IBM 4764 PCI-X 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 F-1.

Third Edition (February, 2007)
IBM does not stock publications at the address given below. This and other publications related to the IBM 4764 PCI-X Coprocessor Card can be obtained in PDF format from the Library page at http://www-03.ibm.com/security/cryptocards.

Reader’s comments can be communicated to IBM by using the Comments and Questions Form located on the product Web site at http://www-03.ibm.com/security/cryptocards, or you can respond by mail to:

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

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

© Copyright International Business Machines Corporation 2005, 2007. All rights reserved.
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.
# Table of Contents

**About This Book** .................................................. v
Prerequisite Knowledge ........................................ v
Organization of This Book ...................................... v
Typographic Conventions ...................................... vi
Syntax Diagrams ................................................ vi
Related Publications .......................................... vii
  General Interest ............................................... vii
  CCA Support Program Publications ............................ vii
  PKCS #11 Support Program Publications ....................... vii
  Custom Software Publications ................................ vii
  Cryptography Publications ................................... viii
  Other IBM Cryptographic Product Publications ............... ix
Summary of Changes ............................................ ix

**Chapter 1. Introduction** ........................................ 1-1
Available Documentation ........................................ 1-1
Prerequisites ..................................................... 1-2
Development Overview ........................................ 1-3
Development Components ...................................... 1-4
Release Components ........................................... 1-5

**Chapter 2. Installation and Setup** ............................... 2-1
Installing the Toolkit ........................................... 2-1
Directories and Files ........................................... 2-1
Installing the IBM 4764 Device Driver ........................ 2-6
Preparing the Development Platform ............................ 2-6

**Chapter 3. Developing and Debugging an xCrypto Application** ....... 3-1
Environment Variables ........................................... 3-1
Development Process Road Map ................................. 3-2
Special Coding Requirements During Development .............. 3-3
  Developer Identifiers ......................................... 3-3
  Attaching with the Debugger ................................ 3-3
Compiling, Assembling, and Linking ............................ 3-4
  Unsupported C Runtime Functions ............................ 3-4
  Compiler Options ........................................... 3-4
  Linker Options ............................................... 3-6
  Programming Serial Port Applications ......................... 3-6
Building JFFS2 Filesystem Images .............................. 3-7
Downloading and Debugging ..................................... 3-9

**Chapter 4. Packaging and Releasing an xCrypto Application** ........... 4-1

**Appendix A. An Overview of the Development Process** ............... A-1

**Appendix B. Using CLU** ........................................ B-1

**Appendix C. How to Reboot the IBM 4764** ........................ C-1
About This Book

The IBM 4764 PCI-X 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 4764 PCI-X Cryptographic Coprocessor
- Load applications under development into a coprocessor
- Debug applications under development running within a coprocessor

The primary audience for this book is 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 book.

Note: <CLU> is used generically throughout this book to indicate either csulclu on Linux, csufclu on AIX or csunclu.exe on Windows, depending on the operating system for the machine on which the adapter is installed.

Prerequisite Knowledge

The reader of this book should understand how to perform basic tasks (including editing, system configuration, file system navigation, and creating application programs) on the host machine. Familiarity with the 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 (as described in the z/OS Cryptographic Services Integrated Cryptographic Service Facility Application Programmer’s Guide) 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 as described in IBM Research Report RC21102, “Building a High-Performance, Programmable Secure Coprocessor.” See “Cryptography Publications” on page viii for information on how to obtain this research report.

Organization of This Book

This book is organized as follows:

Chapter 1, “Introduction” describes the documentation available to a developer of an xC 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 4764 PCI-X Cryptographic Coprocessor for use as a development platform.

Chapter 3, “Developing and Debugging an xCrypto Application” discusses in detail the use of each of the tools used during development of an xC application.

Chapter 4, “Packaging and Releasing an xCrypto Application” describes how to prepare an xC application to be distributed to end users.
Appendix A, “An Overview of the Development Process” lists the steps a developer needs to perform during development and testing of an xC application.

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

Appendix C, “How to Reboot the IBM 4764” 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.

Appendix D, “Building xC Applications with the Developer’s Toolkit Makefiles” describes how to use and customize the makefiles shipped with the toolkit.

Appendix E, “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.

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

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

**Typographic Conventions**

This publication uses the following typographic conventions:

- Commands that you enter verbatim onto the command line are presented in **bold** or **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.
Related Publications

Many of the publications listed below under “General Interest,” “CCA Support Program Publications,” and “Custom Software Publications” are available in Adobe Acrobat portable document format (PDF) at http://www-03.ibm.com/security/cryptocards.

General Interest

The following publications may be of interest to anyone who needs to install, use, or write applications for an IBM 4764 PCI-X Cryptographic Coprocessor:

- IBM 4764 PCI-X Cryptographic Coprocessor General Information Manual (version -01 or later)
- IBM 4764 PCI-X Cryptographic Coprocessor Installation Manual

CCA Support Program Publications

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

- z/OS Cryptographic Services Integrated Cryptographic Service Facility Application Programmer’s Guide
- IBM 4764 CCA Basic Services Reference and Guide

PKCS #11 Support Program Publications

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


Custom Software Publications

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

- IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Interface Reference
- IBM 4764 PCI-X Cryptographic Coprocessor Interactive Code Analysis Tool (ICAT) User’s Guide
- IBM 4764 PCI-X Cryptographic Coprocessor CCA User Defined Extensions Programming Reference
- IBM eServer zSeries CCA User Defined Extensions Reference and Guide
Cryptography Publications

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

- ANSI X9.31 Public Key Cryptography Using Reversible Algorithms for the Financial Services Industry
- IBM Systems Journal Volume 32 Number 3, 1993, G321-5521
- IBM Journal of Research and Development Volume 38 Number 2, 1994, G322-0191
• **USA Federal Information Processing Standard (FIPS):**
  – *Data Encryption Standard*, 46-1-1988
  – *Cryptographic Module Security*, 140-1
• *Derived Test Requirements for FIPS PUB 140-1*, W. Havener, R. Medlock, L. Mitchell, and R. Walcott. MITRE Corporation, March 1995.
• *ISO 9796 Digital Signal Standard*
• *Internet Engineering Taskforce RFC 1321*, April 1992, MD5

**IBM Research Reports can be obtained from:**

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

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

**Other IBM Cryptographic Product Publications**

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

• *IBM Transaction Security System General Information Manual*, GA34-2137
• *IBM Transaction Security System Basic CCA Cryptographic Services*, SA34-2362
• *IBM Transaction Security System I/O Programming Guide*, SA34-2363
• *IBM Transaction Security System Finance Industry CCA Cryptographic Programming*, SA34-2364
• *IBM Transaction Security System Workstation Cryptographic Support Installation and I/O Guide*, GC31-4509
• *IBM 4755 Cryptographic Adapter Installation Instructions*, GC31-4503
• *IBM Transaction Security System Physical Planning Manual*, GC31-4505
• *IBM Common Cryptographic Architecture Services/400 Installation and Operators Guide*, Version 2, SC41-0102
• *IBM Common Cryptographic Architecture Services/400 Installation and Operators Guide*, Version 3, SC41-0102
• *IBM z/OS Integrated Cryptographic Service Facility Overview*, SA22-7519
• *IBM z/OS Integrated Cryptographic Service Facility Application Programmer’s Guide*, SA22-7522
• *IBM z/OS Integrated Cryptographic Service Facility System Programmer’s Guide*, SA22-7520
• *IBM z/OS ICSF Trusted Key Entry Workstation User’s Guide*, SA22-7524

**Summary of Changes**

This first edition of the *IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Developer’s Toolkit Guide* contains product information that is current with the *IBM 4764 PCI-X Cryptographic Coprocessor* announcements.

Changes made to this first edition in June, 2005 include:

• Chapter 2—Added information about changing the default timeout for the xCrypto card.
• Appendix E—Clarification made for trust2_fl under REMBURN3.
Changes made to this second edition in May, 2006 include:

- Chapter 2—Added files that enable changes to the default timeout for the device driver.
- Chapter 3—Added unsupported C runtime functions.
- Changed occurrences of csulclu and csunclu to <CLU> throughout the book, indicating the generic for either Linux or Windows operating system.

Changes made to this second edition in August, 2006 include:

- Chapter 3—Updated init.sh file example.

Changes made to this third edition in February, 2007 include:

- Added AIX specific information throughout.
- Chapter 3—Added AIX switches for compiler options and Outbound Authentication (OA) information.
Chapter 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 4764 PCI-X Cryptographic Coprocessor. An application that runs within the coprocessor is known as an “agent” or an “xC application”.  

The following constitute a complete development environment for the IBM 4764 PCI-X Cryptographic Coprocessor:

- Developer’s Toolkit
- x86 PC
- One of three supported operating systems running on the PC
- SLES 9 (SUSE Linux Enterprise Server 9), Windows 2003, or AIX 5.2/5.3
- C compiler and linker that can cross-compile to the target PowerPC Linux

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 an xC 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 xC 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 an xC application for release

Available Documentation

“Related Publications” on page vii lists over twenty publications, many of which are of particular interest to the developer of an xC application. It may be helpful to read the following manuals in the order listed prior to starting development:

1. IBM 4764 PCI-X Cryptographic Coprocessor General Information Manual which provides a basic understanding of IBM’s Common Cryptographic Architecture for the IBM 4764.  
2. This book, which describes the overall development process and the tools used in the development process.

During development, the following manuals will be of use:

- The IBM 4764 PCI-X 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 4764 PCI-X Cryptographic Coprocessor CCA User Defined Extensions Programming Reference.

1. Linux cryptographic coprocessor (xC) is an alternate name for the IBM 4764 PCI-X Cryptographic Coprocessor.
2. This document will be of particular interest to developers writing user-developed extensions for CCA.
**Prerequisites**

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


   The IBM 4764 PCI-X should be installed in a host following the instructions in the *IBM 4764 PCI-X Cryptographic Coprocessor Installation Manual*, which also lists the hardware and software requirements for the host. For application development, the host must be an Intel PC running SLES 9, AIX 5.2/5.3, or Windows 2003.

2. Developers also need a copy of the IBM CCA application. Available from IBM, which should be installed on the same host as the Developer’s Toolkit. This installation will include the device driver for the 4764 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.

4. A compiler for code running on the host system. For application development, the host must be an Intel PC running SLES 9, AIX 5.2/5.3, or Windows 2003. Only the compiler and linker need to be installed; other components (visual build environments, and so on) are not required.

5. The IBM 4764 PCI-X Application Program Development Toolkit (the Developer’s Toolkit), available from IBM, which should be installed on the same host as the compiler following the instructions in chapter 2 of this manual.

6. Developers also need the IBM 4764 CCA UDX Application Program Development Toolkit Extension (the UDX Toolkit), available from IBM, which should be installed on the same host as the Developer’s Toolkit following the instructions in chapter 2 of this manual.

7. Developers also need to install the proper version of ICAT corresponding to the host system from which they will debug (ICATPCW for Windows 2003 or ICATPCX for SUSE Linux). An AIX hosted version of ICAT is not available at this time. You must use UDP/TCP, or serial communication from a Windows or Linux hosted system to debug applications running on a card residing on an AIX computer.

8. Developers using DRUID need a copy of JFFS2 to build an image that can be downloaded onto the card for development. Depending on the Linux installation, JFFS2 may not be installed. For information, visit [http://sources.redhat.com/jffs2](http://sources.redhat.com/jffs2).

9. Currently for installations of the xCrypto device driver on SLES 9, kernel versions 2.6.5 or later are supported.

**Note:** Refer to the Custom Programming page of the Web site located at [http://www-03.ibm.com/security/cryptocards](http://www-03.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.


Development Overview

The host side of an xCrypto application is compiled and linked using the supported compilers on the host PC. The card side of an xCrypto application is compiled using a cross-compiler (gcc).

As illustrated in Figure 1-1 on page 1-3, an xC 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 an JFFS2 image which can be understood by the xCrypto card and is downloaded to the coprocessor.

![Figure 1-1: Development Process Overview](image)

*Figure 1-1. Development Process Overview*
The following steps are required to build and load xC applications:

1. Write the host and card-side toolkit applications that you want in C, using the Developer’s Toolkit headers as necessary. (Note: The sample applications in xctk/<version>/src/samples provide many examples of how to write/compile a toolkit 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 card-side code using a cross compiler.

5. Link the card-side program using the linker shipped with the cross compiler.

6. Use mkfs.jffs2, along with the Makefile and other files provided in xctk/<version>/build to create a /user0 JFFS2 image.

7. Load the JFFS2 filesystem image into the coprocessor using DRUID.

The Developer’s Toolkit includes the tool needed (DRUID) to perform step 7.

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 (MSVC, xlc, or gcc) compiler and linker for the host application. Use a gcc based cross-compiler for the xC application. These are not shipped with the Developer’s Toolkit.

**Libraries and Include Files**

When developing an application to be run on the IBM 4764 PCI-X, 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 xC applications use to interface with the cryptographic extensions.

**Utilities**

Use the following utilities to prepare and load xC applications:

- **JFFS2 image creation utility:** Use the appropriate jffs2 image creation utility to create a jffs2 image which can be loaded onto the adapter using druid. An example invocation of the jffs2 utility would be as follows (consult the man page for the installed jffs2 image creation utility for more details):

  ```
  cd xctk/<version>/build
  mkfs.jffs2 -b -r user0 -o seg3tk.bin
  ```

- **Device Reload Utility for Insecure Development (DRUID):** A program (druid on Linux, druid.exe on Windows) that loads an application into a coprocessor configured as a development platform.

- **Coprocessor Load Utility (CLU):** A program (csulclu, csufclu, or csunclu.exe) that verifies and loads digitally signed system software and coprocessor commands into a coprocessor. This program is provided with the IBM CCA application.

- **init.sh:** A card side segment 3 initialization shell script which, in addition to other tasks, should set any relevant environment variables for the debugger, and then launch the debugger daemon (pdae-mon) and the new xC application on the card. This shell script should be part of the jffs2 image loaded onto the card.

**CLU Input Files**

The Developer’s Toolkit includes several files used as input to CLU during the development process.

**Debugger**

- **IBM Interactive Code Analysis Tool (ICAT):** A program (icatpcx on Linux, icatpcw.exe on Windows) that controls and debugs xC applications. For more information on ICAT, refer to the IBM 4764 PCI-X
Cryptographic Coprocessor ICAT Debugger Getting Started supplied with ICAT.

**pdaemon:** The card side debugger daemon which must be built into the jffs2 image loaded onto the card and launched by init.sh. This daemon enables ICAT to communicate with the card.

- **Debug Specific Outbound Authentication libraries:** libxcoa.so, libxcoa.so.1, libxcoa.so.1.0.0.
  If pdaemon is started, these files are to be part of the /user0 JFFS2 image for release 325a.

**Coprocessor Operating System**

The Developer’s Toolkit includes one version of the Linux embedded operating system signed with keys corresponding to a production and a development owner ID pair. The development and production version of the Linux operating system are identical, and can be loaded into the coprocessor by CLU. The copy of the operating system corresponding to the development owner ID’s is intended to be used in conjunction with DRUID for more rapid development.

**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 listed as follows.

**Utilities**

Use the following utilities to prepare an xC application for release:

- **Signer:** A program (CRUZSIGNR/CRUSIGNR) that generates RSA keypairs and performs other cryptographic operations and that incorporates a jffs2 image into a coprocessor command and digitally signs the command using a developer’s private key.
  **Note:** CRUZSIGNR should be used if the IBM 4764 Cryptographic Coprocessor resides in a zSeries machine.

- **Packager:** A program (CRUZPKGR/CRUPKGR) that combines one or more signed commands into a single file for download to the coprocessor.
  **Note:** CRUZPKGR should be used if the IBM 4764 Cryptographic Coprocessor resides in a zSeries machine.

**CCA Application and Support Program**

Signer and Packager use IBM’s Common Cryptographic Architecture (CCA) application to generate keypairs and digital signatures required for signing and packaging. See “Prerequisites” on page 1-2 for more information.
Chapter 2.

Installation and Setup

The Developer’s Toolkit includes utilities used to build an xC application and prepare it to be loaded into an IBM 4764 PCI-X Cryptographic Coprocessor. This chapter describes how to install the Developer’s and the UDX Toolkits, discusses the toolkit’s directory structure and contents, lists many of the files used during development, and explains how to prepare the coprocessor for use as a development platform.

Installing the Toolkit

The Developer’s Toolkit is shipped on CD-ROM and can be installed by copying it to a directory of your choosing.

Directories and Files

The Developer's Toolkit is contained in the directory structure depicted in Figure 2-1 on page 2-2.

The icat directory contains versions of the Interactive Code Analysis Tool for SLES 9 and Windows 2003, as well as the subsequent documentation for each version.

The udxtk directory includes the tools required to create User Defined Extensions to IBM’s Common Cryptographic Architecture (CCA) application. Refer to IBM 4764 PCI-X Cryptographic Coprocessor CCA User Defined Extensions Reference and Guide for details.
Figure 2-1. Toolkit Directory Structure
The `xctk/<version>/bin/<platform>` directory contains the following:

- A utility for displaying information about signed toolkit images, fmtTKclu.
- Device reload utility for insecure development (DRUID)
- Signer utility (CRUSIGNR/CRUZSIGN)
- Packager utility (CRUPKGR/CRUZPKG)
- Command files used by the tools in the directory
- Files necessary to enable changes to the default timeout for the device driver for debugging purposes.

A user may want to append `xctk/<version>/bin/<platform>` to be in the appropriate PATH for their given operating system.

When creating a toolkit application, developers may find that the default timeout for the 4764 driver provides them with insufficient time to debug their application before the driver attempts to reset the card. For debugging purposes only, files are provided to alter the default timeout for the 4764 driver.

For Linux, the following files are provided in `xctk/<version>/bin/linux` which will alter the default timeout:

- **xcload** - used as root or by way of `su` or `sudo`, for example:
  
  ```bash
  xcload timeout=N
  ```

  where `N` is specified in seconds. Typically, a timeout of 300 (5 minutes) is suitable for debugging purposes.

  **Notes:** If the driver is already loaded, run `xcunload` to unload the driver before loading it with a user specified device driver timeout. Invoking `xcload` with no arguments loads the device driver with the default timeout.

- **xcunload** - This command unloads the device driver.

For Windows, the following files are provided in `xctk/<version>/bin/windows`:

- `ibm4764w.sys`
- `4764_W2K_timeout.reg`

To change the default timeout on a Windows-based system, do the following:

- Stop the device driver for the 4764 if it is running. This can be accomplished by going to the Windows Device Manager and disabling the IBM 4764 Cryptographic Coprocessor.
- Locate the installed version of `ibm4764w.sys` on your machine and copy/rename it to `ibm4764w.sys.bak`.
- Copy the `ibm4764w.sys` file from `xctk/<version>/bin/windows` to the location where the original `ibm4764w.sys` was located.
- Copy the `ibm4764w.sys` file from `xctk/<version>/bin/windows` to the location where the original `ibm4764w.sys` was located.
- Edit `4764_W2K_timeout.reg` to have the desired timeout value. (Note: The maximum allowable timeout on a Windows system is 420 seconds.)
- Add the information from `4764_W2K_timeout.reg` to the Windows registry (double-click the file, or use regedit).
- Restart the device driver from within the device manager.
To revert back to the original timeout:

- Stop the device driver for the 4764 if it is running. This can be accomplished by going to the Windows Device Manager and disabling the IBM 4764 Cryptographic Coprocessor.
- Locate the debug version of \textit{ibm4764w.sys} on your machine, and replace it with the original version.
- Remove the following registry key using regedit:
  \[
  \text{[HKEY\_LOCAL\_MACHINE\:SYSTEM\:CurrentControlSet\:Services\:ibm4764w\:Parameters]} \]
- Restart the device driver from within the device manager.

- The \texttt{xctk/<version>/etc} directory contains files to be used as input to CLU (as described in “Preparing the Development Platform” on page 2-6 and “Downloading and Debugging” on page 3-8), including those listed as follows. (Many files in this directory have names of the form \texttt{Txxrrss}, where \texttt{rr} indicates the release of the CCA application the file contains or with which the file is associated, \texttt{ss} indicates the revision level of the CCA application, and \texttt{xx} distinguishes the file from all others.)

  - \texttt{CR1rrrss.CLU}, which loads release \texttt{rr} revision \texttt{ss} of IBM’s system software into a coprocessor.\textsuperscript{1}

\texttt{CR1rrrss.CLU} can only be loaded into an \texttt{xCrypto} card in the factory-fresh state.\textsuperscript{2}

  - \texttt{CE1rrrss.CLU}, which updates the system software in a coprocessor.

\texttt{CE1rrrss.CLU} loads release \texttt{rr} revision \texttt{ss} of IBM’s system software into an \texttt{xCrypto} card into which system software has previously been loaded.

\textbf{Warning}

\begin{minipage}{\textwidth}
\texttt{CE1rrrss.CLU} updates the public key associated with segment 1. This key can only be updated a few 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 \texttt{CE1rrrss.CLU} need be loaded only once.
\end{minipage}

  - \texttt{TDVrrrss.CLU}, which prepares a coprocessor for use as a development platform.\textsuperscript{3}

\texttt{TDVrrrss.CLU} can only be loaded into an \texttt{xCrypto} card that contains release \texttt{rr} revision \texttt{ss} of IBM’s system software.\textsuperscript{4}

\begin{minipage}{\textwidth}
\texttt{TDVrrrss.CLU} sets the public key and owner identifier associated with segment 2. Currently, the owner identifier assigned to segment 2 is 3.
\texttt{TDVrrrss.CLU} can only be loaded into an \texttt{xCrypto} card that contains release \texttt{rr} revision \texttt{ss} of IBM’s system software.
\end{minipage}

\begin{itemize}
\item CR1rrrss.CLU and CE1rrrss.CLU also set the public key associated with segment 1.
\item In particular, the public key associated with segment 1 must be the key installed during manufacture.
\item TDVrrrss.CLU sets the public key and owner identifier associated with segment 2. Currently, the owner identifier assigned to segment 2 is 3.
\item In particular, segment 2 must be empty and the public key associated with segment 1 must be the key loaded by CR1rrrss.CLU or CE1rrrss.CLU. Loading CCA also causes the key associated with segment 1 to be set to the proper value.
\end{itemize}
- **TE3rrrss.CLU**, which enables a coprocessor to accept coprocessor applications downloaded by the DRUID utility.5

**TE3rrrss.CLU** can only be loaded into an xCrypto card that has been prepared for use as a development platform using **TDVrrss.CLU**.5

Export regulations may dictate that the version of **TE3rrrss.CLU** shipped to a particular developer be customized so that the file can only be loaded into a specific coprocessor or a specific set of coprocessors.

- **TL3rrrss.CLU**, which clears any state an application under development has saved in nonvolatile memory (so that the application will start next time with a clean slate). **TL3rrrss.CLU** also loads the “reverse-then-echo” application into the coprocessor.

**TL3rrrss.CLU** can only be loaded into an IBM 4764 PCI-X that has been prepared for use as a development platform using **TDVrrss.CLU** and which has been prepared to accept downloaded applications using **TE3rrrss.CLU**.6

- **TR3rrrss.CLU**, which reloads the “reverse-then-echo” application into the coprocessor.

**TR3rrrss.CLU** can only be loaded into an xCrypto card that has been prepared for use as a development platform using **TDVrrss.CLU** and which has been prepared to accept downloaded applications using **TE3rrrss.CLU** and **TL3rrrss.CLU**.6

- One or more RSA key token files (file extension .TKN). The developer uses these files with **CRUSIGNR** to generate RSA keys prior to releasing an application.

- **TRSrrrss.CLU**, which prepares an xCrypto card that has been used for development to be used in a production setting. **TRSrrrss.CLU** essentially restores the coprocessor to the state it is in immediately after **CR1rrss.CLU** or **CE1rrss.CLU** has been loaded.

**TRSrrrss.CLU** can only be loaded into an xCrypto card that has been prepared for use as a development platform using **TDVrrss.CLU**.7

- **ESTOWN2.E2T**, **EMBURN2.L2T**, **REMBURN2.R2T**, and **SUROWN2.S2T**, which are used to generate a version of an application suitable for release. See Chapter 4, “Packaging and Releasing an xCrypto Application” on page 4-1 for details.

- **tdvRRLLL.L2t** is the Toolkit EMBURN2 unpackaged CLU file. This file should only be used when segment2 has ownerID = 3 but is not reliable. In rare instances, such as a power or system failure during a load of **tdvRRLLL.clu**, it is possible for the card to have segment2 as owned, but unreliable. This file performs an EMBURN2 command on segment 2 with segment 2 ownerID = 3 which restores the card to a usable state after a failed toolkit CLU load for segment 2.

- The **xctk/inc** directories contain include files (.h) that are required to build cryptographic applications.

---

5. **TE3rrrss.CLU** sets the owner identifier associated with segment 3. Currently, the owner identifier assigned to segment 3 is 6.
6. In particular, the owner identifier associated with segment 3, the public key and owner identifier associated with segment 2, and the image names associated with segments 1 and 2 must have the values **CR1rrss.CLU**, **TDVrrss.CLU**, and **TE3rrrss.CLU** assign them.
7. In particular, the public key and owner identifier associated with segment 2 must have the values **TDVrrss.CLU** assigns them.
- xctk/inc/common contains include files that are used to build both card applications and host applications that interact with card applications.
- xctk/inc/xcmanager contains include files used by various xCrypto functions.

- xctk/inc/gcc contains Linux specific host-side include files
- xctk/<version>/inc/xlc contains AIX specific host-side include files
- xctk/inc/msvc contains Windows specific host-side include files

- The xctk/<version>/lib directories contain shared object (.so) files that provide specific xCrypto related functionality. The xctk/<version>/lib/card/gcc directory contains shared object files which are stubs, containing symbols which are used to resolve symbol information needed by the linker for the cross compile used to build xC applications. The gcc sub-directories are used when building applications with gcc. The xctk/<version>/lib/host/msvc subdirectory contains the lib file to link Microsoft Visual C++ applications with the device driver.

- The xctk/<version>/src/samples directories contain the source for several sample host and card applications.

These directories are typically broken down into host, card, and shared components. Makefiles always appear in a folder corresponding to the compiler name which should be used when building these examples (gcc for Linux, xlc for AIX, msvc for Windows).

Installing the IBM 4764 Device Driver

The device driver is part of the CCA install downloaded from the IBM Web site located at http://www-03.ibm.com/security/cryptocards/, and it is assumed that the device driver has been installed and properly configured as a prerequisite for using the Toolkit.

Preparing the Development Platform

After the Developer’s Toolkit (and the UDX Toolkit, if appropriate) and all prerequisites (see “Prerequisites” on page 1-2) have been installed, the developer must 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 xctk/<version>/bin/<platform> and the CCA installation directory), and that the various system files (Linux or Windows 4764 device driver) have been installed.

CLU’s ST command can be used to determine what software, if any, is loaded in the coprocessor. For example:

<CLU> /logfile-directory/CLU.LOG ST

An excerpt from a typical response to this command is as follows:

*** ROM Status; PIC ver: 28, ROM ver: 8a59
*** ROM Status; INIT: INITIALIZED
*** ROM Status; SEG2: RUNNABLE, OWNER2: 3
*** ROM Status; SEG3: RUNNABLE, OWNER3: 6
*** Page 1 Certified: YES
*** Segment 1 Image: 3.22 POSTlv2.16 MBlv1.25 FPGAv78 20050215060132200000220000...
*** Segment 1 Revision: 322
*** Segment 1 Hash: C16F 4102 0989 51FE 4FA8 44B6 3B3A 274E 04F4 3DF0
*** Segment 2 Image: 3.25 Linux OS MCP v1.3 No Probes 2006011808593250000000000...
*** Segment 2 Revision: 325
*** Segment 2 Hash: 2FAA 3F97 5078 40E6 9921 C94C 1E37 B987 A93D 956E
*** Segment 3 Image: seg3icat.bin 1077388 Wed May 10 15:00:06 2006 (DRUID v1.0)
*** Segment 3 Revision: 0
*** Segment 3 Hash: 37C2 7CF4 F344 B14F 46A3 4568 60FB 2E2E E8CA 5B0B
*** Query Adapter Status successful ***

The First ROM Status Line
If the first “ROM Status” line does 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.

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 likely be rather cryptic. In this case, the developer loads CR1rrrss.CLU into the coprocessor, for example:

  `<CLU> /logfile-directory/CLU.LOG PL ~/xctk/<version>/etc/CR1rrrss.CLU`

  CR1rrrss updates the system software in segment 1.

  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 TDVrrrss.CLU as indicated in “Segment 1 Current on page 2-7.”

- **Segment 1 Downlevel: Update Segment 1** - If segment 1 contains a downlevel version or revision of CCA segment 1, the developer loads CE1rrrss.CLU into the coprocessor, for example:

  `<CLU> /logfile-directory/CLU.LOG PL ~/xctk/<version>/etc/CE1rrrss.CLU`

  CE1rrrss.CLU updates the system software in segment 1.

  The developer then proceeds to load TDVrrrss.CLU as indicated in “Segment 1 Current” as follows.

- **Segment 1 Current: Load Segments 2 and 3** - If segment 1 contains the appropriate version and revision of CCA segment 1, the developer loads TDVrrrss.CLU into the coprocessor, for example:

  `<CLU> /logfile-directory/CLU.LOG PL ~/xctk/<version>/etc/TDVrrrss.CLU`

  TDVrrrss.CLU loads a production version of the coprocessor operating system into segment 2.

  The developer then loads TE3rrrss.CLU into the coprocessor, for example:

  `<CLU> /logfile-directory/CLU.LOG PL ~/xctk/<version>/etc/TE3rrrss.CLU`

  TE3rrrss.CLU sets the owner identifier for segment 3, which makes it possible to load software into segment 3.
Finally, the developer loads TL3rrrss.CLU into the coprocessor, for example:

```<CLU> /logfile-directory/CLU.LOG PL ~/xctk/<version>/etc/TL3rrrss.CLU```

TL3rrrss sets the public key associated with segment 3 and loads the “reverse-then-echo” application.

If desired, the developer can confirm the software has been properly loaded by resetting the coprocessor to start the “reverse-then-echo” application loaded by TL3rrrss.CLU (see Appendix C, “How to Reboot the IBM 4764” on page C-1 for details) and then running the host reverse-then-echo driver, for example:

```
HRE adapternumber text```

**Segment 2 neither UNOWNED nor RUNNABLE**

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 owner ID is 3, the developer loads TRSrrrss.CLU into the coprocessor, for example:

```<CLU> /logfile-directory/CLU.LOG PL ~/xctk/<version>/etc/TRSrrrss.CLU```

Both files surrender ownership of segment 2. If the command succeeds, segments 2 and 3 become UNOWNED and the developer proceeds according to the instructions given in “Segments 2 and 3 UNOWNED” discussed on page 2-7.

**Segment 2 RUNNABLE, segment 3 neither UNOWNED nor RUNNABLE**

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 owner ID is not 3, the owner of segment 2 must supply a CLU file to surrender that ownership before the coprocessor can be used as a development platform.

If the segment 2 owner ID is 3 and the segment 3 owner ID is not 6, the developer loads TRSrrrss.CLU into the coprocessor, for example:

```<CLU> /logfile-directory/CLU.LOG PL ~/xctk/<version>/etc/TRSrrrss.CLU```

TRSrrrss.CLU surrenders ownership of segment 2. If the command succeeds, segments 2 and 3 become UNOWNED and the developer proceeds according to the instructions given in “Segments 2 and 3 UNOWNED” described previously.

If the segment 2 owner ID is 3 and the segment 3 owner ID is 6, the developer loads TL3rrrss.CLU into the coprocessor, for example:

```<CLU> /logfile-directory/CLU.LOG PL ~/xctk/<version>/etc/TL3rrrss.CLU```

TL3rrrss.CLU sets the public key associated with segment 3 and loads the “reverse-then-echo” application into segment 3.
Figure 2-2 illustrates the steps involved in preparing an xCrypto card for use as a development platform.

![Diagram showing the Development Preparation Process]

**Figure 2-2. Development Preparation Process**
Chapter 3.

Developing and Debugging an xCrypto Application

This chapter describes how to use the Developer’s Toolkit to create the coprocessor-side portion of an xCrypto application and load it into an IBM 4764 PCI-X Cryptographic Coprocessor. (The host-side portion of an xCrypto 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 3-5 are searched in the proper order, define the appropriate host side variables listed in this chapter, and link in with the host side xCrypto driver by way of the -lxcryp option to gcc or by linking in ibm4764w.lib with MSVC.)

This chapter describes:

• 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 of the xCrypto application
• How to load the JFFS2 filesystem into the coprocessor
• How to start the debugger

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 (that is, the PATH environment variable includes either xctk/<version>/bin/linux, xctk/<version>/bin/aix, or xctk/<version>/bin/windows).

Furthermore, 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 environment variables XCTK_FS_ROOT and UDXTK_FS_ROOT must be set in order to use the developer’s toolkit makefiles and, on Linux systems, LD_LIBRARY_PATH should point to a directory containing libxcryp.so.

• XCTK_FS_ROOT should be set to point to the root of the XCTK (that is, the fully qualified path to xctk/<version>)
• UDXTK_FS_ROOT should be set to point to the root of the UDXTK (that is, the fully qualified path to udxtk/<version>)

For card-side applications, two additional environment variables must be defined which tell the makefile where to find the cross compiler:

• CROSS - Should be set to the base of the cross compiler’s directory structure
• GCC_NAME - Should be set to the prefix of the cross compiler uses for the name of it’s version of GCC.

All Toolkit makefiles assume that the correct version of the cross compiler is located at $(CROSS)/bin$(GCC_NAME)gcc and the correct version of the linker is located at $(CROSS)/bin$(GCC_NAME)ld. Note that depending on the version of the cross compiler used these locations may change.

See Appendix D, “Building xC Applications with the Developer’s Toolkit Makefiles” on page D-1 for details.
Development Process Road Map

As introduced in Chapter 1, "Introduction," the procedure to build an xCrypto application and load it into the development coprocessor consists of the following steps:

1. Compile, assemble, and link
2. Build /user0 JFFS2 filesystem
3. Load JFFS2 filesystem into the coprocessor

Figure 3-1 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 1-1 on page 1-3; this flowchart simply provides more detail.

Figure 3-1. Development Process Road Map

The following sections detail how to use the Developer’s Toolkit to perform these steps.
Special Coding Requirements During Development

Developer Identifiers
An xCrypto application must register with an xCrypto device manager before the application can receive requests from the host. The application must supply a “developer identifier” that uniquely identifies the developer as part of the registration process. During development, a developer may use a valid short for the developer identifier. 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.

Attaching with the Debugger
An application that has been downloaded to the coprocessor can be loaded by the /user0/init.sh shell script incorporated into the mkfs image. If it is not started by the shell script, the user attaches and launches the program by supplying the full path (that is, /user0/rteX). To ensure the application does not make too much progress before the debugger takes control, the developer must 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 use the debugger’s Jump to location function to move the execution point to the statement immediately following the loop.

---

1. Refer to the description of xcAttach in IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Interface Reference for details.
2. The card side debugger daemon (pdaemon) reserves the agent ID 0x6668 (“DB”, byte reversed) for PCI communications. This agent ID should not be used for development purposes.
Compiling, Assembling, and Linking

The cross-compiler used in development of an xCrypto application is designed to create applications to run on the cryptographic coprocessor under Linux.

This section lists the base operating system and C run-time library function calls not supported by an xCrypto application. It also lists options that must be specified when compiling, assembling, or linking to ensure that an xCrypto 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 tool. See Appendix D, “Building xC Applications with the Developer’s Toolkit Makefiles” on page D-1 for details on their use.

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

Unsupported C Runtime Functions

A limited number of C runtime functions are not supported by the C runtime library installed as part of the xCrypto operating system on the adapter. These functions must be implemented by the toolkit developer to function properly. All unsupported functions are shown in the following table.

Table 3-1: Unsupported C Runtime Functions

<table>
<thead>
<tr>
<th>isalnum</th>
<th>isalpha</th>
<th>iscntrl</th>
<th>isdigit</th>
</tr>
</thead>
<tbody>
<tr>
<td>isgraph</td>
<td>islower</td>
<td>isprint</td>
<td>ispunct</td>
</tr>
<tr>
<td>isspace</td>
<td>isupper</td>
<td>isxdigit</td>
<td>strupr</td>
</tr>
</tbody>
</table>

Compiler Options

Building applications with the Developer’s Toolkit requires a gcc cross compiler (which must be installed separately).

Reminder: Although you use a C++ compiler to compile xCrypto applications, the applications must be written in C.

When using the cross-compile environment, use the following switches with the gcc command to control the process.

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

<table>
<thead>
<tr>
<th>Switch</th>
<th>Function</th>
<th>Coprocessor</th>
<th>Linux Host</th>
<th>AIX Host</th>
<th>Windows Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>-DDEBUG</td>
<td>Ensure test cases enter a spin loop prior to execution. Use only for debugging.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>-DLINUX_ON_INTEL</td>
<td></td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
**Include File Directory Search Order**

The appropriate compiler options should be used when building the coprocessor-side portion of an xCrypto application to ensure the following directories are searched for include files in the order shown:

1. `xctk/<version>/inc/common`
2. `xctk/<version>/inc/xcmanager`
3. `udxtk/<version>/inc`

For example, the following compiler options might be specified to build the coprocessor-side portion of a card application:

```
-I/home/user/xctk/<version>/inc/common -I/home/user/xctk/<version>/inc/xcmanager
-I/home/user/udxtk/<version>/inc
```

Similarly, the appropriate compiler options should be used when building the host-side portion of an xCrypto application to ensure the following directories are searched for include files in the order shown:

1. `xctk/<version>/inc/common`
2. `xctk/<version>/inc/xcmanager`
3. `xctk/<version>/inc/<compiler>`

For example, the following compiler options might be specified to build the host-side portion of a card application:

```
```

<table>
<thead>
<tr>
<th>Switch</th>
<th>Function</th>
<th>Coprocessor</th>
<th>Linux Host</th>
<th>AIX Host</th>
<th>Windows Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>-DLINUX</td>
<td>Compile code for Linux.</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</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>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>-DNT_ON_I386</td>
<td>Compile code for Windows.</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>-DHOST32BIT</td>
<td>Define 32-bit addressing for communication code.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>-msoft-float</td>
<td>Ensure floating-point libraries are called.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>-gstabs+</td>
<td>Produce stabs debugging information. Needed for the ICAT debugger.</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
<td>No</td>
</tr>
<tr>
<td><strong>AIX</strong></td>
<td>Compile code for AIX flag.</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>UNIX</strong></td>
<td>Another compile code for AIX flag.</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>BIG_ENDIAN</strong></td>
<td>Specify AIX host is Big Endian.</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>AIX_ON_RS6000</td>
<td>Specify compilation is intended for AIX platform.</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

3. The switch "-I" is capital letter "i" (as in "include").
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 Which May be Linked with a Card-Side Application

- `xctk/<version>/lib/card/gcc/libxccomapi_stub.so` If the application will communicate with the host
- `xctk/<version>/lib/card/gcc/libxcmgrapi_stub.so` If the application will use configuration functions
- `xctk/<version>/lib/card/gcc/libxcoa_stub.so` If the application will use Outbound Authentication functions
- `xctk/<version>/lib/card/gcc/libxcrandom_stub.so` If the application will use the random number generator, or will generate DES or RSA keys
- `xctk/<version>/lib/card/gcc/libxcrsalnx_stub.so` If the application will use Large Integer Modular Math, or RSA or DSA functions
- `xctk/<version>/lib/card/gcc/libxcskch_stub.so` If the application will use DSE or Hashing functions

Programming Serial Port Applications
Special considerations must be taken to use the serial port on the IBM 4764. To successfully use the serial port, ensure that the following conditions are met:

- Data communications must occur in 16-byte chunks. The IBM 4764 hardware does not provide software buffering; therefore, all communications should be padded to 16 bytes where appropriate.
- Any printf or other messages that would normally be emitted to stdout/stderr should be removed, as this interferes with serial port communications.
- Any echo statements or other items that print out data in the `init.sh` shell script should be removed, as this prevents output from interfering with the serial port communications.
- If a terminal emulation program has been attached to the card to monitor output, it should be disconnected to prevent interference with the serial port communications.
- A NULL modem cable must be used to enable two-way communications across the serial port.
- The proper communications settings for the serial port are detailed in the serial port sample application for the specific platform

When adding a cable to the serial port, the addition of a ferrite core to the cable is required to maintain compliance with the Class B, EMC directive. To avoid potential ESD disruption when using the serial port, you must use cable lengths less than or equal to eight feet. See the following illustration for placement of the ferrite core.
Ferrite core purchasing information
Vendor: FerriShield, Inc.
Part number: SS28B2032
Phone: (570) 961-5617
Web site: http://www.ferrishield.com/

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 for /user0 is to use the makefile provided in
xctk/<version>/build/Makefile, edit it to copy the desired application to the build directory, and make any
changes to init.sh that are required for your application.

The following example code shows how to build a the segment 3 JFFS2 filesystem:

```bash
# Construct user0 file system (aka segment3) for xCrypto
# Toolkit Example Reverse Then Echo (RTE)
# This makefile should be invoked with root privileges
# or change the following commands below to include "sudo"
#
MKFS.JFFS2 =mkfs.jffs2
CP = cp
CHMOD = chmod
CHOWN = chown
RM = rm -rf
MKDIR = mkdir

PROG_NAME=rteX
IMAGENAME = seg3tk.bin
```
Notes about the example code

Required Files

The name of the directory at the root of this JFFS2 filesystem must be user0.

The xcmgrd file must be run to query the tamper status of the coprocessor. This file may be found in the udxtk/<version>/bin directory.

The user0 directory must contain a shell script named 'init.sh'. This file will be processed as the
initialization of the Segment 3 code, when the Segment 2 code finishes processing. The init.sh script should load all needed device drivers (for the SKCH chip and the communications manager, for instance), set up the non-volatile memory directories (/bbram and /user1) properly, and start the optional and required processes, including xcmgrd, cryptologkd, pdaemon, xcoad, and your process.

**Optional Files**

The log/cryptologkd file may be run to allow logging of coprocessor events. This file may be found in the udxtk/<version>/bin directory. The log data is sent to the host machine via the PCIIX bus, and stored in a file on the host.

The startoa file may be run to start the oa/xcoad file with correct permissions and ownership (see “Ownership of Files on the Coprocessor” below).

The oa/xcoad file may be run to allow the use of Outbound Authentication functions. This file may be found in the udxtk/<version>/bin directory.

The pdaemon file may be run to allow debugging of your applications. This file may be found in the <icat_root>/card directory.

For release 325a, pdaemon will only run once OA has been started using the special OA libraries provided with ICAT. See “Downloading and Debugging” and refer to the *Interactive Code Analysis Tool (ICAT) User’s Guide*.

**Ownership of Files on the Coprocessor**

It is 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.

**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.

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 TL3rrrss.CLU to the coprocessor using CLU:

```
<CLU> /logfile-directory/CLU.LOG PL ~/xctk/etc/TL3rrrss.CLU
```

**Syntax**

```
druid [image_fn [coprocessor_number]]
```

where

- *image_fn* 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.
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 before it downloads the application. The summary includes:

- Coprocessor’s serial number
- Current left and right bootcounts (see “Targeting Arguments” on page E-17 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

Linux on the xCrypto card loads and runs the user0/init.sh file after the coprocessor is rebooted. See Appendix C, “How to Reboot the IBM 4764” on page C-1 for a description of how to reboot the coprocessor. The following code example shows how a developer can start and run code on the coprocessor in an init.sh file:

```
#!/bin/sh

# Script Name: init.sh
# Descriptive Name: Segment 3 Initialization Shell Script

# 04/07/04 BDJ Add xcmgrd start
# 01/17/04 BDJ Remove Tget=13000 parm from commgr

insmod xccommgr
insmod xcpkamgr
insmod xcdesmgr
insmod xcsupmgr
insmod xchlm

chmod 660 /dev/crypto
chmod 444 /dev/hwrng0
chmod 444 /dev/hwrng1
chmod 660 /dev/pka
chmod 660 /dev/skch
chmod 660 /dev/xccommgr
chmod 660 /dev/xchlm

chmod 774 /bbram
chmod 770 /user1

# start the logging daemon
/user0/log/cryptologkd
```

---

4. That is, the value xcGetConfig returns in pConfigData->VPD.sn. Refer to the IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Interface Reference for details.

5. That is, the value of pubkey_fn supplied when DRUID last downloaded an application to the coprocessor.
# initialize the debugger and it’s settings

# For the 325a release, the debugger requires OA to be started
# Using a special set of libraries copied from the card directory
# of ICAT. LD_LIBRARY_PATH *must* be set so that these libraries
# are found first, and the OA daemon *must* be running in order to debug

# set debugger communication type
export CAT_COMMUNICATION_TYPE=PCI
# Set LD_LIBRARY_PATH to point to our debug specific OA libraries
export LD_LIBRARY_PATH=/user0/oa:/lib:$LD_LIBRARY_PATH
# IMPORTANT!!! Start OA before starting ICAT
/user0/startoa

# launch the debugger
/user0/pdaemon &

# start your application here.
/user0/rteX &

# Load and start xcmgrd
/user0/xcmgrd/xcmgrd -t30
exit

**Important:** Linux on the xCrypto card loads and runs the `user0/init.sh` file after the coprocessor is booted. Linux requires that the shell script be a plain text file saved with Unix style line feeds, rather than DOS style line feeds. When editing the `user0/init.sh` file 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 `init.sh` file after editing.)

Details on the use of CLU can be found in the *IBM 4764 PCI-X Cryptographic Coprocessor Installation Manual* or in Appendix B, “Using CLU” on page B-1 of this manual.

After the application is running, it can be debugged using the ICAT debugger. Refer to the *IBM 4764 PCI-X Cryptographic Coprocessor Interactive Code Analysis Tool (ICAT) User’s Guide* for details.
Important Notes:

1. **Warning:** ICAT now requires the Outbound Authentication daemon to be running on the card before pdaemon is started inside init.sh. The proper way to start the OA daemon is documented inside xctk/325a/build/user0/inits.h.

   To properly start the OA daemon, the makefile located in xctk/325a/build must be used to copy the required binaries into the /user0 JFFS2 image, and to set their permissions and ownership correctly. If the permissions and ownership are not correctly set, OA daemon will not launch, and pdaemon will be unable to run on the card.

   An alternate method which is less restrictive, and should be used for debugging purposes only is to launch the OA daemon directly with the following line in init.sh (use this instead of startoa):

   `/user0/oa/xcoad &`

2. When using pdaemon on the card, the Outbound Authentication daemon **must** be started before ICAT. Additionally, the special OA libraries in `<ICAT_ROOT>/card` must be copied to the `/user0/oa` directory, and LD_LIBRARY_PATH must be set with `/user0/oa` as the first item in the search path. For security purposes, pdaemon is only allowed to run in development environments. Specifically, the owner IDs for segments 2 and 3 must be 3 and 6.

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

4. If you decide to debug your application using serial port communications, you must ensure the following conditions are met before debugging can take place:
   - All printfs have been disabled inside the card-side application.
   - All other applications that use the serial port (that is, minicom) must be disabled.
   - All commands inside the initialization shell script (`/user0/init.sh`) must be disabled or forced to run in a quiet mode.
   - cryptologkd must be disabled.
Chapter 4.  
Packaging and Releasing an xCrypto Application

The design for the IBM 4764 PCI-X Cryptographic Coprocessor was motivated by the need to simultaneously satisfy the following requirements1:

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 usually 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 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 contents, owner, and public key for segment 3 must be signed with the private key for segment 2.

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.
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”). When the developer is ready to deploy an application in the field, the developer must obtain a unique developer identifier from IBM and must generate a new, unique RSA keypair. This is summarized in the following table.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Development</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td>“Generic developer”</td>
<td>Developer-unique identifier</td>
</tr>
<tr>
<td>Public Key</td>
<td>Generic (common) key</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 TRSrrrss.CLU:\(^2\)

```
<CLU> /logfile-directory/CLU.LOG PL ~/xctk/etc/TRA sữa.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 4758 PCI Cryptographic Coprocessor CCA Support Program Installation Manual*. This prepares the coprocessor for use by the signer utility (CRUSIGNR/CRUZSIGN) and the packager utility (CRUPKGR/CRUZPKG).

The developer generates three\(^3\) RSA keypairs using CRUSIGNR’s KEYGEN function, for example:\(^4\)

```
CRUSIGNR KEYGEN 2 S3KDEVP PUBLIC P.P.KEY S3DEVPU.KEY ~/xctk/etc/DFT_SKEL.TKN
CRUSIGNR KEYGEN 2 DEVSGNPP PUBLIC P.P.KEY DEVSGNPU.KEY ~/xctk/etc/DFT_SKEL.TKN
CRUSIGNR KEYGEN 2 DEVPKGPP PUBLIC P.P.KEY DEVPKGPU.KEY ~/xctk/etc/DFT_SKEL.TKN
```

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 CCA 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, S3KDEVP.P.KEY) and the other containing just the public key (for example, S3KDEVPU.KEY). The

---

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 xctk/bin/linux).
3. Only the first two keypairs need to be generated when developing a UDX that will be run on a PCI Cryptographic Coprocessor installed in an IBM zSeries server. The CRUZPKG utility (zSeries packager) does not require keys or certificates as input.
4. This version of the KEYGEN command does not encrypt the private keys in the *PP.P.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 S3KDEVP.P.KEY S3DEVPU.KEY ~/xctk/etc/DFT_SKEL.TKN
```

The appropriate actions should be taken to ensure the master key can be regenerated should the need arise. Refer to the *IBM 4764 PCI-X 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 Appendix E, “CCA Roles for Signer and Packager” on page E-25 for more information.
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 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. A UDX that will be run on a PCI Cryptographic Coprocessor installed in an IBM zSeries server is packaged using the CRUZPKG utility, which does not require a certificate.

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

2. The following files generated by CRUSIGNR.\(^5\)

   - ESTOWN2.E2T, which establishes ownership of segment 2.\(^6\) 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 4758 PCI Cryptographic Coprocessor Developer’s Toolkit.

   - EMBURN2.L2T, 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 4758 PCI Cryptographic Coprocessor Developer’s Toolkit.

   - REMBURN2.R2T, which replaces an existing coprocessor operating system in segment 2. This file is shipped with the IBM 4758 PCI Cryptographic Coprocessor Developer’s Toolkit.

   - SUROWN2.S2T, 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.\(^7\) This file is shipped with the IBM 4758 PCI Cryptographic Coprocessor Developer’s Toolkit.

---

5. See Appendix E, “Using Signer and Packager” on page E-1 for details on the contents of these files.
6. The owner identifier assigned to segment 2 (typically 243 \(0xF3\)).
7. 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 accidentally or maliciously remove from a coprocessor a developer’s application and any data it has saved in nonvolatile memory, since SUROWN2.S2T removes any custom application installed on a coprocessor regardless of the application’s origin.
• ESTOWN3.E3T, which establishes ownership of segment 3. IBM assigns the developer\(^8\) 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.

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

3. 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/CRUZSIGN) 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 value of pAgentID->DeveloperID in any calls to xcAttach and the value of pRequestBlock->AgentID->DeveloperID in any calls to xcRequest should be changed to the owner identifier IBM assigns to the developer. 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 tool provides a great deal of flexibility and a discussion of its full potential is beyond the scope of this document. Appendix E, “Using Signer and Packager” on page E-1 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 assumes 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 card side debugger daemon (pdaemon) will run only in a development environment where the OA daemon has been started, and the owner ID corresponding to segment 2 is 3, and where segment 3 ownerID is 6. Additionally, pdaemon 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.

\(^8\) That is, an OEM or organization within an OEM.
The developer uses CRUSIGNR to create an EMBURN3 command that incorporates the application, IBM’s segment 2 owner ID, the developer’s owner ID, and the developer’s private key, for example:

```
CRUSIGNR EMBURN3 MYAPP.L3T
    part version description
    DEVSGNPU.CRT DEVSGNPP.KEY
    SEG3TK.BIN 3 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, and `oem3` is the owner identifier assigned to the developer. See Appendix E, “Using Signer and Packager” on page E-1 for details.

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:

```
CRUSIGNR REMBURN3 MYAPP.R3T
    part version description
    DEVSGNPU.CRT DEVSGNPP.KEY
    SEG3TK.BIN 3 title revision
    S3KDEVPP.KEY S3KDEVPU.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
```

---

9. Developers of a UDX that will be run on a PCI Cryptographic Coprocessor installed in an IBM zSeries server should replace “CRUSIGNR” with “CRUZSIGN” whenever the former appears in the examples in this chapter.

10. Typically 243 (0xF3). The proper value for a UDX that is to be run on a PCI Cryptographic Coprocessor installed in an IBM zSeries server is 2.

11. 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.
To create a package which will be imported to and activated on a PCI Cryptographic Coprocessor installed in a zSeries server, use CRUZPKG. First, the developer creates a SUROWN3 command:

```
CRUZSIGN SUROWN3 MYSUR3.S3T
   part version description
   DEVSGNPU.CRT DEVSGNPP.KEY
   S3KDEVPP.KEY
   ibm2 oem3
   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
```

Then the developer uses CRUZPKG to combine the SUROWN3, EMBURN3, and REMBURN3 commands with the ESTOWN3.E3T file provided by IBM to create a file which can be imported to and activated on a PCI Cryptographic Coprocessor installed in a zSeries server, for example:

```
CRUZPKG ESTOWN3=ESTOWN3.E3T SUROWN3=MYSUR3.S3T EMBURN3=MYAPP.L3T
   REMBURN3=MYAPP.R3T UDXID=000 S3_EMBURN_PREFERRED=NO
```

See Appendix E, “Using Signer and Packager” on page E-1 for details.

Place the IQYVP123.UDX file on a diskette in preparation for importing the CCA extensions to the zSeries PCI Cryptographic Coprocessor.12 (Note that the initial activation of the UDX will cause a SUROWN3 command for the IBM ownerID to be issued, followed by an EMBURN3 command with the developer’s ownerID. The EMBURN3 command will cause the coprocessor to clear data previously stored in BBRAM by code in segment 3.)

---

12. For more information on how to install the CCA extensions on a zSeries PCI Cryptographic Coprocessor, refer to zSeries 900 Support Element Operations Guide.
Appendix A.

An Overview of the Development Process

This appendix 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. Each step in this overview is listed under a heading that notes where in the body of the manual the step or tools it uses is described.

Preparing the Development Platform

1. Determine whether or not the coprocessor is empty:

   `<CLU> /logfile-directory/CLU.LOG ST`

   If coprocessor segment 1 is not 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 coprocessor segment 2 is UNOWNED, continue with step 2.

   If the owner identifier associated with segment 2 is 2, continue with step 3.

   If the owner identifier associated with segment 2 is 3, continue with step 4.

   If the owner identifier associated with segment 2 is neither 2 nor 3, it may not be possible to use the coprocessor for development. To do so requires the assistance of the owner of segment 2, who must supply a CLU file to surrender that ownership.

2. If coprocessor segment 2 is UNOWNED, the contents of segment 1 dictate how to proceed:

   - **Coprocessor in Factory-Fresh State** - 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 likely be rather cryptic. In this case, update the system software in segment 1 by loading CR1rrrss.CLU into the coprocessor, for example:

     `<CLU> /logfile-directory/CLU.LOG PL ~/xctk/etc/CR1rrrss.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, load TDVrrrss.CLU as indicated in “Segment 1 Current” on page A-2.

   - **Segment 1 Downlevel** - If segment 1 contains a downlevel version or revision of CCA segment 1, update the system software in segment 1 by loading CE1rrrss.CLU into the coprocessor, for example:

     `<CLU> /logfile-directory/CLU.LOG PL ~/xctk/etc/CE1rrrss.CLU`

     Then load TDVrrrss.CLU as indicated in “Segment 1 Current” on page A-2.

---

1. The examples in this appendix 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 `xctk/bin/linux`).
• **Segment 1 Current** - If segment 1 contains the appropriate version and revision of CCA segment 1, load a production version of the coprocessor operating system into segment 2 by loading TDVrrrss.CLU into the coprocessor, for example:

```
<CLU> /logfile-directory/CLU.LOG PL ~/xctk/etc/TDVrrrss.CLU
```

Then set the owner identifier for segment 3 by loading TE3rrrss.CLU into the coprocessor, for example:

```
<CLU> /logfile-directory/CLU.LOG PL ~/xctk/etc/TE3rrrss.CLU
```

Finally, set the public key associated with segment 3 and load the “reverse-then-echo” application by loading TL3rrrss.CLU into the coprocessor, for example:

```
<CLU> /logfile-directory/CLU.LOG PL ~/xctk/etc/TL3rrrss.CLU
```

If desired, confirm the software has been properly loaded by resetting the coprocessor to start the “reverse-then-echo” application loaded by TDVrrrss.CLU (see Appendix C, ”How to Reboot the IBM 4764” on page C-1 for details) and then running the host reverse-then-echo, for example:

```
HRE adapternumber text
```

The driver sends `text` to the reverse-then-echo application on the coprocessor identified by `adapternumber`, which reverses it and returns it to the driver. The driver prints the text received. This completes preparation of the coprocessor for use as a development platform. Continue with step 3.

If the owner identifier associated with segment 2 is 3 and the owner identifier associated with segment 3 is 6 and segment 2 is RUNNABLE but segment 3 is not, set the public key associated with segment 3 and load the “reverse-then-echo” application into segment 3 by loading TL3rrrss.CLU into the coprocessor, for example:

```
<CLU> /logfile-directory/CLU.LOG PL ~/xctk/etc/TL3rrrss.CLU
```

If desired, confirm the software has been properly loaded by resetting the coprocessor to start the “reverse-then-echo” application loaded by TDVrrrss.CLU (see Appendix C, ”How to Reboot the IBM 4764” on page C-1 for details) and then running the host reverse-then-echo, for example:

```
HRE adapternumber text
```

The driver sends `text` to the reverse-then-echo application on the coprocessor identified by `adapternumber`, which reverses it and returns it to the driver. The driver prints the text received. Continue with step 3.

If the owner identifier associated with segment 2 is 3 but the owner identifier associated with segment 3 is not 6, or segment 2 is not RUNNABLE relinquish ownership of segment 2 by loading TRSrrrss.CLU into the coprocessor, for example:

```
<CLU> /logfile-directory/CLU.LOG PL ~/xctk/etc/TRSrrrss.CLU
```

Continue with step 2.
Compiling, Assembling, and Linking

3. Compile and link the application under development. Specify the appropriate options to ensure debugging information is incorporated into the executable file produced.\(^2\)

Building JFFS2 Filesystem Images

4. Refer to “Building JFFS2 Filesystem Images” on page 3-7.

Downloading and Debugging

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

<CLU> /logfile-directory/CLU.LOG PL ~/xctk/etc/TL3rrrss.CLU

6. Download the file generated in step 8 to the coprocessor:\(^3\)

DRUID APP ~/xctk/ETC/S3KCLRPU.DRK ~/xctk/ETC/S3KCLRPP.DRK

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

8. Start the debugger and attach to the application:

ICATPCX

Refer to the IBM 4764 PCI-X Cryptographic Coprocessor Interactive Code Analysis Tool (ICAT) User’s Guide for more information.

If changes to the application prove necessary, make them and continue with step 3.

Testing an xC Application in a Production Environment

9. At some point it will be necessary to test the application in a production environment. To do so, remove any debugging code from the application, then rebuild the application by performing steps 3 through 6 of this procedure. In step 3 do not specify the options that incorporate debugging information in the .EXE file. In step 4, only one translation need be performed.

10. Load a production-level copy of the mkfs image into the coprocessor (using a CLU file or DRUID).

11. Clear any state saved in nonvolatile memory using the procedure described in step 7.

12. Download the file generated in step 6 to the coprocessor using the procedure described in step 8.

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

If changes to the application prove necessary, make them and continue with step 5.

Packaging and Releasing an xC Application

14. Reset the development coprocessor using TRSrrrss.CLU:

<CLU> /logfile-directory/CLU.LOG PL ~/xctk/etc/TRSrrrss.CLU

If it again becomes necessary to use the coprocessor for development, begin with step 2 of this procedure.

15. 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 4764 PCI-X Cryptographic Coprocessor CCA Support Program Installation Manual.

16. Generate three\(^4\) RSA keypairs using CRUSIGNR’s KEYGEN function:\(^5\):

CRUSIGNR KEYGEN 2 S3KDEVPP.KEY S3KDEVPU.KEY ~/xctk/etc/DFT_SKEL.TKN
CRUSIGNR KEYGEN 2 DEVSGNPP.KEY DEVSGNPU.KEY ~/xctk/etc/DFT_SKEL.TKN
CRUSIGNR KEYGEN 2 DEVPKGPP.KEY DEVPKGPU.KEY ~/xctk/etc/DFT_SKEL.TKN

---

2. The developer is free to choose the executable file name.
3. The first argument is the name of the file created in step 8.
4. A UDX that will be run on a PCI Cryptographic Coprocessor installed in an IBM zSeries server is packaged using the CRUZPKG utility, which does not require a keypair, so only two keypairs need to be generated in this case.
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.

17. Forward each public key generated in step 20 to IBM. Communicate the hash value of each public key (the hash value is also generated by the commands in step 20) to IBM by way of a separate channel to ensure an adversary has not replaced the developer’s public key file with another.

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 DEVP-KGPU.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.

b. The following files generated by CRUSIGNR. Only ESTOWN3.E3T is used during development of a UDX that will be run on a PCI Cryptographic Coprocessor installed in an IBM zSeries server.

- ESTOWN2.E2T, 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 4764 PCI-X Cryptographic Coprocessor Developer’s Toolkit.

- EMBURN2.L2T, 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 4764 PCI-X Cryptographic Coprocessor Developer’s Toolkit.

- REMBURN2.R2T, which replaces an existing coprocessor operating system in segment 2. This file is shipped with the IBM 4764 PCI-X Cryptographic Coprocessor Developer’s Toolkit.

- SUROWN2.S2T, 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 4764 PCI-X Cryptographic Coprocessor Developer’s Toolkit.

- ESTOWN3.E3T, which establishes ownership of segment 3. 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.

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

5. 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 S3DEVPVU.KEY\xctk\etc\DFT_SKEL.TKN
```

The appropriate actions should be taken to ensure the master key can be regenerated should the need arise. Refer to the IBM 4758 PCI Cryptographic Coprocessor CCA Support Program Installation Manual manual for details.

6. IBM typically provides a form for this purpose that can be returned by way of fax.

7. A UDX that will be run on a PCI Cryptographic Coprocessor installed in an IBM zSeries server is packaged using the CRUZPKG utility, which does not require a certificate.

8. See Appendix E, "Using Signer and Packager" on page E-1 for details on the contents of these files.

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

10. 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 accidentally or maliciously remove from a coprocessor a developer’s application and any data it has saved in nonvolatile memory, since SUROWN2.S2T removes any custom application installed on a coprocessor regardless of the application's origin.

11. That is, an OEM or organization within an OEM.
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/CRUZSIGN) 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 21 in zipped form.

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

23. Create an EMBURN3 command\(^\text{12}\) that incorporates the application, IBM’s segment 2 owner ID, the developer’s owner ID, and the developer’s unique keys:

\[
\begin{array}{c}
\text{CRUSIGNR EMBURN3 MYAPP.L3T} \\
\text{part version description} \\
\text{DEVSGNPU.CRT DEVSGNPP.KEY} \\
\text{SEG3TK.BIN 3 title revision} \\
\text{S3KDEVPP.KEY ESIGDEV.SIG} \\
\text{ibm2 oem3} \\
\text{1 1} \\
\text{a 0 b 0 c 0 d 0 e 0 0} \\
\text{x 0 0 65535 N} \\
\text{x 0 0 65535 N}
\end{array}
\]

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\(^\text{13}\), and oem3 is the owner identifier assigned to the developer. See Appendix E, “Using Signer and Packager” on page E-1 for details.

A user can use \(<\text{CLU}>\) 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:\(^\text{14}\)

\[
\begin{array}{c}
\text{CRUSIGNR REMBURN3 MYAPP.R3T} \\
\text{part version description} \\
\text{DEVSGNPU.CRT DEVSGNPP.KEY} \\
\text{SEG3TK.BIN 3 title revision} \\
\text{S3KDEVPP.KEY S3KDEVPP.KEY} \\
\text{ibm2 oem3} \\
\text{1 1} \\
\text{a 0 b 0 c 0 d 0 e 0 0} \\
\text{x 0 0 65535 N} \\
\text{x 0 0 65535 N} \\
\text{x 0 0 65535 N}
\end{array}
\]

\(^{12}\)Developers of a UDX that will be run on a coprocessor installed in an IBM zSeries server should replace “CRUSIGNR” with “CRUZSIGN” whenever the former appears in the examples of this chapter.

\(^{13}\)Typically 243 (0xF3). The proper value for a UDX that is to be run on a PCI Cryptographic Coprocessor installed in an IBM zSeries server is 2.

\(^{14}\)The public key downloaded with the earlier version of the application must be the public key in S3KDEVPP.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.
24. For zSeries only, create a SUROWN3 command that incorporates IBM’s segment 2 owner ID, the developer’s owner ID, and the developer’s unique keys:

```
CRUZSIGN SUROWN3 MYSUR3.S3T
part version description
DEVSIGNPU.CRT DEVSGNPP.KEY
S3KDEVPP.KEY
ibm2 oem3
  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
```

25. To create a package which will be imported to and activated on a PCI-X Cryptographic Coprocessor installed in a zSeries server, use CRUZPKG to combine the SUROWN3, EMBURN3, and REMBURN3 commands with the ESTOWN3.E3T file provided by IBM, for example:

```
CRUZPKG ESTOWN3=ESTOWN3.E3T SUROWN3=MYSUR3.S3T EMBURN3=MYAPP.L3T REMBURN3=MYAPP.R3T UDXID=000 S3_EMBURN_PREFERRED=NO
```

See Appendix E, “Using Signer and Packager” on page E-1 for details.

Place the IQYVP123.UDX file on a diskette in preparation for importing the CCA extensions to the zSeries PCI-X Cryptographic Coprocessor.\(^\text{15}\) (Note that the initial activation of the UDX will cause a SUROWN3 command for the IBM ownerID to be issued, followed by an EMBURN3 command with the developer’s ownerID. The EMBURN3 command will cause the coprocessor to clear data previously stored in BBRAM by code in segment 3.)

\(^\text{15}\): For more information on how to install the CCA extensions on a zSeries PCI Cryptographic Coprocessor, refer to zSeries 900 Support Element Operations Guide.
An Overview of the Development Process
Appendix B.

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.

Syntax

<CLU> logfilename {PL | RS | SS | ST | VA | EX} [coprocessornumber][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.

CLU also appends log information in machine-readable form to a file with the same name as the log file name and the extension .MRL.

- The second argument specifies the operation CLU is to perform. Recognized values are as follows:
  - **PL** - Download a file containing software and/or commands to the coprocessor.
  - **RS** - Reset the coprocessor.
  - **SS** - Print information about every coprocessor installed in a host and the application each coprocessor contains.
  - **ST** - Print information about the coprocessor and the software it contains.
  - **VA** - Print and validate information about the coprocessor and the software it contains.
  - **EX** - Exit <CLU>.

- More than one coprocessor may be installed in a host. **coprocessornumber** identifies the coprocessor with which CLU is to interact. The default is 0.

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 name appears only if the **PL** or **VA** operation is specified.

If no arguments are provided CLU runs interactively and prompts for them.
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:

0  OK.
1  Command line parameters not valid.
2  Cannot access the coprocessor. Be sure that the coprocessor and its driver have been properly installed.
3  Check the utility log file for an abnormal condition report.
4  No coprocessor installed. Be sure that the coprocessor and its driver have been properly installed.
5  Invalid coprocessor number specified.
6  A data file is required with this command.
7  The data file specified with this command is incorrect or invalid.
Appendix C.
How to Reboot the IBM 4764

An IBM 4764 can be rebooted in any of several ways:

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

   <CLU> /logfile-directory/CLU.LOG RS

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)
      /xctk/bin/linux/xcunload
   b. Subsequently reloading the driver (as root)
      /xctk/bin/linux/xcload

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

4. If an application on the host calls xcOpenAdapter and the card needs to be rebooted, the device driver will do so.

¹ The examples in this appendix 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 xctk/bin/linux).
Appendix D. Building xC Applications with the Developer’s Toolkit Makefiles

This appendix describes how to use and customize the makefiles shipped with the Developer’s Toolkit. These makefiles make it relatively straightforward to build the coprocessor-side portion of an xC application from a command prompt. (The host-side portion of an xC application may be built in the same manner as any other Linux application. The only requirement is to use the appropriate compiler options to ensure the directories listed in “Include File Directory Search Order” on page 3-5 are searched in the proper order.)

All applications targeted for execution on an xCrypto card must be built with a cross compiler. This cross compiler should be downloaded and installed separately. All sample applications have been tested with the MCP cross compilers.

Each sample application has an associated card-side makefile which can be used to build an application that can then be incorporated into an mkfs image that can be loaded onto the xCrypto adapter using DRUID.

Before invoking any of the card-side makefiles, ensure that the CROSS environment variable is set in the makefile or as an environment variable. The cross environment variable should be set so that $(CROSS)gcc points to the cross compiler’s version of gcc. If the CROSS environment variable is not set, the application may compile, but will not link, often producing a message “<stub library> could not read symbols. invalid operation”. If this occurs, remove any object files, set the CROSS environment variable appropriately and rebuild.

When linking a cross-compiled xCrypto application, use the following options:

```
-L$(XCTK_FS_ROOT)/lib/card/linux/gcc/stubs followed by -l <stub library> for each library to be linked against. Please note that the name of the stub library usually does not contain the “lib” prefix or the “.so” suffix. For example to link against libxccomap_stub.so, use -lxccomapi_stub. Refer to the man page of the cross compiler for more details.
```

Once an application has been successfully built, it can be combined into an mkfs image to be loaded onto the card by way of DRUID. See “Building JFFS2 Filesystem Images” on page 3-7 and “Downloading and Debugging” on page 3-9 for more details.
Appendix E. Using Signer and Packager

This appendix describes the use of the signer and packager utilities and explains why the design of the coprocessor makes these utilities necessary.\(^1\)

Coprocessor Memory Segments and Security

The design for the *IBM 4764 PCI-X Cryptographic Coprocessor* was motivated by the need to simultaneously satisfy 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.\(^2\)

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 use commands that affect segment 1. The remainder of this chapter therefore deals with changes to segments 2 and 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.\(^3\) IBM owns segment 1 and issues an owner identifier to any party that is developing code to be loaded into segment 2. An owner of segment 2 issues an owner identifier to any party that is developing code that is to be loaded into segment 3.

---

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.

2. 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.

3. 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 4764 PCI-X Cryptographic Coprocessor Custom Software Interface Reference*. 
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 battery-backed RAM 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:

1. If segment 2’s state changes to UNOWNED for any reason, segment 3’s state is also changed to UNOWNED.
2. 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.
3. 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).

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.

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.

Miniboot changes the state of a segment in response to certain commands Miniboot receives from the host. Figure F-1 shows the state transitions for segment 2 and Figure F-2 on page F-4 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

---

4. The rules can be expressed in the following manner: 1) a segment can’t be owned if its “parent” isn’t owned and 2) a segment can’t be RUNNABLE if its parent isn’t RUNNABLE.

5. Segment 3’s state is maintained in BBRAM. Information on how to access segment 3’s state will appear in the forthcoming Miniboot interface document.
several Miniboot commands. A developer who makes a change to an application during development must use the Signer and the Packager 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.

* 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 E-17 for details.

Figure E-1. State Transitions for Segment 2

6. Or segment 2 if the developer is writing an operating system for the coprocessor.
The Signer Utility (CRUSIGNR.EXE)

The Signer utility (CRUSIGNR.EXE or CRUZSIGN.EXE) 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.7

The signer utility used to develop a UDX that will be run on a PCI-X Cryptographic Coprocessor installed in an IBM zSeries server is CRUZSIGN.EXE. Unless otherwise noted, the syntax of the CRUZSIGN command is identical to the syntax of the CRUSIGNR command.

---

7. 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 xctk/bin/linux).
Syntax

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

CRUSIGNR [-S profile] function [arguments] [-Q]

CRUSIGNR [-S profile]

CRUSIGNR ignores the case of its options (for example, -Q and -q are equivalent). Options may be prefixed with a hyphen or a forward slash (for example, -Q and /Q are equivalent).

The -Q option suppresses all prompts and messages (including error messages). If -Q is specified and CRUSIGNR finds it necessary to issue a prompt, the program ends in failure.

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 Appendix E, “CCA Roles for Signer and Packager” on page E-25 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.

CRUSIGNR does not support the -S option.

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 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.

The third form causes CRUSIGNR to issue a prompt for the function and each required argument and read the data from stdin.

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).

CRUSIGNR writes messages to a file named $SIGNER.RSP.

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.
**Signer Operations**
The first argument to CRUSIGNR specifies the Miniboot command CRUSIGNR is to generate or the cryptographic function CRUSIGNR is to perform and may be one of the following: 8

**Signer Cryptographic Functions**

- **KEYGEN** Generate an RSA key pair.
- **KEYCERT** Create a certificate for a file containing an RSA public key.
- **HASH_GEN** Generate the hash for a file using the SHA1 algorithm.
- **HASH_VER** Verify the hash of a file using the SHA1 algorithm.

**Signer Miniboot Command Functions**

- **EMBURN2** Load software into segment 2.
- **REMBURN2** Replace the software in segment 2.
- **SUROWN2** Surrender ownership of segment 2.
- **ESIG3** Generate emergency signature for segment 3.
- **ESTOWN3** Establish ownership of segment 3.
- **EMBURN3** Load software into segment 3.
- **REMBURN3** Replace the software in segment 3.
- **SUROWN3** Surrender ownership of segment 3.

**Signer Miscellaneous Functions**

- **HELP** Display instructions about how to use the program.

---

8. Numbers may be used in place of the words listed, as follows: 0 (HELP), 1 (KEYGEN), 2 (HASH_GEN), 3 (HASH_VER), 4 (IBM_INIT), 5 (SIGNFILE), 6 (KEYCERT), 7 (DATACERT), 8 (FCVCERT), 9 (REMBURN1), 10 (REMBURN2), 11 (REMBURN3), 12 (EMBURN2), 13 (EMBURN3), 14 (ESTOWN2), 15 (ESTOWN3), 16 (SUROWN2), 17 (SUROWN3), 18 (ESIG2), 19 (ESIG3), and 20 (RECERT).
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) and they are not otherwise described.

DATACERT
ESIG2
ESTOWN2
FCVCERT
IBM_INIT
KEYCERT
RECERT
REMBURN1
SIGNFILE

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, and briefly discusses how it is used during the development process.

**EMBURN2 - Load Software into Segment 2**

Syntax

```
EMBURN2 out_fn filedesc_args sigkey_args image_args privkey_fn esig_fn
ownid trust1_fl type1_target_args
```

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 this 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 or segment 3.

This command takes the following arguments:

- `out_fn` is the name of the file CRUSIGNR generates to hold the EMBURN2 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` provides certain descriptive information that is incorporated into the output file. See “File Description Arguments” on page E-15 for details.
- `sigkey_args` 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 E-15 for details.
• *image_args* specifies the name of the file that contains the code to be loaded into segment 2 and provides certain descriptive information about the code that is also downloaded to the coprocessor. See “Image File Arguments” on page E-16 for details.

• *privkey_fn* 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 2.\(^9\) This key is downloaded to the coprocessor and is used to authenticate subsequent commands that affect segment 2. 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 EMBURN2 command if the validation fails.

• *esig_fn* 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 *privkey_fn* file and includes a hash of the emergency signature information enciphered using the private key corresponding to the public key associated with segment 1. The coprocessor uses the public key associated with segment 1 to validate the hash and rejects the EMBURN2 command if the validation fails.

• *ownid* is the owner identifier currently associated with segment 2. The coprocessor compares this value to the owner identifier stored in the coprocessor and rejects the EMBURN2 command if the two identifiers are not equal.

• *trust1_fl* indicates whether or not segment 2’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 E-16 for details.

• *type2_target_args* specifies certain conditions that the coprocessor checks before it accepts the new segment 2 information. See “Targeting Arguments” on page E-17 for details.

---

**EMBURN3 - Load Software into Segment 3**

**Syntax**

```
EMBURN3 out_fn filedesc_args sigkey_args image_args privkey_fn esig_fn
    seg2_ownid seg3_ownid trust1_fl trust2_fl type2_target_args
```

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 this 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* 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* provides certain descriptive information that is incorporated into the output file. See “File Description Arguments” on page E-15 for details.

• *sigkey_args* 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 E-15 for details.

---

\(^9\) If desired, the new public key may be the same as the public key currently associated with segment 2, if there is one.
• **image_args** 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 E-16 for details.

• **privkey_fn** 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 3.10 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** 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 **privkey_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_ownid** 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_ownid** 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** 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 E-16 for details.

• **trust2_fl** 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 E-16 for details.

• **type2_target_args** specifies certain conditions that the coprocessor checks before it accepts the new segment 3 information. See “Targeting Arguments” on page E-17 for details.

### ESIG3 - Build Emergency Signature for Segment 3

**Syntax**

```
ESIG3 out_fn pubkey_fn privkey_fn seg2_ownid seg3_ownid type2_target_args
```

ESIG3 creates a file containing an “emergency signature” that can be provided as an argument to the EMBURN3 command. A developer will only need to use this command if the developer is writing an operating system for the coprocessor: the developer owns segment 2 and uses the ESIG3 command to certify a public key supplied by an agent developing a segment 3 application to run on top of the operating system.

This command takes the following arguments:

• **out_fn** is the name of the file CRUSIGNR generates to hold the emergency signature. Path information must also be provided if the file is not in the current directory. By convention, the file extension is BIN.

• **pubkey_fn** 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.

---

10. If desired, the new public key may be the same as the public key currently associated with segment 3, if there is one.
• `privkey_fn` 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 2. 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 2 to validate the hash and rejects the EMBURN3 command that contains the emergency signature if the validation fails.

• `seg2_ownid` 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 that contains the emergency signature if the two identifiers are not equal.

• `seg3_ownid` is the owner identifier associated with segment 3. This identifier is assigned by the developer (that is, the segment 2 owner).

• `type2_target_args` specifies certain conditions that the coprocessor checks before it accepts the new segment 3 information provided by the EMBURN3 command that contains the emergency signature. See “Targeting Arguments” on page E-17 for details.

**ESTOWN3 - Establish Ownership of Segment 3**

**Syntax**

```
ESTOWN3 out_fn filedesc_args sigkey_args privkey_fn seg2_ownid seg3_ownid
    type2_target_args
```

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

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

This command takes the following arguments:

• `out_fn` is the name of the file CRUSIGNR generates to hold the ESTOWN3 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` provides certain descriptive information that is incorporated into the output file. See “File Description Arguments” on page E-15 for details.

• `sigkey_args` 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 E-15 for details.

• `privkey_fn` 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 2. 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 2 to validate the hash and rejects the ESTOWN3 command if the validation fails.

• `seg2_ownid` is the owner identifier currently associated with segment 2. The coprocessor compares this value to the owner identifier stored in the coprocessor and rejects the ESTOWN3 command if the two identifiers are not equal.

• `seg3_ownid` is the owner identifier to be associated with segment 3. This identifier is assigned by the developer (that is, the segment 2 owner).

• `type2_target_args` specifies certain conditions that the coprocessor checks before it accepts the ESTOWN3 command. See “Targeting Arguments” on page E-17 for details.
**HASH GEN - Generate Hash for File**

Syntax

```
HASH_GEN in_fn out_fn
```

HASH_GEN uses the SHA1 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` and `out_fn` must include path information if either file is not in the current directory.

**HASH_VER - Verify Hash of File**

Syntax

```
HASH_VER data_fn hash_fn
```

HASH_VER verifies that the hash in the file `hash_fn` matches the hash the HASH_GEN function would generate given `data_fn` as input and issues a message indicating the result (unless the -Q option is specified when CRUSIGNR is invoked). The `hash_fn` file has the same format as the `out_fn` file generated by the HASH_GEN function.

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

**KEYGEN - Generate RSA Key Pair**

Syntax

```
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}
```

KEYGEN generates an RSA keypair and saves it in the file `keypair_fn`. The public key is also saved in the file `pubkey_fn` and the hash of the public key\(^{11}\) is saved in a file with the same name as `pubkey_fn` and extension HSH. The file `skeleton_fn` 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` contains a DES IMPORTER or DES EXPORTER key-encrypting key.

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

CRUSIGNR uses the PKA_Key_Generate CCA verb to generate the keypair. The first argument to KEYGEN 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 key-pair 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).
- 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.

---

\(^{11}\) The KEYGEN command computes the hash in the same manner and stores it in the same format as the HASH_GEN command.
Refer to the *IBM 4764 PCI-X CCA Basic Services Reference and Guide* for details on the format of skeleton files and the PKA_Key_Generate CCA verb.

**REMBURN2 - Replace Software in Segment 2**

**Syntax**

```
REMBURN2 out_fn filedesc_args sigkey_args image_args pubkey_fn privkey_fn
  ownid trust1_fl type2_target_args type3_csign_args
```

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.

This command takes the following arguments:

- **out_fn** is the name of the file CRUSIGNR generates to hold the REMBURN2 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** provides certain descriptive information that is incorporated into the output file. See “File Description Arguments” on page E-15 for details.
- **sigkey_args** 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 E-15 for details.
- **image_args** specifies the name of the file that contains the code to be loaded into segment 2 and provides certain descriptive information about the code that is also downloaded to the coprocessor. See “Image File Arguments” on page E-16 for details.
- **pubkey_fn** is the name of the file that contains the public key to be associated with segment 2. 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 2.
- **privkey_fn** 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 2. 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 2 to validate the hash and rejects the REMBURN2 command if the validation fails.
- **ownid** is the owner identifier associated with segment 2. The coprocessor compares this value to the owner identifier stored in the coprocessor and rejects the REMBURN2 command if the two identifiers are not equal.
- **trust1_fl** indicates whether or not segment 2’s state is to be changed to UNOWNED if the contents of segment 1 change. See “Trust and Countersignature Arguments” on page E-16 for details.
- **type2_target_args** specifies certain conditions that the coprocessor checks before it accepts the new segment 2 information. See “Targeting Arguments” on page E-17 for details.
- **type3_csign_args** specifies certain conditions that determine whether or not Miniboot changes segment 3’s state to RELIABLE_BUT_UNRUNNABLE while updating segment 2. See “Trust and Countersignature Arguments” on page E-16 for details.

---

12. If desired, the new public key may be the same as the public key currently associated with the segment.
13. The change to segment 3’s state and the updates of segment 2 are performed automatically.
REMBURN3 - Replace Software in Segment 3

Syntax

\texttt{REMBURN3 \ out\_fn\ \ filedesc\_args\ \ sigkey\_args\ \ image\_args\ \ pubkey\_fn\ \ privkey\_fn}\n\texttt{\ seg2\_ownid\ \ seg3\_ownid\ \ trust1\_fl\ \ trust2\_fl\ \ type3\_target\_args}

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:

- \texttt{out\_fn} 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.
- \texttt{filedesc\_args} provides certain descriptive information that is incorporated into the output file. See “File Description Arguments” on page E-15 for details.
- \texttt{sigkey\_args} 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 E-15 for details.
- \texttt{image\_args} 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 E-16 for details.
- \texttt{pubkey\_fn} 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.
- \texttt{privkey\_fn} 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 \texttt{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.
- \texttt{seg2\_ownid} 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.
- \texttt{seg3\_ownid} 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{trust1\_fl} 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 E-16 for details.
- \texttt{trust2\_fl} 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 E-16 for details.
- \texttt{type3\_target\_args} specifies certain conditions that the coprocessor checks before it accepts the new segment 3 information. See “Targeting Arguments” on page E-17 for details.

14. If desired, the new public key may be the same as the public key currently associated with the segment.
**SUROWN2 - Surrender Ownership of Segment 2**

**Syntax**

```
SUROWN2 out_fn filedesc_args sigkey_args privkey_fn ownid type2_target_args
```

SUROWN2 creates a file that directs Miniboot to surrender ownership of segment 2, that is, to change segment 2's state to UNOWNED.\(^\text{15}\) 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).

This command takes the following arguments:

- **out_fn** is the name of the file CRUSIGNR generates to hold the SUROWN2 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** provides certain descriptive information that is incorporated into the output file. See “File Description Arguments” on page E-15 for details.
- **sigkey_args** 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 E-15 for details.
- **privkey_fn** 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 2. 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 2 to validate the hash and rejects the SUROWN2 command if the validation fails.
- **ownid** is the owner identifier associated with segment 2. The coprocessor compares this value to the owner identifier stored in the coprocessor and rejects the SUROWN2 command if the two identifiers are not equal.
- **type2_target_args** specifies certain conditions that the coprocessor checks before it accepts the SUROWN2 command. See “Targeting Arguments” on page E-17 for details.

**SUROWN3 - Surrender Ownership of Segment 3**

**Syntax**

```
SUROWN3 out_fn filedesc_args sigkey_args image_args privkey_fn seg2_ownid seg3_ownid type3_target_args
```

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** 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.

---

\(^{15}\)This also changes segment 3’s state to UNOWNED.
• **filedesc_args** provides certain descriptive information that is incorporated into the output file. See “File Description Arguments” on page E-15 for details.

• **sigkey_args** 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 E-15 for details.

• **privkey_fn** 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_ownid** 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.

• **seg3_ownid** 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.

• **type3_target_args** specifies certain conditions that the coprocessor checks before it accepts the SUROWN3 command. See “Targeting Arguments” on page E-17 for details.

### 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:

\[
\text{partnumber} \ \text{ECnumber} \ \text{description}
\]

where

• **partnumber** is a string containing up to eight characters. The string is padded with blanks to the full eight characters before it is incorporated into the output file.

• **ECnumber** is a string containing up to eight characters. The string is padded with blanks to the full eight characters before it is incorporated into the output file.

• **description** 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.

**partnumber** is intended to uniquely identify a particular component of a software package (for example, a particular application in a suite). **ECnumber** is intended to identify the revision level of the component.

### 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).\(^{16}\) The format of these arguments is

\[
sigkey_cert_fn \ \text{sigkey_fn}
\]

where

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

• **sigkey_fn** is the name of the file containing the 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.

---

16. The signature key arguments are for the purposes of administrative control. Core security is provided by verification of other signatures and is performed inside the coprocessor.
When CRUSIGNR creates an output file containing a Miniboot command, CRUSIGNR incorporates the certificate from the `sigkey_cert_fn` file, computes a hash of the output file, encrypts the hash with the private key in the `sigkey_fn` file, and appends the encrypted hash to the output file. When CLU processes the file, CLU computes the hash of the relevant portions of the file, extracts the public key from the certificate using the public key corresponding to the private key used to create the certificate\(^\text{17}\), uses the extracted key to decrypt the hash, and verifies that the two hash values match.

### 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 are 1 (for code that targets the IBM 4758 Model 001 and Model 013) and 2 (for code that targets the IBM 4758 Model 002 and Model 023), and 3 (for code that targets the IBM 4764 PCI-X 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 PCI Cryptographic Coprocessor installed in an IBM zSeries server, the CRUZSIGN utility enforces certain restrictions on the `title` argument: CRUZSIGN inserts the characters “UDX” before the first three characters of the title; (note that the title string can only occupy the first 36 bytes of the string) bytes 37-48 of the title string will be overlaid with the timestamp of the ROD 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 E-17 for details.

### Trust and Countersignature Arguments

Recall that one of the primary design goals for the **IBM 4764 PCI-X 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 1 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 E-1 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 S’, S always trusts changes to S’. A REMBURN command that changes the contents of S does not affect the state of S.

\(^{17}\) The public key is compiled into CLU.
If a segment S specifies a trust flag of 2 with respect to a more privileged segment S, S never trusts changes to S. A REMBURN command that changes the contents of S changes the state of S to RELIABLE_BUT_UNRUNNABLE or to UNOWNED.\textsuperscript{18} Note that an EMBURN command that changes the contents of S causes S’s state 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 S, S trusts changes to S only if the new image of S 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 format of the countersignature arguments is

\[
\{\text{NoCSig2} / \text{privkey_fn type2_target_args} \} \{\text{NoCSig3} / \text{privkey_fn type3_target_args} \}
\]

where

- **NoCSig2** indicates there is no countersignature provided by segment 2. This option is only applicable to the REMBURN1 command and must be specified exactly as shown (that is, case is important).
- **NoCSig3** indicates there is no countersignature provided by segment 3. This option applies to the REMBURN1 and REMBURN2 commands and must be specified exactly as shown (that is, case is important).
- **privkey_fn** 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 the segment that requires the countersignature (for example, the public key for segment 3 if privkey_fn appears instead of NoCSig3). If privkey_fn appears, the segment providing the key can also provide a set of targeting arguments for the segment providing the key and each more privileged segment. See “Targeting Arguments” on page E-17 for details.

### Targeting Arguments

The CRUSIGNR functions that generate Miniboot commands (EMBURN2, EMBURN3, ESIG3, ESTOWN3, REMBURN2, REMBURN3, SUROWN2, 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

\[
\text{RTCid} \text{ RTCid_mask} \text{ VPDserno} \text{ VPDserno_mask} \text{ VPDpartno} \text{ VPDpartno_mask} \text{ VPDecno} \text{ VPDecno_mask} \text{ VPDecflags} \text{ VPDecflags_mask} \text{ bootcount_fl} \text{ [bootcount_left[bootcount_right]] seg1_info [seg2_info[seg3_info]]}
\]

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.\textsuperscript{19} 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.

\textsuperscript{18} See Figure E-1 on page E-3 and Figure E-1 on page E-4.
\textsuperscript{19} That is, the value xcGetConfig returns in pInfo->AdapterID. Refer to the IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Interface Reference for details.
CRUSIGNR logically ANDs RTCid with the hexadecimal number derived from RTCid_mask and passes the result to the coprocessor. The coprocessor logically ANDs the serial number incorporated in the coprocessor’s real-time clock chip with the hexadecimal number derived from RTCid_mask and compares the result to the value generated by CRUSIGNR. If they are not equal, the Miniboot command that incorporates the targeting information is rejected.

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

- VPDserno and VPDserno_mask specify a range of permitted values for the coprocessor’s IBM serial number. 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. 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.

20. That is, the value xcGetConfig returns in pInfo->VPD.sn. Refer to the IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Interface Reference for details.
21. That is, the value xcGetConfig returns in pInfo->VPD.pn. Refer to the IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Interface Reference for details.
CRUSIGNR logically ANDs \texttt{VPDecno} with the hexadecimal number derived from \texttt{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 \texttt{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 \texttt{VPDecno} and 0 for \texttt{VPDecno\_mask}.

- \texttt{VPDecno} and \texttt{VPDecno\_mask} specify a range of permitted values for the coprocessor’s IBM engineering change level. 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 \texttt{VPDecno\_mask} must be either ASCII 0 or ASCII 1. CRUSIGNR uses \texttt{VPDecno\_mask} to construct a 7-byte hexadecimal number. Each byte in the hexadecimal number is set to 0xFF if the corresponding character in \texttt{VPDecno\_mask} is ASCII 1 and is set to 0x00 otherwise.

CRUSIGNR logically ANDs \texttt{VPDecno} with the hexadecimal number derived from \texttt{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 \texttt{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 \texttt{VPDecno} and 0 for \texttt{VPDecno\_mask}.

- \texttt{VPDflags} and \texttt{VPDflags\_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 \texttt{VPDflags\_mask} must be either ASCII 0 or ASCII 1. CRUSIGNR uses \texttt{VPDflags\_mask} to construct a 32-byte hexadecimal number. Each byte in the hexadecimal number is set to 0xFF if the corresponding character in \texttt{VPDflags\_mask} is ASCII 1 and is set to 0x00 otherwise.

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

If a command is intended to apply to all possible coprocessors, specify an arbitrary character for \texttt{VPDflags} and 0 for \texttt{VPDflags\_mask}.

\begin{itemize}
\item \texttt{VPDecno} and \texttt{VPDecno\_mask} specify a range of permitted values for the coprocessor’s IBM engineering change level. Each of these arguments is a string and may contain as many as seven characters. The arguments should have the same length.
\item \texttt{VPDflags} and \texttt{VPDflags\_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.
\end{itemize}

\textbf{Using Signer and Packager page E-19}

\footnotesize
\begin{itemize}
\item That is, the value xcGetConfig returns in pInfo->VPD.ec. Refer to the IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Interface Reference for details.
\item That is, the value xcGetConfig returns in the last sixteen bytes of pInfo->VPD.reserved. Refer to the IBM 4764 PCI-X Cryptographic Coprocessor Custom Software Interface Reference for details.
\end{itemize}
• 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 non-zeroized state. It is set to zero if the coprocessor detects an attempt to compromise the coprocessor’s security. bootcount_fl, bootcount_left, and bootcount_right specify a range of permitted values for the left and right counts.

bootcount_fl may be 0, 1, or 2. If bootcount_fl is 0, 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.

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 E-16 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.

24. 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.
25. The DRUID utility displays the current left and right counts each time it is run.
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 E-16 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, ESIG3, ESTOWN3, REMBURN2, and SUROWN2 commands incorporate 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, and the REMBURN2 command may include type 3 targeting information in its countersignature.
The Packager Utility (CRUPKGR.EXE)

The packager utility (CRUPKGR.EXE) 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.26

Syntax

CRUPKGR -H

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

CRUPKGR [-S profile] [sigkey_args [num_files [in_fn_list [out_fn [outtype [filedesc_args]]]]] [-Q]

CRUPKGR ignores the case of its options (for example, -H and -h are equivalent). Options may be prefixed with a hyphen or a forward slash (for example, -Q and /Q are equivalent).

The -Q option suppresses all prompts and messages (including error messages). If -Q is specified and CRUPKGR finds it necessary to issue a prompt, the program ends in failure.

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 E-25 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 displays instructions about how to use the program. In addition to -H and its equivalents, the program accepts ?, -?, and /?.

The second 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 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.

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** 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 E-15 for details.
- **num_files** 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** 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** 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.

26. 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 xctk/bin/linux).
outtype specifies how the output file is intended to be used. Recognized values are as follows:
- 2 for segment 1
- 3 for segment 2
- 4 for segment 3
- 5 for the Hardware Lock Monitor
- 6 for the Function Control Vector
- 7 for a key certificate (KEYCERT)
- 8 for a data certificate (DATACERT)
- 9 for any other image
- 10 for reload segment 1 (REMBURN1)
- 11 for reload segment 2 (REMBURN2)
- 12 for reload segment 3 (REMBURN3)
- 13 for reload segment 2 (EMBURN2)
- 14 for reload segment 3 (EMBURN3)
- 15 for establish ownership of segment 2 (ESTOWN2)
- 16 for establish ownership of segment 3 (ESTOWN3)
- 17 for surrender ownership of segment 2 (SUROWN2)
- 18 for surrender ownership of segment 3 (SUROWN3)
- 19 for recertify the coprocessor (RECERT)

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 provides certain descriptive information that is incorporated into the output file. See “File Description Arguments” on page E-15 for details.

The Packager Utility (CRUZPKG.EXE)

The zSeries packager utility (CRUZPKG.EXE) generates a file containing four Miniboot commands (each generated by CRUZSIGN). The resulting package file (IQYVPxxx.UDX) is of a form which can be imported to and activated on a PCI Cryptographic Coprocessor which is installed in an IBM zSeries server. This section describes the syntax of the CRUZPKG command and explains the function of the various CRUZPKG options.

Syntax

CRUZPKG -H
CRUZPKG -F parm_file_name [-Q]

CRUZPKG ([ESTOWN3=x] | [SUROWN3=x] | [EMBURN3=x] | [REMBURN3=x] | [UDXID=x] | [S3_EMBURN_PREFERRED=x])* [COMMENT=x] [-Q]

CRUZPKG ignores the case of its options (for example, -H and -h are equivalent). Options may be prefixed with a hyphen or a forward slash (for example, -Q and /Q are equivalent).

The -Q option suppresses all prompts and messages (including error messages). If -Q is specified and CRUZPKG finds it necessary to issue a prompt, the program ends in failure.

The first form displays instructions about how to use the program. In addition to -H and its equivalents, the program accepts ?, -?, and /?.

27. 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 xck/bin/zSeries).
The second form causes CRUZPKG 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 and may not span lines. Any line that contains the character # in column one will be considered a comment and will be ignored by the packager. Once the file is exhausted, CRUZPKG issues a prompt for each additional argument required and reads the argument from stdin.

The third form causes CRUZPKG to read arguments from the command line. Once the command line is exhausted, CRUZPKG issues a prompt for each additional argument required and reads the argument from stdin.

If CRUZPKG 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).

CRUZPKG takes the following arguments:

The first four arguments listed below are used to specify the names of the files which CRUZPKG is to combine into a single image. Path information must also be provided if the file is not in the current directory. The names of the files are each preceded by a keyword which identifies the intended use of the file. The file names and corresponding keywords may be specified in any order. The keywords are as follows:

- **ESTOWN3** - The file identified by this keyword is the ESTOWN3.E3T file received from IBM which establishes the owner of segment 3 as the owner identifier assigned by IBM to the developer.

- **SUROWN3** - The file identified by this keyword is the signed SUROWN3 command created by the developer which incorporates IBM’s segment 2 owner ID, the developer’s owner ID, and the developer’s unique keys.

- **EMBURN3** - The file identified by this keyword is the signed EMBURN3 command created by the developer which incorporates the application, IBM’s segment 2 owner ID, the developer’s owner ID, and the developer’s unique key.

- **REMBURN3** - The file identified by this keyword is the signed REMBURN3 command created by the developer which incorporates the application, IBM’s segment 2 owner ID, the developer’s owner ID, and the developer’s unique key.

The default values if these keywords are not specified are as follows:

- ESTOWN3=ESTOWN3.DAT
- SUROWN3=SUROWN3.DAT
- EMBURN3=EMBURN3.DAT
- REMBURN3=REB3.DAT

- **UDXID** - This keyword is used to specify a unique identifier for the custom Segment 3 image. The value specified must be three alphanumeric characters. These three characters will be used as a part of the file name of the package file. The output of CRUZPKG will be a file named IQYVPxxx.UDX, where xxx are the three characters specified via the UDXID keyword. The default is UDXID=000.

- **S3_EMBURN_PREFERRED** - Specifies whether activation of the Segment 3 image containing the UDX application is always to be performed via an EMBURN3 command or whether a REMBURN3 command is to be attempted when possible. Allowable values for the option are:

  - S3_EMBURN_PREFERRED=NO

Indicates that activation of the Segment 3 image containing the UDX application is to be performed via a REMBURN3 command in preference to an EMBURN3 command whenever possible. Specification of S3_EMBURN_PREFERRED=NO will ensure that any state information or other data saved in nonvolatile memory by code in Segment 3 will be preserved if possible.
**S3_EMBURN_PREFERRED=YES**
Indicates that activation of the Segment 3 image containing the UDX application is always to be performed via an EMBURN3 command. Specification of S3_EMBURN_PREFERRED=YES causes the coprocessor to clear data previously stored in nonvolatile memory by code in Segment 3. The default is S3_EMBURN_PREFERRED=NO.

- **COMMENT=** Provides descriptive information which is incorporated into the output file. The description specified via this keyword is a string containing up to 80 characters. When CRUZPKG is invoked from the command line, if the COMMENT keyword is specified, it must be the last keyword specified. The default is a null comment.

### 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

**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
Appendix F.
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.

Copying and Distributing Softcopy Files
For online versions of this book, we authorize you to:

• Copy, modify, and print the documentation contained on the media, for use within your enterprise, provided you reproduce the copyright notice, all warning statements, and other required statements on each copy or partial copy.

• Transfer the original unaltered copy of the documentation when you transfer the related IBM product (which may be either machines you own, or programs, if the program’s license terms permit a transfer). You must, at the same time, destroy all other copies of the documentation.

You are responsible for payment of any taxes, including personal property taxes, resulting from this authorization.

THERE ARE NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING THE WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Some jurisdictions do not allow the exclusion of implied warranties, so the above exclusion may not apply to you.

Your failure to comply with the terms above terminates this authorization. Upon termination, you must destroy your machine readable documentation.
Trademarks
The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States or other countries or both:

AIX
IBM
eServer
z/OS
zSeries

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

FerriShield is a trademark of FerriShield, Inc. in the United States, other countries, or both.

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

Other company, product, and service names may be trademarks or service marks of others.
### List of Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>application program interface</td>
</tr>
<tr>
<td>ASCII</td>
<td>American National Standard Code for Information Exchange</td>
</tr>
<tr>
<td>CCA</td>
<td>Common Cryptographic Architecture</td>
</tr>
<tr>
<td>CLU</td>
<td>Coprocessor Load Utility</td>
</tr>
<tr>
<td>FIPS</td>
<td>Federal Information Processing Standard</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines</td>
</tr>
<tr>
<td>ICAT</td>
<td>Interactive Code Analysis Tool</td>
</tr>
<tr>
<td>I/O</td>
<td>input/output</td>
</tr>
<tr>
<td>IPL</td>
<td>initial program load</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>MD5</td>
<td>message digest 5 (hashing algorithm)</td>
</tr>
<tr>
<td>PCI</td>
<td>peripheral component interconnect</td>
</tr>
<tr>
<td>PCI-X</td>
<td>peripheral component interconnect express</td>
</tr>
<tr>
<td>PDF</td>
<td>portable document format</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest-Shamir-Adleman (algorithm)</td>
</tr>
<tr>
<td>TOD</td>
<td>time-of-day (clock)</td>
</tr>
<tr>
<td>UART</td>
<td>universal asynchronous receiver/transmitters</td>
</tr>
<tr>
<td>VPD</td>
<td>vital product data</td>
</tr>
<tr>
<td>xC</td>
<td>Linux Cryptographic Coprocessor</td>
</tr>
</tbody>
</table>
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 4764 PCI-X 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.

**ASCII.** American National Standard Code for Information Interchange.

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

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

**authorize.** To permit or give authority to a user to communicate with or make use of an object, resource, or function.

**B**

**battery-backed random access memory (BBRAM).** Random access memory that uses battery power to retain data while the system is powered off. The IBM 4764 PCI-X Cryptographic Coprocessor uses BBRAM to store persistent data for xC applications, as well as the coprocessor device key.

**BBRAM.** Battery-backed random access memory.

**bus.** In a processor, a physical facility along which

**C**

**call.** The action of bringing a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and jumping to an entry point. (I) (A)
card. (1) An electronic circuit board that is plugged into an expansion slot of a system unit. (2) A plug-in circuit assembly. (3) See also expansion card.

CBC. Cipher block chain.

CCA. Common Cryptographic Architecture.

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 4764 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 4764 PCI-X Cryptographic Coprocessor.

Cryptographic Coprocessor (IBM 4764). An expansion card that provides a comprehensive set of cryptographic functions to a workstation.

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

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

D

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

Data Encryption Standard Manager (DES_Mgr). A Linux extension that manages the IBM 4764 PCI-X 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 decrypt.

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 in a PC or RS/6000 machine into which a user can install an expansion card.

F

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


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.

**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 4764 PCI Cryptographic Coprocessor, the workstation into which the coprocessor is installed.

**ICAT.** Interactive Code Analysis Tool.

**Initial Program Load (IPL).** (1) The initialization procedure that causes an operating system to commence operation. (2) The process by which a configuration image is loaded into storage. (3) The process of loading system programs and preparing a system to run jobs.

**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 4764 PCI-X 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 4764 PCI-X 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.

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

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

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

**Miniboot.** Software within the IBM 4764 PCI-X Cryptographic Coprocessor designed to initialize the Linux 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.

**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.
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.

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

procedure call. In programming languages, a language construct for invoking execution of a procedure. (1) 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 4764 PCI-X Cryptographic Coprocessor PKA processing hardware.

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.

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.

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) (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.

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

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.

workstation. A terminal or microcomputer, usually one that is connected to a mainframe or a network, and from which a user can perform applications.

IBM 4764. IBM 4764 PCI-X Cryptographic Coprocessor.
Index

A
agent 1-1
arguments
  file description E-15
  image file E-16
  signature key E-15
  targeting E-17
  trust and countersignature E-16

B
building JFFS2 filesystem images 3-7, A-3
building XC applications with the developer's toolkit makefiles D-1

C
CCA roles for Signer and Packager E-25
CCA support program
code signing utility 3-9
CLU
  See Coprocessor Load Utility (CLU)
coding requirements, special
debugger attaching 3-3
developer identifiers 3-3
components
development 1-4
  release 1-5
Coprocessor Load Utility (CLU)
  commands B-1
  files used as input 2-4
  return codes B-2
  syntax B-1
  using B-1
coprocessor memory segments E-1
CRUPKG.EXE (Packager utility) E-22
CRUSIGNR.EXE (Signer utility) E-4
  operations E-6
CRUZPKG.EXE (zSeries Packager utility) E-23
CRUZSIGN.EXE (zSeries Signer utility) E-4, E-16

D
debugger, attaching 3-3
debugging 3-9, A-3
developer identifiers 3-3, 4-2
development environment
  road map 3-2
  special coding requirements 3-3
  toolkit components 1-4
development platform, preparing 2-6, A-1
development process, overview 1-3, A-1
device driver, installing 2-6
Device Reload Utility (DRUID)
  description 1-4
  syntax 3-9
directory structure 2-2
documentation, available 1-1
downloading and debugging 3-9, A-3

H
how to load software into the coprocessor 3-1

I
ICAT debugger, using 3-11
identifiers, developer 3-3, 4-2
installing the 4764 device driver 2-6
introduction 1-1

J
JFFS2 filesystem images, building 3-7, A-3

K
KEYGEN function 4-2, E-11

L
loading JFFS2 filesystem 3-9

M
makefiles, building XC applications with the toolkit D-1
makeup coprocessor memory segments and security, coprocessor E-1

O
overview of the development process 1-3, A-1

P
Packager utility
  CCA roles E-25
  CRUPKG.EXE E-22
  CRUZPKG.EXE E-23
  using E-1
packaging and releasing an XC application 4-1, A-3
platform, preparing development 2-6, A-1
prerequisite knowledge 1-v
prerequisites 1-2
programming serial port applications 3-6

R
rebooting the IBM 4764 C-1
related publications 1-vii
release components 1-5
releasing an XC application 4-1
return codes, CLU B-2
road map, development process 3-2
roles Signer and Packager, CCA E-25
RSA keypairs, generating 4-2

S
Signer utility
  CCA roles E-25
cryptographic functions E-6
  EMBURN2 E-7
  EMBURN3 E-8
  ESIG3 E-9
  ESTOWN3 E-10
  HASH_GEN E-11
  HASH_VER E-11
IBM-specific functions E-7
KEYGEN E-11
miniboot command functions E-6
miscellaneous functions E-6
REMBURN2 E-12
REMBURN3 E-13
signing an XC application E-4
SUROWN2 E-14
SUROWN3 E-14
syntax E-5
using E-1, E-4
state transitions for segment 2 E-3
syntax
  Coprocessor Load Utility (CLU) B-1
  Packager (CRUPKGR.EXE) E-22
  Packager (CRUZPKG.EXE) E-23
  Signer (CRUSIGNR.EXE) E-5

T
testing an XC application A-3
toolkit components 1-4

U
using Signer and Packager E-1
utilities
  Coprocessor Load Utility (CLU) B-1
  JFFS2 filesystem script 1-4
  Packager E-22, E-23
  Signer E-4

X
XC application 1-1

Z
zSeries Signer Utility (CRUZSIGN.EXE) E-4, E-5, E-16