that demonstrates how to write a simple user-defined extension.

#### 2.1.1.13 Shells directory

The y4tk/<version>/shells directory contains various versions of the init.sh segment 3 startup script invoked when control is passed to segment 3 on the 4765 adapter. 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 were standardized for the 4765. Changes necessary for xCrypto, such as specifying the name of the sample to start, are no longer required. See the JFFS2 discussion in “Development components” on page 4 for details about JFFS2.

#### 2.1.1.14 Signing directories

The y4tk/<version>/signing<arch> 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.

As part of an ongoing development effort to support current and future adapters on 64-bit systems, both Signer and Packager have been completely rewritten to support host-side 64-bit architectures. This rewrite includes additional changes to Signer/Packager to clarify the input gathering procedures, improve logging, improve error messages and error handling, simplify usage, and other miscellaneous improvements.

Therefore, the 32- and 64-bit signing directories are noticeably different in structure and contents. However, both versions of Signer and Packager will produce equivalent outputs. That is, files signed or created with 64-bit Signer/Packager can be loaded on a 32-bit system and vice-versa. This includes key files, certificates, and CLU files.

Customers who are used to using the 32-bit versions of Signer/Packager do not need to move up to the 64-bit versions unless they want to incorporate the benefits (such as improved logging) of the 64-bit versions of these tools, or if they do not wish to install any backwards compatibility libraries which may be needed to run the 32-bit applications on certain operating systems.
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 his security architect or toolkit provider.

2.2 Installing the IBM 4765 device driver

The device driver is part of the CCA install downloaded from the IBM Web site located at http://www.ibm.com/security/cryptocards/, and it is required that the device driver has been installed as a prerequisite for using the Toolkit.

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

```
lsmod | grep ycrypto
```

If this command returns with a line similar to:

```
ycrypto               157943  0
```

Then the device driver has been loaded.

If this command returns no output, then the y4load script can be run as root to attempt to load the device driver. If lsmod returns no output after a y4load command, it is likely that the host support program has not been installed or not been installed correctly.

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.

Note: for the IBM 4765, the device driver is part of the combined CCA library libcsulcca.so, and does not reside in its own separate library as on xCrypto. On Linux, libcsulcca.so will typically reside in the /usr/lib directory.
3 Developing and debugging a 4765 application

This chapter describes how to use the Developer's 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 23 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 36 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 y4tk/<version>/bin/host<arch>/<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:

- Y4TK_FS_ROOT must be set to point to the root of the Developer's Toolkit (that is, the fully qualified path to y4tk/<version>-). For example, if the toolkits are unzipped/untarred from /home/user, then Y4TK_FS_ROOT becomes /home/user/y4tk/<version>.

To build the coprocessor-side piece of a coprocessor application, 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/y4cross, CROSS becomes /home/user/y4cross, as in export CROSS=/home/user/y4cross.
- 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 powerpc-405-linux-, as in export GCC_NAME=powerpc-405-linux-.
- ICAT_FS_ROOT must be set to point to the root of the ICAT installation (that is, the fully qualified path to y4tk/<version>/debuggers/<platform>/icatpyx<arch>-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/y4tk/<version>/debuggers/linux/icutpyx<arch>-nnn.
Y4_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 Y4_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. Compile, assemble, and link.
2. Build a JFFS2 filesystem.
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 3; this chart simply provides more detail.

![Figure 3 Development process road map](image)
The following sections detail how to use the Developer's 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 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 along the following lines:

```c
{
    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

---

\(^1\) Refer to the description of `xcAttach` in *IBM 4765 PCIe Cryptographic Coprocessor Custom Software Interface Reference* for details.

\(^2\) The coprocessor-side debugger daemon (ydaemon) reserves the agent IDs 0xF0FF and 0xF1FF for PCI communication. CCA reserves the agent ID 0x4341 (“CA” in ASCII). These agent IDs should not be used for development purposes.
use the debugger's Jump to location function to move the execution point to the statement immediately following the loop.

Note: This loop is not necessary for UDX applications.

### 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.

The Developer's Toolkit includes makefiles that specify the proper options for each sample. See “Overview of the development process” on page 36 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 Developer's Toolkit requires a gcc cross-compiler (which must be installed separately).

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

Table 3 describes the various compiler switches and whether they are applicable to a given system.

**Table 3 Compiler switches**

<table>
<thead>
<tr>
<th>Switch</th>
<th>Function</th>
<th>Coprocessor</th>
<th>Linux Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>-DDEBUG</td>
<td>Ensure test cases enter a spin loop prior to execution. Use only for debugging. Valid for toolkit samples only.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>-DLINUX_ON_INTEL</td>
<td>Compile code for Linux.</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>-DLINUX</td>
<td>Compile code for Linux.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>-DLINUX_ON_PPC</td>
<td>Compile code for the coprocessor, rather than the host.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>-DHOST32BIT</td>
<td>Define 32-bit addressing for communication code.</td>
<td>Yes</td>
<td>Yes * - if you want a 32-bit host-side application.</td>
</tr>
<tr>
<td>-DHOST64BIT</td>
<td>Define 64-bit addressing for your application.</td>
<td>No</td>
<td>Yes * - if you want a 64-bit host-side application. For host-side applications, one of HOST32BIT or HOST64BIT must be defined.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
<td>Default</td>
<td>Custom Information</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------------------</td>
<td>---------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td><code>-m32</code></td>
<td>Force binaries built to be 32-bit executables.</td>
<td>No</td>
<td>Yes* - if you want a 32-bit host-side application.</td>
</tr>
<tr>
<td><code>-m64</code></td>
<td>Force binaries built to be 64-bit executables.</td>
<td>No</td>
<td>Yes* - if you want a 64-bit host-side application.</td>
</tr>
<tr>
<td><code>-msqrt-float</code></td>
<td>Ensure floating-point libraries are called.</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><code>-gstabs+</code></td>
<td>Produce stabs debugging information. Needed for the ICAT debugger.</td>
<td>Yes if debugging, no otherwise</td>
<td>N/A</td>
</tr>
<tr>
<td><code>-Wall</code></td>
<td>List all warnings</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><code>-Wstrict-prototypes</code></td>
<td>Force prototype warnings</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>various optimization switches</td>
<td>Optimize compiled code.</td>
<td>No if debugging</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### 3.4.1.1 Include file directory search order

The appropriate compiler options 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. `y4tk/<version>/inc`
2. `y4tk/<version>/inc/shared/include`
3. `y4tk/<version>/inc/xcmanager`
4. `y4tk/<version>/inc/cpqenv/<linux>/<gcc>`

For examples, see the makefiles in the samples directory (`y4tk/<version>/samples/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 4764 and IBM 4765 do not produce compatible output. For example, if you attempt to cross-compile the UDX sample for the IBM 4765 using the cross-compiler for the IBM 4764, you will encounter several errors when linking.

The errors are usually of the form:

```bash
libcsullib.so: undefined reference to __ctype_b_loc@GLIBC_2.3'
libcsullib.so: undefined reference to snprintf@GLIBC_2.4'
libcsullib.so: undefined reference to __sigsetjmp@GLIBC_2.3.4'
libcsullib.so: undefined reference to syslog@GLIBC_2.4'
libcsullib.so: undefined reference to sprintf@GLIBC_2.4'
libcsullib.so: undefined reference to printf@GLIBC_2.4'
libcsullib.so: undefined reference to __pthread_unregister_cancel'
libcsullib.so: undefined reference to __pthread_register_cancel'
libcsullib.so: undefined reference to fprintf@GLIBC_2.4'
```

Undefined references such as these a symptom of using the incorrect cross-compiler.
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.

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

<table>
<thead>
<tr>
<th>File Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y4tk/&lt;version&gt;/lib32/card/gcc/stubs/libxccomapi_stub.so</td>
<td>If the application will communicate with the host</td>
</tr>
<tr>
<td>y4tk/&lt;version&gt;/lib32/card/gcc/stubs/libxcmgrapi_stub.so</td>
<td>If the application will use configuration functions</td>
</tr>
<tr>
<td>y4tk/&lt;version&gt;/lib32/card/gcc/stubs/libxcoa_stub.so</td>
<td>If the application will use Outbound Authentication functions</td>
</tr>
<tr>
<td>y4tk/&lt;version&gt;/lib32/card/gcc/stubs/libxcrandom_stub.so</td>
<td>If the application will use the random number generator, or will generate DES or RSA keys</td>
</tr>
<tr>
<td>y4tk/&lt;version&gt;/lib32/card/gcc/stubs/libxcrsalnx_stub.so</td>
<td>If the application will use Large Integer Modular Math, or RSA or DSA functions</td>
</tr>
<tr>
<td>y4tk/&lt;version&gt;/lib32/card/gcc/stubs/libxcskch_stub.so</td>
<td>If the application will use AES, DES, or Hashing functions</td>
</tr>
<tr>
<td>y4tk/&lt;version&gt;/lib32/card/gcc/udxlibs/libcsullib.so</td>
<td>If the application is a UDX</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>File Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/usr/lib/libcsulcca.so or /usr/lib64/libcsulcca.so</td>
<td>Any host-side toolkit or UDX application</td>
</tr>
<tr>
<td>y4tk/&lt;version&gt;/lib&lt;arch&gt;/host/linux/debug/libcsulcca</td>
<td>Any UDX application beging debugged that needs to print out CPRB data for inspection</td>
</tr>
</tbody>
</table>

3.5 Using the adapter’s Ethernet port
The IBM 4765 supports Ethernet communication via a special shaved cable and a USB-to-Ethernet
doncle. See Figure 4 on page 25 for placement of the cable and dongle.
Figure 5 illustrates the differences between an official manufactured (or custom shaved) cable, located at the top of the picture, and a standard mini-USB to USB cable. The shaved cable at the top has a longer mini-USB connector when compared to the standard cable immediately below it.

When placed end-to-end, the length of the mini-USB connectors can be more easily compared. As Figure 6 illustrates, the manufactured/shaved mini-USB connector (located at the bottom left) is approximately 1/16" of an inch longer than the standard mini-USB connector (located at the top right). Because the IBM 4765 has a back plate placed over the mini-USB adapter on the card, this extra space is necessary to ensure that the USB cable remains firmly in place.

While it is possible to use a standard unshaved cable with the mini-USB adapter on an IBM 4765 for testing, IBM requires the use of the manufactured/shaved cable for production environments.
The IBM 4765 kernel contains driver support for USB-to-Ethernet dongles that use the following chipsets: ADMtek AN986 Pegasus, ADM8511 Pegasus II, and the ASIX AX8817x family of chipsets.

There are many commercially available dongles that use one of these chipsets. If you have any questions about the cable, chip sets, or dongles to use with the USB port, please contact your Toolkit provider.

### 3.6 Building JFFS2 filesystem Images

The development process requires the creation of a JFFS2 filesystem image that can be loaded into the coprocessor using DRUID or can be signed using CRUSIGNR and placed into a CLU file by CRUPKGR for subsequent download by CLU.

The preferred method to build JFFS2 filesystem images is to use the makefile provided in $y4tk/<version>/build_seg3_image. The makefile should be invoked from the $y4tk/<version>/build_seg3_image directory as follows:

```bash
cd ~y4tk/<version>/build_seg3_image
make -f y4tk.seg3.image.mak SAMPLE_NAME=samplename
    BUILD_TYPE={debug|prod}
```

where `samplename` is the name of the subdirectory under $y4tk/<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 is incorporated into the JFFS2 image. The `samplename` must be `wks` to build a UDX.

Listed below is an example of the output from a JFFS2 build. In this example, `y4tk/<version>` has been substituted for that directory, which was defined by Y4TK_FS_ROOT.

```bash
user@server:y4tk/<version>/build_seg3_image> make -f 
    y4tk.seg3.image.mak SAMPLE_NAME=rte BUILD_TYPE=prod
sudo rm -rf ./flashS3
user's password:
sudo mkdir ./flashS3
sudo cp y4tk/<version>/shells/y4tk.toolkit.prod.init.sh
    ./flashS3/init.sh
sudo cp y4tk/<version>/bin/card/cryptologkd ./flashS3/.
sudo cp y4tk/<version>/bin/card/xcoad ./flashS3/.
sudo cp y4tk/<version>/samples/toolkit/rte/card/gcc/sampleCardApp
    ./flashS3/.
sudo chmod -R 550 ./flashS3
sudo chmod 500 ./flashS3/init.sh
sudo chown -R 0.0 ./flashS3
sudo chown -R 501.0 ./flashS3/xcoad
sudo chmod ug+s ./flashS3/xcoad
sudo chmod a+rx ./flashS3
sudo chmod a+rx ./flashS3/*
y4tk/<version>/build_seg3_image/mkfs.jffs2 -b -r ./flashS3 -o 
    y4tk.rte.prod.`date "+%H%M%m%d%y"`.in
```

In this example, the resulting image file is named `y4tk.rte.prod.<timestamp>.bin`.

26 IBM 4765 Toolkit Guide
The image file contains the contents of the \(-y4tk/<\text{version}>/\text{build}_\text{seg3}_\text{image}/\text{flashS3}\) directory as a JFFS2 filesystem. This flashS3 directory’s contents form the basis for what is loaded on the IBM 4765 with DRUID.

**Ownership of files on the coprocessor**

It is sometimes undesirable to run code on the coprocessor as "root". The xcoad file in particular must be run as user "501". To accomplish this, if you are using Outbound Authentication, you should "chown" your xcoad file to user 501, then ensure that the suid/sgid bits are set using chmod ug+s. Refer to the man page of chmod for more details. You may want to follow a similar process with your own applications; however, you should not use "501" as the user number.

### 3.7 Downloading and debugging

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 36.

DRUID does not affect any data in the nonvolatile memory (battery-backed RAM and flash) associated with the application. 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_rjm.clu` to the coprocessor using CLU.

The syntax of the command is

32-bit:

```bash
druid [image_filename [coprocessor_number]]
```

64-bit:

```bash
druid [-l log_file] [-a adapter_number] [-f 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.
- *coprocessor_number* identifies the coprocessor to which the read-only disk image is downloaded. (More than one coprocessor may be installed in a host.) The default is 0.
- *log_file* is the name of the log file the 64-bit version of DRUID will use when loading the adapter (N/A on 32-bit DRUID).

The number assigned to a particular coprocessor depends on the order in which information about devices in the system is presented to the device driver by the host operating system. At the present time there is no way to tell *a priori* which coprocessor will be assigned a given number. If DRUID is invoked without arguments, it prompts for them.

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
- Current boot count (see “Targeting arguments” on page 73 for details)

---

3 That is, the value xcGetConfig returns in pConfigData->VPD.sn. Refer to the IBM 4765 PCIe Cryptographic Coprocessor Custom Software Interface Reference 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: DRUID can return messages that are informational only and do not indicate a “true” error condition. For example:

• Remburn3 0x85400080  on xMBRequest 5
• Too many characters in build_seg3_image/y4tk.comm.prod.1341012511.bin 56304 Tue Jan 25 13:41:12 2011 (DRUID v2.0) 6

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. 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:


Linux on the coprocessor loads and runs the initialization script file after the coprocessor is rebooted. See “How to reboot the IBM 4765” on page 51 for a description of how to reboot the coprocessor.

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 :set list! In vim and visually search for ^M characters.

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 4765 PCIe Cryptographic Coprocessor Interactive Code Analysis Tool (ICAT) Getting Started for details.

4 That is, the value of pubkey_fn supplied when DRUID last downloaded an application to the coprocessor.
5 This error can occur when a CDU operation does not complete. CDU may not complete if the application is not CDU-able.
6 This error indicates that the name of the segment being loaded on the card is too long for DRUID. The truncated name is placed on the coprocessor. For example: Truncated value = “build_seg3_image/y4tk.comm.prod.1341012511.bin 56304 Tue Jan 25 13:41:12 2011 (D”.

28 IBM 4765 Toolkit Guide
Important Notes:

1. **Warning**: ICAT requires the Outbound Authentication daemon to be running before ydaemon is started. The proper way to start the OA daemon is documented inside the initialization scripts that support debugging.

2. The coprocessor-side debugger daemon (ydaemon) will run only in a development environment where the OA daemon has been started, where the ownerID corresponding to segment 2 is 3, and where segment 3 ownerID is 6. Additionally, ydaemon 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.

3. 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 ydaemon’s startup message instead of your application’s startup message in `/var/log/messages.`
4 Packaging and releasing a 4765 application

The design for the IBM 4765 PCIe Cryptographic Coprocessor was motivated by the need to satisfy simultaneously the following requirements 1:

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.

To meet these requirements, 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 security 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 contains the coprocessor operating system.
- Segment 3 is flash and contains one or more coprocessor applications.

The security protocols that enforce these design goals are based on RSA 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 RSA 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.

The files shipped in the Developer's 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 RSA 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

---

1 For a thorough overview of the coprocessor's security goals and a description of the security architecture, refer to Building a High-Performance, Programmable Secure Coprocessor, Research Report RC21102 published by the IBM T.J. Watson Research Center in February 1998.
developer must obtain a unique developer identifier from IBM and must generate a new, unique RSA
keypair. This is summarized in Table 4 on page 31.

Table 4 Developer identifiers

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Development</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td>&quot;Generic developer&quot; Segment-2 Owner ID 3 Segment-3 Owner ID 6</td>
<td>Developer-unique identifier for Segment-3. Segment-2 Owner ID is 243 for workstation code.</td>
</tr>
<tr>
<td>Public Key</td>
<td>Generic (common) keys. For Segment-3, these keys are shipped in the sample_keys directory of the toolkit.</td>
<td>Developer-generated key.</td>
</tr>
</tbody>
</table>

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_rjm.clu:

32-Bit CLU:

<CLU> /logfile-directory/clu.log pl
~/y4tk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID3_rjm.clu

64-Bit CLU:

<CLU> -c pl -l /logfile-directory/clu_log -d
~/y4tk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID3_rjm.clu

The developer must then install the CCA Support Program on the host, install the CCA application on the
coprocessor, and configure a CCA test node. Instructions on how to complete these steps are included in
chapters 3, 4, and 5, respectively, of the IBM 4765 PCIe Cryptographic Coprocessor CCA Support
Program Installation Manual. This prepares the coprocessor for use by the signer utility (CRUSIGNR) and
the packager utility (CRUPKGR).

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

- S3KDEVPU.KEY / S3KDEVPP.KEY – The developer's segment-3 public/private keypair.
- DEVSGNPU.KEY / DEVSGNPP.KEY – A key pair used to authorize and indicate that a file being signed is being signed by a party IBM recognizes.
- 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.

for example:

---

2 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 y4tk<version>/bin/host<arch>/<platform>.

3 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
32-Bit Signer:

```
CRUSIGNR KEYGEN 2 S3KDEVPP.KEY S3KDEVPU.KEY
   ~/y4tk/<version>/signing/default_4096crt_skeleton.tkn
CRUSIGNR KEYGEN 2 DEVSGNPP.KEY DEVSGNPU.KEY
   ~/y4tk/<version>/signing/default_4096crt_skeleton.tkn
CRUSIGNR KEYGEN 2 DEVPKGPP.KEY DEVPKGPU.KEY
   ~/y4tk/<version>/signing/default_4096crt_skeleton.tkn
```

For 64-bit Signer, please see the example scripts in `signing64/cru/signer/tasks/keygen`.

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.

**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.**

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.

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

2. The following files generated by CRUSIGNR\(^4\) and included with the IBM 4765 Toolkit:

---

2 for the second argument, for example:

```
CRUSIGNR KEYGEN 0 S3KDEVPP.KEY S3DEVPU.KEY ~/y4tk/signing/default_4096crt_skeleton.tkn
```

The appropriate actions should be taken to ensure the master key can be regenerated should the need arise. Refer to the `IBM 4765 PCIe Cryptographic Coprocessor CCA Support Program Installation Manual` for details.

Users who do not wish to generate the private keys in clear may also wish to establish a profile that restricts the actions CRUSIGNR and CRUPKGR can perform. See “Using Signer and Packager” on page 55 for more information.

\(^4\) See “Using Signer and Packager” on page 55 for details on the contents of these files.
• *establish_ownership_seg2_toolkit_OID243_rjm.clu*, which establishes ownership of segment 2.\(^5\)
  Segment 2 must be owned before an application or an operating system can be loaded into the coprocessor. This file is shipped with the Developer's Toolkit.

• *emergency_reload_seg2_toolkit_OID243_rjm.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 Developer's Toolkit.

• *reload_seg2_toolkit_OID243_rjm.clu*, which replaces an existing coprocessor operating system in segment 2. This file is shipped with the Developer's Toolkit.

• *surrender_ownership_seg2_toolkit_OID243_rjm.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.\(^6\) This file is shipped with the Developer's Toolkit.

• *estown3.clu*, which establishes ownership of segment 3. IBM assigns the developer \(^7\) 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 (*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.

The developer must build a version of the application (a filesystem built with JFFS2) suitable for release. This version of the application 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.

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.

The examples in the remainder of this chapter assume that the application is not to be restricted in any way and assume that the end user will either load the application into a coprocessor shipped from the factory or will replace an earlier version of the application.

Note: The coprocessor-side debugger daemon (ydaemon) 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, ydaemon should only be incorporated into development JFFS2 images, and

---

\(^5\) The owner identifier assigned to segment 2 (typically 243 [0xF3]).

\(^6\) 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_rrrss.clu removes any custom application installed on a coprocessor regardless of the application's origin.

\(^7\) That is, an OEM or an organization within an OEM.
should never be included in any image signed with the signer utility intended for a production
environment.

The developer uses CRUSIGNR to create an EMBURN3 command that incorporates the application,
IBM's segment 2 ownerID (243), the developer's ownerID, and the developer's private key, for example:

32-Bit Signer Example:

CRUSIGNR EMBURN3 MYAPP.L3T
  part version description
  DEVSGNP1.CRT DEVSNPP.KEY
  SEG3TK.BIN 4 title revision
  S3KDEVPP.KEY ESIG3DEV.SIG
  ibm2 oem3
  1 1
  a 0 b 0 c 0 d 0 e 0 0
  x 0 0 65535 N
  x 0 0 65535 N

where part, version, and description supply information that is incorporated into the output file, title and
revision supply information that is downloaded to the coprocessor and stored with the application in
segment 3, ibm2 is the owner identifier for segment 2 8, and oem3 is the owner identifier assigned to the
developer. See “Using Signer and Packager” on page 55 for details.

Please refer to the scripts in signing64/cru/signer/tasks/emburn3 for an example of how to create an
EMBURN3 file using the 64-bit version of Signer.

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, the developer
creates a REMBURN3 command instead, for example 9:

CRUSIGNR REMBURN3 MYAPP.R3T
  part version description
  DEVSGNP1.CRT DEVSNPP.KEY
  SEG3TK.BIN 4 title revision
  S3KDEVPP.KEY S3KDEVPP.KEY
  ibm2 oem3
  1 1
  a 0 b 0 c 0 d 0 e 0 0
  x 0 0 65535 N
  x 0 0 65535 N
  x 0 0 65535 N

See “Using Signer and Packager” on page 55 for details.

8 Typically 243 (0xF3).
9 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.
Please refer to the scripts in signing64/cru/signer/tasks/remburn3 for an example of how to create an REMBURN3 file using the 64-bit version of Signer.
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.

5.1 Preparing the development platform

After the Developer's 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 `y4tk/<version>/bin/host32/<platform>` and the CCA installation directory), and that the various system files (Linux 4765 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:

32-Bit:

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

64-Bit:

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

See Figure 7 on page 38 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.

If the SEG2 ROM Status line indicates that coprocessor segment 2 is UNOWNED, continue with “Segments 2 and 3 UNOWNED” on page 37.

If the owner identifier associated with segment 2 is 2, continue with “Segments 2 and 3 RUNNABLE with OWNER 2/2” on page 40.

If the owner identifier associated with segment 2 is 3, continue with “Segments 2 and 3 RUNNABLE with OWNER 3/6” on page 41.

If the owner identifier associated with segment 2 is 243, continue with “Segment 2 OWNER 243” on page 41.

---

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 `y4tk/bin/host32/linux`).

36 IBM 4765 Toolkit Guide
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.

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 49 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.

32-bit CLU:

```bash
<CLU> /logfile-directory/clu.log PL [coprocessornumber]
~/y4tk/<version>/clufiles/<clufilename>.clu
```

64-bit CLU:

```bash
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessor_number] -d
~/y4tk/<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.

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:

**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 7 shows a typical response from a CLU ST command for a factory fresh 4765. Note the “Factory” text in the Segment 1 Image:

```
======================================================================
CSULCLU V4.1.1 4765.log st 0    begun Tue Sep 20 10:08:01 2011
*********** Command st started. ---- Tue Sep 20 10:08:01 2011
*** VPD data;  PartNum = 41U8608
*** VPD data;  EC Num = 0N44178
*** VPD data;  Ser Num = 91011140
*** VPD data;  Description = IBM 4765-001 PCI-e Cryptographic Coprocessor
```

Overview of the development process  37
Figure 7 Factory-fresh CLU status

In this case, the developer updates the system software in segment 1 by loading `reload_seg1_xipz_ecc_factory_keyswap_rjm.clu` into the coprocessor, for example:

32-Bit CLU:

```
<CLU> /logfile-directory/clu.log pl [coprocessornumber]
$IBM4765_INSTALL_DIR/clu/reload_seg1_xipz_ecc_factory_keyswap_rjm.clu
```

64-Bit CLU:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
$IBM4765_INSTALL_DIR/clu/reload_seg1_xipz_ecc_factory_keyswap_rjm.clu
```

If this command fails, further assistance from IBM is required. (The failure may indicate the public key associated with segment 1 has not been set to the expected factory default.)

If this command succeeds, the developer proceeds to load `establish_ownership_seg2_toolkit_OID3_rjm.clu` as indicated in "Segment 1 Current" below.

**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_rjm.clu` into the coprocessor, for example:

32-Bit CLU:

```
<CLU> /logfile-directory/clu.log pl [coprocessornumber]
$IBM4765_INSTALL_DIR/clu/reload_seg1_xipz_ecc_rjm.clu
```

64-Bit CLU:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
$IBM4765_INSTALL_DIR/clu/reload_seg1_xipz_ecc_rjm.clu
```
If this command succeeds, the developer proceeds to load `establish_ownership_seg2_toolkit_OID3_rjm.clu` as indicated in “Segment 1 current (development load)” below.

**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.
2. Load segment 2 with the coprocessor operating system corresponding to ownerID 3.
3. Establish ownership of segment 3 as ownerID 6.
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_rjm.clu`, followed by loading `emergency_reload_seg2_toolkit_OID3_rjm.clu`. Note: on xCrypto, this file is `tdvRRRSS.clu`. For example:

32-Bit CLU:

```plaintext
<CLU> /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
$IBM4765_INSTALL_DIR/clu/reload_seg1_xipz_ecc_rjm.clu
```

The developer then sets the owner identifier for segment 3 by loading `establish_ownership_seg3_toolkit_OID6_rjm.clu`, followed by loading `emergency_reload_seg3_toolkit_OID6_rjm.clu`, for example:

32-Bit CLU:

```plaintext
<CLU> /logfile-directory/clu.log pl [coprocessornumber]
~/y4tk/<version>/clufiles/establish_ownership_seg3_toolkit_OID6_rjm.clu
<CLU> /logfile-directory/clu.log pl [coprocessornumber]
~/y4tk/<version>/clufiles/emergency_reload_seg3_toolkit_OID6_rjm.clu
```

64-Bit CLU:

```plaintext
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/y4tk/<version>/clufiles/establish_ownership_seg2_toolkit_OID3_rjm.clu
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/y4tk/<version>/clufiles/emergency_reload_seg2_toolkit_OID3_rjm.clu
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/y4tk/<version>/clufiles/emergency_reload_seg2_toolkit_OID3_rjm.clu
```

Overview of the development process 39
64-Bit CLU:

```clu
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/<version>/clufiles/establish_ownership_seg3_toolkit_OID6_rjm.clu
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/<version>/clufiles/emergency_reload_seg3_toolkit_OID6_rjm.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 IBM 4765” on page 51),
2. compiling and linking the host reverse-then-echo driver if necessary (see “Compiling, assembling, and linking” on page 22), and
3. running the host driver, for example:

```bash
~/<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:

```bash
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 45.

### 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_rjm.clu` into the coprocessor, for example:

32-Bit CLU:

```clu
<CLU> /logfile-directory/clu.log pl [coprocessornumber]
~/<version>/clufiles/surrender_ownership_seg2_xipz_rjm.clu
```

64-Bit CLU:

```clu
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/<version>/clufiles/surrender_ownership_seg2_xipz_rjm.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 39.

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_rjm.clu into the coprocessor, for example:

32-Bit CLU:

<CLU> /logfile-directory/clu.log pl [coprocessornumber] 
~/y4tk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID3_rjm.clu

64-Bit CLU:

<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d 
~/y4tk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID3_rjm.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 39.

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_rjm.clu into the coprocessor, for example:

32-Bit CLU:

<CLU> /logfile-directory/clu.log pl [coprocessornumber] 
~/y4tk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID243_rjm.clu

64-Bit CLU:

<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d 
~/y4tk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID243_rjm.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 39.
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_rjm.clu` followed by `surrender_ownership_seg2_xipz_rjm.clu` into the coprocessor, for example:

32-Bit CLU:

```bash
<CLU> /logfile-directory/clu.log pl [coprocessornumber]
~/y4tk/<version>/clufiles/emergency_reload_seg2_xipz_rjm.clu
<CLU> /logfile-directory/clu.log pl [coprocessornumber]
~/y4tk/<version>/clufiles/surrender_ownership_seg2_xipz_rjm.clu
```

64-Bit CLU:

```bash
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/y4tk/<version>/clufiles/emergency_reload_seg2_xipz_rjm.clu
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/y4tk/<version>/clufiles/surrender_ownership_seg2_xipz_rjm.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 39.

If the segment 2 ownerID is 3, the developer loads `emergency_reload_seg2_toolkit_OID3_rjm.clu` followed by `surrender_ownership_seg2_toolkit_OID3_rjm.clu` into the coprocessor, for example:

32-Bit CLU:

```bash
<CLU> /logfile-directory/clu.log pl [coprocessornumber]
~/y4tk/<version>/clufiles/emergency_reload_seg2_toolkit_OID3_rjm.clu
<CLU> /logfile-directory/clu.log pl [coprocessornumber]
~/y4tk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID3_rjm.clu
```

64-Bit CLU:

```bash
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/y4tk/<version>/clufiles/emergency_reload_seg2_toolkit_OID3_rjm.clu
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/y4tk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID3_rjm.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 39.

If the segment 2 ownerID is 243, the developer loads `emergency_reload_seg2_toolkit_OID243_rjm.clu` followed by `surrender_ownership_seg2_toolkit_OID243_rjm.clu` into the coprocessor, for example:

42 IBM 4765 Toolkit Guide
32-Bit CLU:

```
<CLU> /logfile-directory/clu.log pl [coprocessornumber]
~/y4tk/<version>/clufiles/emergency_reload_seg2_toolkit_OID243_rjm.clu
```

```
<CLU> /logfile-directory/clu.log pl [coprocessornumber]
~/y4tk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID243_rjm.clu
```

64-Bit CLU:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/y4tk/<version>/clufiles/emergency_reload_seg2_toolkit_OID243_rjm.clu
```

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/y4tk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID243_rjm.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 39.

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_rjm.clu` into the coprocessor, for example:

32-Bit CLU:

```
<CLU> /logfile-directory/clu.log pl [coprocessornumber]
~/y4tk/<version>/clufiles/surrender_ownership_seg2_xipz_rjm.clu
```

64-Bit CLU:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/y4tk/<version>/clufiles/surrender_ownership_seg2_xipz_rjm.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 39.

If the segment 2 ownerID is 3, the developer loads `surrender_ownership_seg2_toolkit_OID3_rjm.clu` into the coprocessor, for example:

32-Bit CLU:

```
<CLU> /logfile-directory/clu.log pl [coprocessornumber]
~/y4tk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID3_rjm.clu
```

64-Bit CLU:

```
<CLU> -l /logfile-directory/clu.log -c PL [-a coprocessornumber] -d
~/y4tk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID3_rjm.clu
```

Overview of the development process 43
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 39.

If the segment 2 ownerID is 243, the developer loads `surrender_ownership_seg2_toolkit_OID243_rjm.clu` into the coprocessor, for example:

32-Bit CLU:

```
<CLU> /logfile-directory/clu.log pl [coprocessornumber]  
~/y4tk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID243_rjm.clu
```

64-Bit CLU:

```
<CLU> -l /logfile-directory/clu.log  -c PL [-a coprocessornumber] -d  
~/y4tk/<version>/clufiles/surrender_ownership_seg2_toolkit_OID243_rjm.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 39.

5.1.2.7 Development preparation summary

Figure 8 illustrates the steps involved in preparing a coprocessor for use as a development platform.
Figure 8 Development preparation process

- **Reload seg 1** means to load `reload_seg1_xipz_ecc_factory_keyswap_rjm.clu`, which loads IBM’s system software into segment 1.
- **Update seg 1** means to load `reload_seg1_xipz_ecc_rjm.clu`, which updates IBM’s system software in segment 1 of the coprocessor.
- **Own seg 2** means to load `establish_ownership_seg2_toolkit_OID3_rjm.clu`, which establishes ownership of segment 2.
- **Load seg 2** means to load `emergency_reload_seg2_toolkit_OID3_rjm.clu`, which loads the embedded OS into the coprocessor.
- **Own seg 3** means to load `establish_ownership_seg3_toolkit_OID6_rjm.clu`, which establishes ownership of segment 3.
- **Load seg 3** means to load `emergency_reload_seg3_toolkit_OID6_rjm.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. Refer to “Compiling, assembling, and linking” on page 22.

---

2 The developer is free to choose the executable file name.
5.1.2.9 Building JFFS2 filesystem images
Refer to “Building JFFS2 filesystem Images” on page 26.

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

32-BIT CLU:

<CLU> /logfile-directory/clu.log pl
~/y4tk/clufiles/emergency_reload_seg3_toolkit_OID6_rjm.clu

64-BIT CLU:

<CLU> -l /logfile-directory/clu.log -c PL -d
~/y4tk/clufiles/emergency_reload_seg3_toolkit_OID6_rjm.clu

Note: This is rarely necessary.

2. Download the JFFS2 image onto the coprocessor:

32-BIT DRUID:

druid <JFFS2 Image> {adapternumber}

64-BIT DRUID:

druid [-a adapternumber] -l log_file -f <JFFS2 Image>

3. Wait for the coprocessor to reboot and start the application.

4. Set all appropriate ICAT-related environment variables, then start the debugger and attach to the application:

icatpyx

Refer to the IBM 4765 PCIe Cryptographic Coprocessor Interactive Code Analysis Tool (ICAT) Getting Started manual for more information.

If changes to the application prove necessary, make them and then continue with “Compiling, assembling, and linking” on page 45.

5.1.2.11 Testing a 4765 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_rjm.clu
   - emergency_reload_seg2_toolkit_OID243_rjm.clu
   - reload_seg2_toolkit_OID243_rjm.clu
   - ESTOWN3.clu (file provided by IBM as part of the signing process)
   Your segment 3 CLU file for EMBURN3 (file name depends on you)

4. Wait for the coprocessor to reboot and start the application.

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 `y4tk/<version>/samples/toolkit/rte`. This sample assumes that the Linux development environment has been set up as described in “Installation and setup” on page 8 and “Loading the coprocessor” on page 37.

Follow these steps. Refer back to the previous sections in this chapter for details.

1. Set environment variables. For example:
   
   ```bash
   export Y4TK_FS_ROOT=/home/user/y4tk/<version>
   export CROSS=/home/user/y4cross
   export GCC_NAME=powerpc-405-linux-
   export Y4_JFFS2_DIR=/usr/sbin
   ```

   Note: It is assumed that the cross-compiler obtained from the IBM Linux Technology Center is built and installed following instructions from your toolkit provider.

2. Make the host side:
   ```bash
   cd <Y4TK_FS_ROOT>/samples/toolkit/rte/host/gcc
   make -f host.mak
   ```
   Note: Host side executables are built as 64-bit objects by default if the host is 64-bit.

3. Make the coprocessor side:
   ```bash
   cd <Y4TK_FS_ROOT>/samples/toolkit/rte/card/gcc
   make -f card.mak
   ```

4. Build the JFFS2 image:
   ```bash
   cd <Y4TK_FS_ROOT>/build_seg3_image
   make -f y4tk_seg3.image.mak SAMPLE_NAME=rte BUILD_TYPE=prod
   ```
5. Load the image onto the adapter:
   32-Bit DRUID:
   ```
   druid y4tk.rte.prod.<timestamp>.bin 0
   ```
   where `<timestamp>` is the timestamp of the image file produced in step 4.

   64-Bit Druid:
   ```
   druid -a 0 -l sample.log -f y4tk.rte.prod.<timestamp>.bin
   ```

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

   Note: This command:
   ```
   sudo tail -f /var/log/messages
   ```
   will display useful messages from the adapter, including a message that the application has started.

7. Run the host-side application:
   ```
   cd <Y4TK_FS_ROOT>
   samples/toolkit/rte/host/gcc/sampleHostApp 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.

The 32-bit version of CLU is unchanged from previous releases. The 64-bit version of CLU has been rewritten as part of an ongoing development effort to support current and future adapters on 64-bit systems. New usage options have been added to 64-bit CLU such that arguments are no longer positionally dependent (that is, the first argument no longer needs to be the logfilename, the second the command, etc.). For backwards compatibility, the old style usage options are still present in 64-bit CLU. Additionally, the outputs for commands like CLU ST have been slightly reformatted such that they are easier to parse using scripts. The machine readable log (MRL) output has been deprecated in 64-bit CLU, as the output was redundant when compared to the log file.

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.

Syntax

32-Bit CLU:

<CLU> logfilename command [coprocessornumber] [clufilename]

64-Bit CLU:

<CLU> -c command [-l logfilename] [-a coprocessornumber] [-d clufilename]

where:

- **logfilename** is the name of a file to which CLU appends information about the operation and its results. The file is created if it does not exist. Path information must also be provided if the file is not in the current directory.

  It is strongly recommended that the coprocessor serial number be used as the log file name. The serial number appears on the label on the bracket located at the end of the coprocessor. This practice ensures a complete history of status and code changes for the contents of each coprocessor is available. The 64-bit version of CLU uses this value by default if no log file name is specified.

- The **command** argument specifies the operation CLU is to perform. Recognized values are as follows:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL</td>
<td>Download a file containing software and/or commands to the coprocessor.</td>
</tr>
<tr>
<td>RS</td>
<td>Reset the coprocessor.</td>
</tr>
<tr>
<td>SS</td>
<td>Print information about every coprocessor installed in a host and the application each coprocessor contains.</td>
</tr>
<tr>
<td>ST</td>
<td>Print information about the coprocessor and the software it contains.</td>
</tr>
<tr>
<td>VA</td>
<td>Print and validate information about the coprocessor and the software it contains.</td>
</tr>
<tr>
<td>MR</td>
<td>Rigolino Update – Update the contents of Rigolino on the adapter.</td>
</tr>
<tr>
<td>EX</td>
<td>Exit CLU.</td>
</tr>
</tbody>
</table>
More than one coprocessor may be installed in a host. `coprocessornumber` identifies the coprocessor with which CLU is to interact. The default is the first adapter in the system, or adapter 0 (adapter numbering starts with zero).

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

- `clufilename` is the name of the file containing software and commands to download to the coprocessor. Path information must also be provided if the file is not in the current directory. This file name is used only if the PL, VA, or VF operation is specified.

### 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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OK.</td>
</tr>
<tr>
<td>Nonzero</td>
<td>Command failed. Check the log file for more information.</td>
</tr>
</tbody>
</table>
7 How to reboot the IBM 4765

An IBM 4765 can be rebooted in any of several ways:

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

   32-Bit CLU:
   <CLU> /logfile-directory/clu.log RS
   64-Bit CLU:
   <CLU> -c RS -1 /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>/y4tk/<version>/bin/host32/linux/y4unload
   b. Subsequently reloading the driver (as root or via sudo):
      <full path to>/y4tk/<version>/bin/host32/linux/y4load

   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

The process for changing the default host device driver timeout for the 4765 has changed as compared to the process for the 4764. On SLES 11, the file /etc/modprobe.conf.local must be amended to specify the timeout options passed to the 4765 host device driver. Options passed to the y4load script are overridden by the settings in /etc/modprobe.conf.local on SLES-based systems. This means that specifying a timeout value as a parameter to the y4load script will have no effect on the timeout value.

Note: To see the current value of the 4765 device driver timeout values, issue the following command:

    cat /proc/driver/ycrypto/timeout

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

Then, as root, edit /etc/modprobe.conf.local to indicate that the “install ycrypto” line specifies the desired timeout values:

---

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 y4tk/bin/host32/linux).
install ycrypto /sbin/modprobe --ignore-install ycrypto
opcodes_dir=/opt/IBM/4765 timeout=<Request Window 1 Timeout Value>,<Request Window 2 Timeout Value>,<Request Window 3 Timeout Value>,<Request Window 4 Timeout Value> ssp_timeout=<SSP timeout value> mb_timeout=<Miniboot timeout value>; /sbin/ycrypto_mknod

where all timeout values are specified in seconds. For example, if you wanted to have the timeout value set to 1200 seconds, specify the following line in /etc/modprobe.conf.local:

install ycrypto /sbin/modprobe --ignore-install ycrypto
opcodes_dir=/opt/IBM/4765 timeout=1200,1200,1200,1200 ssp_timeout=1200 mb_timeout=1200 ; /sbin/ycrypto_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 y4unload and y4load scripts in the toolkit, or you can use the modprobe command to unload and load the driver:

modprobe -r ycrypto
modprobe ycrypto

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

cat /proc/driver/ycrypto/timeout
# 8 Coprocessor-side operating system files

This chapter contains a sample listing of the 4765's operating system segment 2 files that are available in the root, bin, sbin, usr/bin, and usr/sbin directories on the adapter itself.

Note: A listing similar to this one can be obtained by installing and using the diagnostic sample program. See “Samples directories” on page 16 for a description of the samples and “Overview of the development process” on page 36 for an explanation of how to load the coprocessor and use the sample programs.

```
> ./sampleHostApp 0 'ls -l /'
```

<table>
<thead>
<tr>
<th>Permissions</th>
<th>Owner</th>
<th>Group</th>
<th>Size</th>
<th>Date/Time</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>drwxrwxr--</td>
<td>4 root</td>
<td>root</td>
<td>1024</td>
<td>Nov 20 15:31</td>
<td>bbram</td>
</tr>
<tr>
<td>drwxr-x-x</td>
<td>2 root</td>
<td>root</td>
<td>0</td>
<td>Nov 21 18:34</td>
<td>bin</td>
</tr>
<tr>
<td>drwxr-x-x</td>
<td>2 root</td>
<td>root</td>
<td>0</td>
<td>Nov 21 18:34</td>
<td>boot</td>
</tr>
<tr>
<td>drwxrwxrwx</td>
<td>5 root</td>
<td>root</td>
<td>380</td>
<td>Nov 21 18:34</td>
<td>dev</td>
</tr>
<tr>
<td>drwxr-x-x</td>
<td>3 root</td>
<td>root</td>
<td>0</td>
<td>Nov 21 18:34</td>
<td>etc</td>
</tr>
<tr>
<td>drwxr-x-x</td>
<td>3 root</td>
<td>root</td>
<td>0</td>
<td>Jan 1 1970</td>
<td>flashS3</td>
</tr>
<tr>
<td>drwxr-x-x</td>
<td>2 root</td>
<td>root</td>
<td>0</td>
<td>Nov 21 18:34</td>
<td>fornfs</td>
</tr>
<tr>
<td>drwxr-x-x</td>
<td>3 root</td>
<td>root</td>
<td>0</td>
<td>Nov 21 18:34</td>
<td>lib</td>
</tr>
<tr>
<td>drwxr-x-x</td>
<td>2 root</td>
<td>root</td>
<td>0</td>
<td>Nov 21 18:34</td>
<td>mnt</td>
</tr>
<tr>
<td>dr-x-x-x-x</td>
<td>35 root</td>
<td>root</td>
<td>0</td>
<td>Nov 21 18:34</td>
<td>proc</td>
</tr>
<tr>
<td>drwxrwx--</td>
<td>4 root</td>
<td>root</td>
<td>80</td>
<td>Nov 21 18:34</td>
<td>ramS3</td>
</tr>
<tr>
<td>drwx------</td>
<td>2 root</td>
<td>root</td>
<td>0</td>
<td>Nov 21 18:34</td>
<td>root</td>
</tr>
<tr>
<td>drwxr-x-x</td>
<td>3 root</td>
<td>root</td>
<td>0</td>
<td>Nov 21 18:34</td>
<td>sbin</td>
</tr>
<tr>
<td>drwxr-x-x</td>
<td>10 root</td>
<td>root</td>
<td>0</td>
<td>Nov 21 18:34</td>
<td>sys</td>
</tr>
<tr>
<td>drwxrwxrwt</td>
<td>2 root</td>
<td>root</td>
<td>60</td>
<td>Nov 21 18:34</td>
<td>tmp</td>
</tr>
<tr>
<td>drwxr-x-x</td>
<td>4 root</td>
<td>root</td>
<td>0</td>
<td>Nov 21 18:34</td>
<td>usr</td>
</tr>
</tbody>
</table>

```

> ./sampleHostApp 0 'ls -l bin'
```

<table>
<thead>
<tr>
<th>Permissions</th>
<th>Owner</th>
<th>Group</th>
<th>Size</th>
<th>Date/Time</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>550636</td>
<td>Nov 21 19:09</td>
<td>ash -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>cat -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>chgrp -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>chmod -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>chown -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>cp -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>date -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>df -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>dmesg -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>echo -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>false -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>grep -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>kill -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>ln -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>ls -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>md5sum -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>mkdir -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>mkdir -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>mount -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>mv -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>ping -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>ps -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>pwd -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>rm -&gt; busybox</td>
</tr>
<tr>
<td>lrwxrwxrwx</td>
<td>1 root</td>
<td>root</td>
<td>7</td>
<td>Nov 21 19:09</td>
<td>rmdir -&gt; busybox</td>
</tr>
</tbody>
</table>
> ./sampleHostApp 0 'ls -l sbin'
-r-xr-x--- 1 root root 26508 Nov 21 19:09 badblocks
lrwxrwxrwx 1 root root 9 Nov 21 19:09 e2fsck -> fsck.ext3
lrwxrwxrwx 1 root root 9 Nov 21 19:09 fsck -> fsck.ext3
-r-xr-x--- 1 root root 195856 Nov 21 19:09 fsck.ext3
-r-xr-x--- 1 root root 11540 Nov 21 19:09 hexpr
lrwxrwxrwx 1 root root 14 Nov 21 19:09 ifconfig -> ../bin/busybox
lrwxrwxrwx 1 root root 14 Nov 21 19:09 ifdown -> ../bin/busybox
lrwxrwxrwx 1 root root 14 Nov 21 19:09 ifup -> ../bin/busybox
lrwxrwxrwx 1 root root 14 Nov 21 19:09 init -> ../bin/busybox
lrwxrwxrwx 1 root root 14 Nov 21 19:09 insmod -> ../bin/busybox
lrwxrwxrwx 1 root root 14 Nov 21 19:09 klogd -> ../bin/busybox
lrwxrwxrwx 1 root root 14 Nov 21 19:09 lsmod -> ../bin/busybox
lrwxrwxrwx 1 root root 9 Nov 21 19:09 mke2fs -> mkfs.ext3
lrwxrwxrwx 1 root root 9 Nov 21 19:09 mkfs.ext2 -> mkfs.ext3
-r-xr-x--- 1 root root 47348 Nov 21 19:09 mkfs.ext3
lrwxrwxrwx 1 root root 14 Nov 21 19:09 modprobe -> ../bin/busybox
lrwxrwxrwx 1 root root 14 Nov 21 19:09 reboot -> ../bin/busybox
lrwxrwxrwx 1 root root 14 Nov 21 19:09 rmmod -> ../bin/busybox
lrwxrwxrwx 1 root root 14 Nov 21 19:09 route -> ../bin/busybox
-r-xr-x--- 1 root root 32832 Nov 21 19:09 tune2fs
lrwxrwxrwx 1 root root 14 Nov 21 19:09 watchdog -> ../bin/busybox

> ./sampleHostApp 0 'ls -l usr/bin'
lrwxrwxrwx 1 root root 17 Nov 21 19:09 [ -> ../../bin/busybox
lrwxrwxrwx 1 root root 17 Nov 21 19:09 find -> ../../bin/busybox
lrwxrwxrwx 1 root root 17 Nov 21 19:09 id -> ../../bin/busybox
lrwxrwxrwx 1 root root 17 Nov 21 19:09 killall -> ../../bin/busybox
lrwxrwxrwx 1 root root 17 Nov 21 19:09 test -> ../../bin/busybox

> ./sampleHostApp 0 'ls -l usr/sbin'
lrwxrwxrwx 1 root root 17 Nov 21 19:09 traceroute -> ../../bin/busybox

54 IBM 4765 Toolkit Guide
9 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.

9.1 Coprocessor memory segments and security

The design for the IBM 4765 PCIe Cryptographic 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 4765 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.

56 IBM 4765 Toolkit Guide
Miniboot changes the state of a segment in response to certain commands Miniboot receives from the host. Figure 9 on page 57 shows the state transitions for segment 2, and Figure 10 on page 58 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 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.

Figure 9 State transitions for segment 2

* This transition occurs if the trust arguments associated with segment 2 indicate the new segment 1 is not trusted. See “Trust and countersignature arguments” on page 72 for details.

4 Or segment 2 if the developer is writing an operating system for the coprocessor.
9.2 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 4765 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.

### 9.3 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.

### 9.4 The 64-bit Signer and Packager utilities

Toolkit versions 4.4.20 and later include a 64-bit Signer and Packager. These versions operate differently than the 32-bit versions, but produce equivalent output. Outputs that are signed with 32-bit Signer can be loaded with 64-bit CLU, and outputs that are signed with 64-bit Signer can be loaded with 32-bit CLU.

These utilities 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.

The main difference from a developer’s perspective is that the 64-bit versions of Signer and Packager use parameter=value pairs for inputs. 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:

```plaintext
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 the 64-bit version of Signer and Packager, please review the signing64/cru directories in the Toolkit.

Under no circumstances should files signed with the keys provided with this sample be used in a production environment.
9.4.1 Initializing CCA for use with Signer

Before running Signer / Packager, CCA or a UDX must be loaded on the 4765 and the 4765 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 4765 as a signing station. For production signing, additional configuration of the 4765 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 restrict these operations:

0331: Allow RSA Key Tokens to Be Wrapped by Weaker DES KEK
0332: Warn When Wrapping Key with Weaker Master Key
0333: Disallow Wrapping Key with Weaker Master Key

9.4.2 Using Signer

The 64-bit Signer and Packager utilities are in the signer64/cru directory in the Toolkit. This directory's structure looks like this:

<table>
<thead>
<tr>
<th>Directory / File</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>cru</td>
<td>Top-level directory for CRU scripts. Contains script and configuration files needed to sign and/or package custom images for test and/or production.</td>
</tr>
<tr>
<td>cru/packager</td>
<td>Contains scripts that demonstrate how to package signed images into a single loadable binary.</td>
</tr>
<tr>
<td>cru/signer</td>
<td>Top-level directory for all Toolkit signing-related tasks.</td>
</tr>
<tr>
<td>cru/signer/include</td>
<td>Contains shared defines and subroutines used by the various signing and packaging scripts.</td>
</tr>
<tr>
<td>cru/signer/tasks</td>
<td>Contains directories for each of the Signer tasks necessary to release a Toolkit application.</td>
</tr>
<tr>
<td>test_key_manifest.inc</td>
<td>Contains a list of tag/value pairs for keys used in the signing process. These tags allow us to reference keys by variable names instead of by their fully-qualified names.</td>
</tr>
<tr>
<td>signer_parameter_tags.inc</td>
<td>Contains a list of tag/value pairs that describe the various tags expected by Signer and Packager.</td>
</tr>
<tr>
<td>common_signer_functions.inc</td>
<td>Contains various script subroutines used by other scripts.</td>
</tr>
<tr>
<td>common_segment_information.inc</td>
<td>Contains information about the description fields used as input to various Signer and Packager scripts.</td>
</tr>
</tbody>
</table>

The Toolkit includes script and include files that make it easy to configure and run the 64-bit 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 4765 with CCA (or a UDX) as a signing station, as discussed in Initializing CCA for use with Signer on page 60, you can use the Signer utility. Perform the following tasks to create sample outputs:

```
export Y4TK_FS_ROOT=<fully-qualified path>/y4tk/<current Toolkit version>
cd $Y4TK_FS_ROOT/signing_x64/cru
source setenv.sh
./process_toolkit_signing_request.sh
```

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

### 9.4.3 Using Packager

#### 9.5 The Signer utility

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.

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 `y4tk/<version>/bin/host32/linux` directory. On AIX, crusignr is in the `y4tk/<version>/bin/host/aix` directory.

**Syntax**

32-Bit Signer:

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

64-Bit Signer:

```
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 a 4765 application” on page 79 for a list of operations CRUSIGNR must perform and Chapter 6, “Secured Code-Signing Node” of the CCA Support Program Installation Manual for roles and profiles that may be of interest.

The first form 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

---

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 y4tk/bin/host32/linux).
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.

The second form causes CRUSIGNR to read arguments from the command line. Once the command line is exhausted, CRUSIGNR issues a prompt for each additional argument required and reads the argument from stdin. This method is generally discouraged for anything other than testing, as CRUSIGNR requires up to dozens of arguments.

If CRUSIGNR reads an argument from stdin, you may select the default for the argument (if there is one) by entering a null line (that is, by pressing the Enter key when prompted for the argument).

32-bit CRUSIGNR writes messages to a file named $SIGNER.RSP. 64-bit CRUSIGNR has improved logging functionality and writes all messages to stdout, 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.

9.5.1 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:

9.5.1.1 Signer cryptographic functions

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEYGEN</td>
<td>Generate an RSA key pair.</td>
</tr>
<tr>
<td>HASH_GEN</td>
<td>Generate the hash for a file using the SHA1 algorithm.</td>
</tr>
<tr>
<td>HASH_VER</td>
<td>Verify the hash of a file using the SHA1 algorithm.</td>
</tr>
</tbody>
</table>

9.5.1.2 Signer miniboot command functions

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMBURN3</td>
<td>Load software into segment 3.</td>
</tr>
<tr>
<td>REMBURN3</td>
<td>Replace the software in segment 3.</td>
</tr>
<tr>
<td>SUROWN3</td>
<td>Surrender ownership of segment 3.</td>
</tr>
</tbody>
</table>

9.5.1.3 Signer miscellaneous functions

<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>HELP</td>
<td>Display instructions about how to use the program.</td>
</tr>
</tbody>
</table>
9.5.1.4 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
EMBURN2
ESIG2
ESTOWN2
FCVCERT
KEYCERT
REMBURN1
REMBURN2
SIGNFILE
SUROWN2

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.

9.5.2 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.
9.5.3 EMBURN3 - Load software into segment 3

Syntax

32-Bit Signer:

EMBURN3 out_fn filedesc_args sigkey_args image_args key_fn esig_fn
   seg2_ownerid seg3_ownerid trust1_fl trust2_fl type2_target_args

64-Bit Signer:

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:

- `out_fn` (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 TSK.
- `filedesc_args` (EMBURN3_PART_NUMBER, EMBURN3_EC_NUMBER, EMBURN3_DESCRIPTIVE_TEXT) provides certain descriptive information that is incorporated into the output file. See “File description arguments” on page 71 for details.
- `sigkey_args` (EMBURN3_SIGNING_KEY_CERTIFICATE_FILENAME and EMBURN3_PRIVATE_SIGNING_KEY_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 71 for details.
- `image_args` (EMBURN3_ADAPTER_FAMILY_TARGET, EMBURN3_SEGMENT_IMAGE_TITLE, and
- EMBURN3_SEGMENT_REVISION_NUMBER) specifies 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 provides certain descriptive information about the image that is also downloaded to the coprocessor. See “Image file arguments” on page 72 for details.

- key_fn (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 RSA keypair. 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. This key is downloaded to the coprocessor and is used 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 esig_fn 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.

- esig_fn (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 key_fn 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.

- seg2_owid (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.

- seg3_owid (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.

- trust1_fl (defaults) indicates whether or not segment 3's state is to be changed to UNOWNED if the contents of segment 1 change. This flag is downloaded to the coprocessor. See “Trust and countersignature arguments” on page 72 for details.

- trust2_fl (Defaults) indicates whether or not segment 3's state is to be changed to UNOWNED if the contents of segment 2 change. This flag is downloaded to the coprocessor. See “Trust and countersignature arguments” on page 72 for details.

- type2_target_args (defaults) specifies certain conditions that the coprocessor checks before it accepts the new segment 3 information. See “Targeting arguments” on page 73 for details.

9.5.4 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.

9.5.5 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

---

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

9.5.6 HASH_GEN - Generate hash for file

Syntax

32-Bit Signer:

HASH_GEN in_fn out_fn

64-Bit Signer:

HASHGEN_INPUT_FILENAME_TO_HASH=<value>
HASHGEN_OUTPUT_FILENAME=<value>
HASHGEN_HASH_ALGORITHM="SHA-256 "

HASH_GEN uses the SHA256 algorithm to generate a hash for the file in_fn and writes the result to the file out_fn. The output file consists of groups of four characters representing hexadecimal digits separated by blanks (for example, 03A2 8989 BD90 FFED 0078).

in_fn (HASHGEN_INPUT_FILENAME_TO_HASH) and out_fn (HASHGEN_OUTPUT_FILENAME) must include path information if either file is not in the current directory. When using 64-bit CRUSIGNR, HASHGEN_HASH_ALGORITHM must be “SHA-256 “ (note the extra space at the end).

9.5.7 HASH_VER - Verify hash of file

Syntax

32-Bit Signer:

HASH_VER data_fn hash_fn

64-Bit Signer:

HASHVER_INPUT_HASH_TO_VERIFY_FILENAME=<value>
HASHVER_INPUT_FILENAME_TO_VERIFY=<value>
HASHVER_HASH_ALGORITHM="SHA-256 

HASH_VER verifies that the hash in the file hash_fn (HASHVER_INPUT_HASH_TO_VERIFY_FILENAME) matches the hash the HASH_GEN function would generate given data_fn (HASHVER_INPUT_FILENAME_TO_VERIFY) as input and issues a message indicating the result. The hash_fn file has the same format as the out_fn file generated by the
HASH_GEN function. When using 64-bit CRUSIGNR, HASHVER_HASH_ALGORITHM must be “SHA-256” (note the extra space at the end).

hash_fn and data_fn must include path information if either file is not in the current directory.

9.5.8 KEYGEN - Generate RSA key pair

Syntax

32-Bit Signer:

KEYGEN {0 | 2} keypair_fn pubkey_fn skeleton_fn
KEYGEN 1 keypair_fn pubkey_fn skeleton_fn transkey_fn
KEYGEN 3 pubkey_fn skeleton_fn {0 | 1}

64-Bit Signer:

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 RSA keypair and saves it in the file keypair_fn (KEYGEN_PRIVATE_KEY_OUTPUT_FILENAME). The public key is also saved in the file pubkey_fn (KEYGEN_PUBLIC_KEY_OUTPUT_FILENAME) and the hash of the public key is saved in a file with the same name as pubkey_fn and extension HSH. The file skeleton_fn (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 Developer's Toolkit. A developer can also generate customized skeleton files. The file transkey_fn (KEYGEN_TRANSPORT_KEYTOKEN_FILENAME) contains a DES IMPORTER or DES EXPORTER key-encrypting key.

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 first argument to KEYGEN when using 32-bit CRUSIGNR, or KEYGEN_PRIVATE_KEY_ENCRYPTION_MECHANISM on 64-bit CRUSIGNR 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 RSA keypair in keypair_fn with the coprocessor CCA master key before returning the keypair.
• 1 - Use XPORT for the rule_array parameter. This causes the coprocessor to encrypt the RSA keypair in keypair_fn with the key-encrypting key in transkey_fn before returning the keypair.
• 2 - Use CLEAR for the rule_array parameter. This causes the coprocessor to return the RSA keypair in keypair_fn "in the clear" (that is, the file is not encrypted).

7 The KEYGEN command computes the hash in the same manner and stores it in the same format as the HASH_GEN command.
3 - Use RETAIN for the `rule_array` parameter. This causes the coprocessor to retain the RSA keypair and not write it to the host. Specify 1 as the last argument if the retained key may be cloned and specify 0 if it may not.

Refer to the IBM 4765 PCIe CCA Basic Services Reference and Guide for details on the format of skeleton files and the CCA PKA_Key_Generate verb.

**9.5.9 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.

**9.5.10 REMBURN3 - Replace software in segment 3**

**Syntax**

32-Bit Signer:

```
REMBURN3 out_fn filedesc_args sigkey_args image_args pubkey_fn privkey_fn
   seg2_ownid seg3_ownid trust1_fl trust2_fl type3_target_args
```

64-Bit Signer:

```
REMBURN3_OUTPUT_FILENAME=<value>
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_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:
- **out_fn** (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 TSK.

- **filedesc_args** (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 71 for details.

- **sigkey_args** (REMBURN3_PRIVATE_SIGNING_KEY_FILENAME, REMBURN3_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 71 for details.

- **image_args** (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 72 for details.

- **pubkey_fn** (REMBURN3_SEGMENT_PUBLIC_KEY_FILENAME) is the name of the file that contains the public key to be associated with segment 3. 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.

- **privkey_fn** (REMBURN3_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 `privkey_fn` file. The coprocessor uses the public key associated with segment 3 to validate the hash and rejects the REMBURN3 command if the validation fails.

- **seg2_ownid** (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.

- **seg3_ownid** (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.

- **trust1_fl** (defaults) indicates whether or not segment 3's state is to be changed to UNOWNED if the contents of segment 1 change. See “Trust and countersignature arguments” on page 72 for details.

- **trust2_fl** (defaults) indicates whether or not segment 3 should trust changes made to segment 2. If the trust flags are set to NEVER, segment 3's state is to be changed to UNOWNED if the contents of segment 2 change. See “Trust and countersignature arguments” on page 72 for details.

- **type3_target_args** (defaults) specifies certain conditions that the coprocessor checks before it accepts the new segment 3 information. See “Targeting arguments” on page 73 for details.

---

**9.5.11 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. 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 If desired, the new public key may be the same as the public key currently associated with the segment.

9 This also changes segment 3’s state to UNOWNED.
SUROWN3 - Surrender Ownership of Segment 3

Syntax

32-Bit Signer:

SUROWN3 out_fn filedesc_args sigkey_args privkey_fn seg2_ownerid
    seg3_ownerid type3_target_args

64-Bit Signer:

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:

- **out_fn** (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 TSK.
- **filedesc_args** (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 71 for details.
- **sigkey_args** (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 71 for details.
- **privkey_fn** (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 privkey_fn file. The coprocessor
  uses the public key associated with segment 3 to validate the hash and rejects the SUROWN3
  command if the validation fails.
- **seg2_ownerid** (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.
• \texttt{seg3\_ownid} (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.

• \texttt{type3\_target\_args} specifies certain conditions that the coprocessor checks before it accepts the SUROWN3 command. See “Targeting arguments” on page 73 for details.

9.5.13 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:

\texttt{partnumber ECnumber description}

where

• \texttt{partnumber} (\texttt{<\text{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.

• \texttt{ECnumber} (\texttt{<\text{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.

• \texttt{description} (\texttt{<\text{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.

9.5.14 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).\footnote{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.} The format of these arguments is

\texttt{sigkey\_cert\_fn sigkey\_fn}

where

• \texttt{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.

• \texttt{sigkey\_fn} is the name of the file containing the RSA private key to be used to sign the output file. Path information must also be provided if the file is not in the current directory.

When CRUSIGNR creates an output file containing a Miniboot command, CRUSIGNR incorporates the certificate from the \texttt{sigkey\_cert\_fn} file, computes a hash of the output file, encrypts the hash with the private key in the \texttt{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\footnote{The public key is compiled into CLU.}, uses the extracted key to decrypt the hash, and verifies that the two hash values match.

\texttt{Using Signer and Packager 71}
9.5.15 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:

\[\text{image\_fn family title revision}\]

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 4765 are:
  - 4 for code that targets the IBM 4765 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. When the image file is a segment 3 image which is to be run on a PCIe Cryptographic Coprocessor, the CRUSIGNR utility enforces certain restrictions on the title argument: CRUSIGNR may insert the characters "UDX" before the first three characters of the title; bytes 37-48 of the title string will be overlaid with the timestamp of the BIN file used to create the image.
- **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 73 for details.

9.5.16 Trust and countersignature arguments

Recall that one of the primary design goals for the IBM 4765 PCIe Cryptographic Coprocessor was to ensure that software in the coprocessor must not run or accumulate state unless the environment in which it runs is trustworthy. The use of digital signatures ensures that changes to a segment are authorized (hence trusted) by segments with greater privilege (for example, the initial load of segment 3 must be authorized by the owner of segment 2). But trust operates both ways: changes to a segment that are not trusted by a segment with lesser privilege cause the state of the segment with lesser privilege to become unrunnable (for example, untrusted changes to segment 2 make segment 3 unrunnable).

The CRUSIGNR functions that replace the contents of a segment (EMBURN2, EMBURN3, REMBURN2, and REMBURN3) include a flag that indicates how the coprocessor is to change the state of the segment if the contents of a more privileged segment change as a result of a REMBURN command. (Changes caused by an EMBURN command are always untrusted.) See “Coprocessor memory segments and security” on page 55 for details on segment states. The flag may be 1 (always trust the new more privileged segment), 2 (never trust the new more privileged segment), or 3 (trust the new more privileged segment only if it is countersigned).

If a segment S specifies a trust flag of 1 with respect to a more privileged segment T, S always trusts changes to T. A REMBURN command that changes the contents of T does not affect the state of S.

If a segment S specifies a trust flag of 2 with respect to a more privileged segment T, S never trusts changes to T. A REMBURN command that changes the contents of T changes the state of S to RELIABLE\_BUT\_UNRUNNABLE or to UNOWNED. Note that an EMBURN command that changes the contents of T causes the state of S to change in this manner regardless of the value of the trust flag.
If a segment S specifies a trust flag of 3 with respect to a more privileged segment T, S trusts changes to T only if the new image of T is countersigned with the private key corresponding to the public key associated with S. The coprocessor validates the countersignature and changes the state of S to RELIABLE_BUT_UNRUNNABLE or to UNOWNED if the countersignature is incorrect.

REMBURN commands that affect segments other than segment 3 (for example, REMBURN2) must therefore include arguments to supply a countersignature. The use of countersignatures has been deprecated. As such, all countersignature related arguments should specify that no countersignatures are present. The format of the countersignature arguments is

{NoCSig3}

where

- **NoCSig3** indicates there is no countersignature provided by segment 3. This option applies to the REMBURN3 command and must be specified exactly as shown (that is, case is important).

### 9.5.17 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

```text
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. 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 **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.

---

12 That is, the value xcGetConfig returns in pInfo->AdapterID. Refer to the IBM 4765 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 RTCid and 0 for RTCid_mask.

- **VPDserno** and **VPDserno_mask** specify a range of permitted values for the coprocessor’s IBM serial number.\(^\text{13}\) 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 **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.

  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.\(^\text{14}\) 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.

\(^{13}\) That is, the value xcGetConfig returns in pInfo->VPD.sn. Refer to the IBM 4765 PCIe Cryptographic Coprocessor Custom Software Interface Reference for details.

\(^{14}\) That is, the value xcGetConfig returns in pInfo->VPD.pn. Refer to the IBM 4765 PCIe Cryptographic Coprocessor Custom Software Interface Reference for details.
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.

- VPflags and VPflags_mask specify a range of permitted values for the coprocessor's VPD flags. 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 VPflags_mask must be either ASCII 0 or ASCII 1. CRUSIGNR uses VPflags_mask to construct a 32-byte hexadecimal number. Each byte in the hexadecimal number is set to 0xFF if the corresponding character in VPflags_mask is ASCII 1 and is set to 0x00 otherwise.

CRUSIGNR logically ANDs VPflags with the hexadecimal number derived from VPflags_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 VPflags_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 VPflags and 0 for VPflags_mask.

- bootcount_fl, bootcount_left, and bootcount_right are used as follows: each time the coprocessor boots, it increments one of two counters. The "left count" is a 16-bit number kept in EEPROM that is zero when the coprocessor leaves the factory and is incremented each time the coprocessor boots in a zeroized state (that is, each time the coprocessor is revived after having cleared memory upon detecting an attempt to compromise the coprocessor's security). 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.

bootcount_fl may be 0, 1, or 2. If bootcount_fl is 0, bootcount_left and bootcount_right do not appear and the Miniboot command that incorporates the targeting information is accepted regardless of the left and right counts.

If bootcount_fl is 1, bootcount_left is compared to the left count. The Miniboot command that incorporates the targeting information is rejected if the left count is greater than bootcount_left. bootcount_left must be between 0 and 65535, inclusive, and bootcount_right does not appear in this case.

---

15 That is, the value xcGetConfig returns in the last sixteen bytes of pInfo->VPD.reserved. Refer to the IBM 4765 PCIe Cryptographic Coprocessor Custom Software Interface Reference for details.

16 Every boot increments either the left count or the right count, so the full 48-bit boot count always increases with each boot. If incrementing either the left count or the right count would cause the counter to overflow, the boot process halts in error.

17 The DRUID utility displays the current left and right counts each time it is run.
If `bootcount_fl` is 2, `bootcount_left` is compared to the left count and `bootcount_right` is compared to the right count. The Miniboot command that incorporates the targeting information is rejected if the left count is greater than `bootcount_left` or if the left count is equal to `bootcount_left` and the right count is greater than `bootcount_right`. Use of both counts in this manner can create a Miniboot command that can be downloaded to the coprocessor only once. `bootcount_left` must be between 0 and 65535, inclusive, and `bootcount_right` must be between 0 and 4294967295, inclusive, in this case.

If a command is intended to apply to all possible coprocessors, specify 0 for `bootcount_fl` and omit `bootcount_left` and `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 72 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 72 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.

### 9.6 The Packager utility

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.\(^\text{18}\)

#### Syntax

**32-Bit Packager:**

```bash
CRUPKGR [-S profile] -F parm_file_name
CRUPKGR [-S profile] [sigkey_args [num_files [in_fn_list [out_fn
[ outtype [filedesc_args]]]]]]
CRUPKGR -H
```

\(^\text{18}\) 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 y4tk/bin/host32/linux).
64-Bit Packager:

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 79 for a list of operations CRUPKGR must perform and Chapter 6, “Secured Code-Signing Node” of the CCA Support Program Installation Manual for roles and profiles that may be of interest.

The first form causes CRUPKGR 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 appears on a separate line. Once the file is exhausted, CRUPKGR issues a prompt for each additional argument required and reads the argument from stdin.

The third form (available only with 32-bit PACKAGER) causes CRUPKGR to read arguments from the command line. Once the command line is exhausted, CRUPKGR issues a prompt for each additional argument required and reads the argument from stdin. This invocation is discouraged for anything other than testing.

If CRUPKGR reads an argument from stdin, you may select the default for the argument (if there is one) by entering a null line (that is, by pressing the Enter key when prompted for the argument).

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.

CRUPKGR takes the following arguments:

- `sigkey_args` (PACKAGER_OBJECT_SIGNING_KEY_PAIR, PACKAGER_OBJECT_SIGNING_KEY_CERTIFICATE) specifies the RSA private key that CRUPKGR 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 71 for details.
- `num_files` (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.
- `in_fn_list` (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.
- `out_fn` (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`.
- `outtype` (PACKAGER_DISK_IMAGE_ID) specifies how the output file is intended to be used. Recognized values are as follows:
  - 4 for segment 3

78 IBM 4765 Toolkit Guide
• 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 outtype 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.

• filedesc_args (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 71 for details.

9.7 Summary of steps to package and release a 4765 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 IBM 4765 PCIe Cryptographic Coprocessor CCA Support Program Installation Manual.

2. Generate three RSA keypairs using CRUSIGNR’s KEYGEN function:

32-bit Signer:

```
CRUSIGNR KEYGEN 2 S3KDEVPP.KEY S3KDEVPU.KEY
~/y4tk/<version>/signing/default_4096crtSkeleton.tkn
CRUSIGNR KEYGEN 2 DEVSGNPP.KEY DEVSGNPU.KEY
~/y4tk/<version>/signing/default_4096crtSkeleton.tkn
CRUSIGNR KEYGEN 2 DEVPKGPP.KEY DEVPKGPU.KEY
~/y4tk/<version>/signing/default_4096crtSkeleton.tkn
```

Note: when parameter “2” is passed to CRUSIGNR KEYGEN, this indicates that the keys will be generated in the clear. You may wish to use parameter “0” to encrypt the keys under a master key (which means you will also need to securely store the master key).

64-Bit Signer:

```
export Y4TK_FS_ROOT=<full path to>/y4tk/<version>/signing64/cru
```

cd $Y4TK_FS_ROOT/signtu

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 second argument, for example:

```
CRUSIGNR KEYGEN 0 S3KDEVPP.KEY S3DEVPU.KEY
~/y4tk/<version>/signing/default_4096crtSkeleton.tkn
```

The appropriate actions should be taken to ensure the master key can be regenerated should the need arise. Refer to the IBM 4765 PCIe Cryptographic Coprocessor CCA Support Program Installation Manual for details.

Using Signer and Packager 79
source setenv
cd signer/tasks/keygen
./create_4765_toolkit_signing_keys.sh test segment3_key_oid6.inc
./create_4765_toolkit_signing_keys.sh test
segment3_toolkit_signing_key.inc
./create_4765_toolkit_signing_keys.sh test
segment3_toolkit_packaging_key.inc

The public key files are S3KDEVPU.KEY, DEVSGNPU.KEY, and DEVPKGPU.KEY. The corresponding hash files are SDKDEVPU.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 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 Developer’s 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. Note: rjm in these filenames refers to release r major revision j minor revision m of the files.

   - establish_ownership_seg2_toolkit_OID243_rjm.clu, which establishes ownership of segment 2. Segment 2 must be owned before an application or an operating system can be loaded into the coprocessor. This file is shipped with the IBM 4765 PCIe Cryptographic Coprocessor Developer's Toolkit. Note: on xCrypto, this file is es2RRRSS.e2t.

   - emergency_reload_seg2_toolkit_OID243_rjm.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 IBM 4765 PCIe Cryptographic Coprocessor Developer's Toolkit. Note: on xCrypto, this file is eb2RRRSS.l2t.

   - reload_seg2_toolkit_OID243_rjm.clu, which replaces an existing coprocessor operating system in segment 2. This file is shipped with the IBM 4765 PCIe Cryptographic Coprocessor Developer's Toolkit. Note: on xCrypto, this file is

---

20 IBM typically provides a form for this purpose that can be returned by way of fax.
21 See Using Signer and Packager for details on the contents of these files.
22 The owner identifier assigned to segment 2 (typically 243 \[0xF3\]).
rb2RRRSS.r2t.

- surrender_ownership_seg2_toolkit OID243_rjm.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. This file is shipped with the IBM 4765 PCIe Cryptographic Coprocessor Developer’s Toolkit. Note: on xCrypto, this file is sr2rrrss.s2t.

- ESTOWN3.E3T, which establishes ownership of segment 3. This file is provided by IBM as part of the signing process. IBM assigns the developer an owner identifier, and ESTOWN3.E3T saves that value in the coprocessor. Segment 3 must be owned before an application can be loaded into the coprocessor. Note: on xCrypto, this file is ESTOWN3.E3T.

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:

32-Bit Signer:

```
CRUSIGNR EMBURN3 MYAPP.L3T
part version description
DEVSGNPU.CRT DEVSGNPP.KEY
SEG3TK.BIN 4 title revision
S3KDEVPP.KEY ESIGDEV.SIG
ibm2 oem3
1 1
```

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 sr2rrrss.s2t removes any custom application installed on a coprocessor regardless of the application’s origin.

That is, an OEM or organization within an OEM.
where part, version, and description supply information that is incorporated into the output file, title and revision supply information that is downloaded to the coprocessor and stored with the application in segment 3, ibm2 is the owner identifier for segment 2, and oem3 is the owner identifier assigned to the developer. See “Using Signer and Packager” on page 55 for details.

Note: Automated examples for the creation of an EMBURN3 image using 32-bit Signer can be found in the makefiles in the y4tk/<version>/signing32 directory. A corresponding example for use with 64-bit Signer can be found in y4tk/<version>/signing64/cru/signer/tasks/emburn3. Please refer to the readmes or 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, for example:

```
CRUSIGNR REMBURN3 MYAPP.R3T
part version description
DEVSGNPU.CRT DEVSGNPP.KEY
SEG3TK.BIN 4 title revision
S3KDEVPU.KEY S3KDEVPP.KEY
ibm2 oem3
1 1
a 0 b 0 c 0 d 0 e 0 0
x 0 0 65535 N
x 0 0 65535 N
```

Note: Automated examples for the creation of an REMBURN3 image using 32-bit Signer can be found in the makefiles in the y4tk/<version>/signing32 directory. A corresponding example for use with 64-bit Signer can be found in y4tk/<version>/signing64/cru/signer/tasks/remburn3. Please refer to the readmes or script prologues in each directory for more information.

### 9.8 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:

- **CRUSIGNR KEYGEN 0**
  - 0103 PKA Key Generate

- **CRUSIGNR KEYGEN 2**
  - 0103 PKA Key Generate
  - 0205 PKA Clear Key Generate

---

25 Typically 243 (0xF3).
26 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.
CRUSIGNR EMBURN3/REMBURN3; CRUPKGR
- 0100 Digital Signature Generate
- 0101 Digital Signature Verify
- 0104 PKA Key Import (if the key used in the operation was generated in the clear)
- 0107 PKA96 One Way Hash
10 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.

10.1 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.

10.2 Trademarks

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

- AIX
- IBM

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

Other company, product, and service names may be trademarks or service marks of others.
## 11 List of abbreviations and acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
<td>PCIe</td>
<td>Peripheral component interconnect express</td>
</tr>
<tr>
<td>API</td>
<td>Application program interface</td>
<td>PCI-X</td>
<td>peripheral component interconnect extended</td>
</tr>
<tr>
<td>ASCII</td>
<td>American National Standard Code for Information Exchange</td>
<td>PDF</td>
<td>portable document format</td>
</tr>
<tr>
<td>BBRAM</td>
<td>battery-backed random access memory</td>
<td>PIN</td>
<td>personal identification number</td>
</tr>
<tr>
<td>CCA</td>
<td>Common Cryptographic Architecture</td>
<td>PKA</td>
<td>public key algorithm</td>
</tr>
<tr>
<td>CLU</td>
<td>Coprocessor Load Utility</td>
<td>RNG</td>
<td>random number generator</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard</td>
<td>RSA</td>
<td>Rivest-Shamir-Adleman (algorithm)</td>
</tr>
<tr>
<td>EPROM</td>
<td>Erasable Programmable Read-Only Memory</td>
<td>SHA</td>
<td>Secure Hash Algorithm</td>
</tr>
<tr>
<td>FIPS</td>
<td>Federal Information Processing Standard</td>
<td>SLES</td>
<td>SUSE Linux Enterprise Server</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines Corporation</td>
<td>SRDI</td>
<td>Security relevant data item</td>
</tr>
<tr>
<td>ICAT</td>
<td>Interactive Code Analysis Tool</td>
<td>TOD</td>
<td>time-of-day (clock)</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/output</td>
<td>UART</td>
<td>universal asynchronous receiver/transmitters</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
<td>UDX</td>
<td>user-defined extension</td>
</tr>
<tr>
<td>MD5</td>
<td>Message digest 5 (hashing algorithm)</td>
<td>VPD</td>
<td>vital product data</td>
</tr>
<tr>
<td>PCI</td>
<td>peripheral component interconnect</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
12 Glossary

This glossary includes terms and definitions from the IBM Dictionary of Computing, New York: McGraw Hill, 1994. This glossary also includes terms and definitions taken from:

- The American National Standard Dictionary for Information Systems, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42 Street, New York, New York 10036. Definitions are identified by the symbol (A) following the definition.

- The Information Technology Vocabulary, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) following the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) following the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

A

access. In computer security, a specific type of interaction between a subject and an object that results in the flow of information from one to the other.

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

access method. A technique for moving data between main storage and input/output devices.

adapter. Synonym for expansion card.

agent. (1) An application that runs within the IBM 4765 PCIe Cryptographic Coprocessor. (2) Synonym for secure cryptographic coprocessor application.

American National Standard Code for Information Interchange (ASCII). The standard code, using a coded character set consisting of seven-bit characters (eight bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. (A)

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

ANSI. American National Standards Institute.

API. Application program interface.

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


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

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

authorize. To permit or give authority to a user to communicate with or make use of an object, resource, or function.
battery-backed random access memory (BBRAM). Random access memory that uses battery power to retain data while the system is powered off. The IBM 4765 PCIe Cryptographic Coprocessor uses BBRAM to store persistent data for IBM 4765 applications, as well as the coprocessor device key.

BBRAM. Battery-backed random access memory.

bus. In a processor, a physical facility along which data is transferred.

call. The action of bringing a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and jumping to an entry point. (1) (A)

card. (1) An electronic circuit board that is plugged into an expansion slot of a system unit. (2) A plug-in circuit assembly. (3) See also expansion card.

CCA. Common Cryptographic Architecture.

ciphertext. (1) Data that has been altered by any cryptographic process. (2) See also plaintext.

cleartext. (1) Data that has not been altered by any cryptographic process. (2) Synonym for plaintext. (3) See also ciphertext.

CLU. Coprocessor Load Utility.

Common Cryptographic Architecture (CCA). A comprehensive set of cryptographic services that furnishes a consistent approach to cryptography on major IBM computing platforms. Application programs can access these services through the CCA application program interface.

Common Cryptographic Architecture (CCA) API. The application program interface used to call Common Cryptographic Architecture functions; it is described in the IBM 4765 CCA Basic Services Reference and Guide.

coprocessor. (1) A supplementary processor that performs operations in conjunction with another processor. (2) A microprocessor on an expansion card that extends the address range of the processor in the host system, or adds specialized instructions to handle a particular category of operations; for example, an I/O coprocessor, math coprocessor, or a network coprocessor.

Coprocessor Load Utility (CLU). A program used to load validated code into the IBM 4765 PCIe Cryptographic Coprocessor.

Cryptographic Coprocessor (IBM 4765). An expansion card that provides a comprehensive set of cryptographic functions to a workstation.

cryptographic node. A node that provides cryptographic services such as key generation and digital signature support.

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

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

Data Encryption Standard Manager (DES_Mgr). A Linux extension that manages the IBM 4765 PCIe Cryptographic Coprocessor DES processing hardware.

decipher. (1) To convert enciphered data into clear data. (2) Contrast with encipher.

DES_Mgr. Data Encryption Standard Manager.

device driver. (1) A file that contains the code needed to use an attached device. (2) A program that enables a computer to communicate with a specific peripheral device; for example, a printer, videodisc player, or a CD drive.
**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.

**expansion board.** Synonym for expansion card.

**expansion card.** A circuit board that a user can plug into an expansion slot to add memory or special features to a computer.

**expansion slot.** One of several receptacles available inside a computer into which a user can install an expansion card.

**F**

**feature.** A part of an IBM product that can be ordered separately from the essential components of the product.

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

**FIPS.** Federal Information Processing Standard.

**flash memory.** A specialized version of erasable programmable read-only memory (EPROM) commonly used to store code in small computers.

**H**

**hertz (Hz).** A unit of frequency equal to one cycle per second. Note: In the United States, line frequency is 60 Hz, a change in voltage polarity 120 times per second; in Europe, line frequency is 50 Hz, a change in voltage polarity 100 times per second.

**host.** As regards to the IBM 4765 PCIe Cryptographic Coprocessor, the workstation into which the coprocessor is installed.

**ICAT.** Interactive Code Analysis Tool.

**inline code.** In a program, instructions that are executed sequentially without branching to routines, subroutines, or other programs.

**input/output (I/O).** (1) Pertaining to input, output, or both. (A) (2) Pertaining to a device, process, or channel involved in data input, data output, or both.

**Interactive Code Analysis Tool (ICAT).** A remote debugger used to debug applications running within the IBM 4765 PCIe Cryptographic Coprocessor.

**interface.** (1) A boundary shared by two functional units, as defined by functional characteristics, signal characteristics, or other characteristics as appropriate. The concept includes specification of the connection between two devices having different functions. (T) (2) Hardware, software, or both that links systems, programs, and devices.

**International Organization for Standardization (ISO).** An organization of national standards bodies established to promote the development of standards that facilitate the international exchange of goods and services; also, to foster cooperation in intellectual, scientific, technological, and economic activity.

**intrusion latch.** A software-monitored bit that can be triggered by an external switch connected to a jumper on the IBM 4765 PCIe Cryptographic Coprocessor. This latch can be used, for example, to detect when the cover of the coprocessor host workstation has been opened. The intrusion latch does not trigger the destruction of data stored within the coprocessor.

**I/O.** Input/output.

**IPL.** Initial program load.

**ISO.** International Organization for Standardization.
J

**jumper.** A wire that joins two unconnected circuits.

K

**key.** In computer security, a sequence of symbols used with an algorithm to encipher or decipher data.

M

**master key.** In computer security, the top-level key in a hierarchy of KEKs.

**miniboot.** Software within the IBM 4765 PCIe Cryptographic Coprocessor designed to initialize the operating system and to control updates to flash memory.

**multi-user environment.** A computer system that supports terminals and keyboards for more than one user at the same time.

N

**National Institute of Science and Technology (NIST).** Current name for the US National Bureau of Standards.

**NIST.** National Institute of Science and Technology.

**node.** (1) In a network, a point at which one or more functional units connects channels or data circuits. (I) (2) The endpoint of a link or junction common to two or more links in a network. Nodes can be processors, communication controllers, cluster controllers, or terminals. Nodes can vary in routing and other functional capabilities.

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.

**PCI.** Peripheral Component Interconnect.

**PCle.** Peripheral Component Interconnect Express.

**PCI-X.** Peripheral Component Interconnect eXtended.

**Peripheral Component Interconnect (PCI).** A 32-bit parallel computer expansion card standard.

**Peripheral Component Interconnect Express (PCle).** A high-speed serial connection computer expansion card standard that replaces the PCI and PCI-X standards.

**Peripheral Component Interconnect eXtended (PCI-x).** A 64-bit version of the PCI, utilized in the IBM 4764 Cryptographic Adapter.

**private key.** (1) In computer security, a key that is known only to the owner and used with a public key algorithm to decipher data. Data is deciphered using the related public key. (2) Contrast with public key. (3) See also public key algorithm.

**procedure call.** In programming languages, a language construct for invoking execution of a procedure. (I) A procedure call usually includes an entry name and the applicable parameters.

**public key.** (1) In computer security, a key that is widely known and used with a public key algorithm to encipher data. The enciphered data can be deciphered only with the related private key. (2) Contrast with private key. (3) See also public key algorithm.

**Public Key Algorithm Manager (PKA_Mgr).** A Linux extension that manages the IBM 4765 PCIe Cryptographic Coprocessor PKA processing hardware.

R

**reduced instruction set computer (RISC).** A computer that processes data quickly by using only a small, simplified instruction set.
return code. (1) A code used to influence the execution of succeeding instructions. (A) (2) A value returned to a program to indicate the results of an operation requested by that program.

RSA algorithm. A public key encryption algorithm developed by R. Rivest, A. Shamir, and L. Adleman.

S

security. The protection of data, system operations, and devices from accidental or intentional ruin, damage, or exposure.

system administrator. The person at a computer installation who designs, controls, and manages the use of the computer system.

Security Relevant Data Item (SRDI). Data that is securely stored by the IBM 4765 Cryptographic Adapter.

T

time-of-day (TOD) clock. A hardware feature that is incremented once every microsecond, and provides a consistent measure of elapsed time suitable for indicating date and time. The TOD clock runs regardless of whether the processing unit is in a running, wait, or stopped state.

throughput. (1) A measure of the amount of work performed by a computer system over a given period of time; for example, number of jobs-per-day. (A) (I) (2) A measure of the amount of information transmitted over a network in a given period of time; for example, a network data-transfer-rate is usually measured in bits-per-second.

TOD clock. Time-of-day clock.

U

utility program. A computer program in general support of computer processes. (T)

V

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

vital product data (VPD). A structured description of a device or program that is recorded at the manufacturing site.

VPD. Vital product data.

W

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.
13 Index

4765 application..........................................................1
abbreviations..........................................................85
agent.................................................................1
arguments.....................................................................
  file description...................................................71
  image file..........................................................72
  signature key......................................................71
  targeting...........................................................73
  trust..................................................................72
card directory..........................................................9
CCA...............................................................7
CCA roles............................................................82
CCA roles for Signer and Packager..................................82
changing the host device driver timeout..............51
CLU.................................................................49
CLU file names......................................................13
CLU files directory..................................................11
coding requirements, special.................................
  developer identifiers.........................................21
compiler errors.....................................................23
3.4 compiling.......................................................22
  options..........................................................22
Coprocessor Load Utility (CLU).................................49
  commands.......................................................49
  files used as input...........................................11
  return codes...................................................50
  syntax..........................................................49
coprocessor memory segments.............................55
CRUPKGR (Packager utility).................................77
CRUSIGNR (Signer utility).......................................61
debugger, attaching...............................................21
debuggers directory...............................................15
debugging..........................................................27, 46
determining coprocessor status.............................36
development components.................................4
development environment......................................
  overview.........................................................3
  road map........................................................20
  special coding requirements.............................21
  toolkit components............................................4
development process example............................47
development process, overview...........................3, 36
device driver, installing.......................................18
Device Reload Utility (DRUID).............................
syntax..........................................................47
Device Reload Utility for Insecure Development
  (DRUID)............................................................
  description...................................................5
  syntax.........................................................27
directories........................................................8
directory search order...........................................23
disaster recovery.................................................59
docs directory.....................................................15
documentation, available.................................1
downloading and debugging..................................27
DRUID.................................................................
  description...................................................5
  syntax.........................................................27
EMBURN2 function..................................................63
EMBURN3 function..................................................64
ESIG3 function....................................................65
ESTOWN3 function..................................................65
Ethernet port.......................................................24
file ownership, coprocessor...................................27
glossary..........................................................87
HASH_GEN function.............................................66
HASH_VER function.............................................66
how to load software into the coprocessor.............19
ICAT debugger, using...........................................28
image directory...................................................11
inc directory......................................................15
installing the 4765 device driver..........................18
installing the toolkit...........................................8
introduction.......................................................1
JFFS2...............................................................4
  filesystem images, building.............................26
  JFFS2 filesystem images, building.....................46
KEYGEN function................................................31, 67
lib directory......................................................15
library files......................................................24
3.4 linking........................................................22
  options........................................................24
loading the coprocessor........................................37
memory segments and security, coprocessor...........55
notices............................................................84
operating system files, coprocessor-side.............53
options...........................................................24
  compiling.......................................................22
overview of the development process...................36
9.6 Packager utility..............................................77
CCA roles..........................................................82
CRUPKGR.EXE..................................................77
using...........................................................55