IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer's Toolkit Guide

02-NOV-01, 10:07

Note!

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

Third Edition (January, 2001)

IBM does not stock publications at the address given below. This and other publications related to the IBM 4758 Coprocessor can be obtained in PDF format from the Library page at *http://www.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.ibm.com/security/cryptocards*, or you can respond by mail to:

Department VM9A, MG81/204-3
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.

Table of Contents

Figures

About This Book

The *IBM 4758 PCI 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 4758 PCI Cryptographic Coprocessor
- Load applications under development into a coprocessor
- Debug applications under development running within a coprocessor

The primary audience for this book are 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.

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 CP/Q++ operating system that runs within the coprocessor (as described in the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Overview*), and the use of the IBM's Common Cryptographic Architecture (CCA) application and support program (as described in the *IBM 4758 PCI Cryptographic Coprocessor CCA Support Program Installation Manual*) may also be helpful.

People who are interested in packaging, distribution, and security issues for custom software will 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 x 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 SCC 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 4758 PCI cryptographic coprocessor for use as a development platform.

Chapter 3, "Developing and Debugging an SCC Application" discusses in detail the use of each of the tools used during development of an SCC application.

Chapter 4, "Testing an SCC Application in a Production Environment" describes how to load production-level software into the coprocessor used as a development platform.

Chapter 5, "Packaging and Releasing an SCC Application" describes how to prepare an SCC 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 SCC application.

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

Appendix C, “How to Reboot the IBM 4758” 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 SCC Applications with Microsoft Developer Studio 97 or Microsoft Visual Studio 6.0” describes how to configure Microsoft Developer Studio 97** or Microsoft Visual Studio 6.0 to ensure the proper compiler and linker options are used to build an SCC application.

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

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

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

Typographic Conventions

This publication uses the following typographic conventions:

- Commands that you enter verbatim onto the command line are presented in **bold** type.
- Variable information 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, checkboxes, 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 non-GUI 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 under “General Interest,” “CCA Support Program Publications,” and “Custom Software Publications” are available in Adobe Acrobat** portable document format (PDF) at *http://www.ibm.com/security/cryptocards*.

General Interest

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

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

CCA Support Program Publications

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

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

PKCS #11 Support Program Publications

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

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

Custom Software Publications

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

- *IBM 4758 PCI Cryptographic Coprocessor Custom Software Installation Manual*
- *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference*
- *IBM 4758 PCI Cryptographic Coprocessor Interactive Code Analysis Tool (ICAT) User’s Guide*
- *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Overview*
- *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference*
- *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System C Runtime Library Reference*
- *IBM 4758 PCI Cryptographic Coprocessor CCA User Defined Extensions Programming Reference*
- *IBM ⓔserver zSeries CCA User Defined Extensions Reference and Guide*
- *AMCC S5933 PCI Controller Data Book*, available from Applied Micro Circuits Corporation, 6290 Sequence Drive, San Diego, CA 92121-4358. Phone

1-800-755-2622 or 1-619-450-9333. The manual is available online as an Adobe Acrobat** PDF file at *http://www.amcc.com/pdfs/pciprod.pdf*.

Cryptography Publications

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

- “Application Support Architecture for a High-Performance, Programmable Secure Coprocessor,” J. Dyer, R. Perez, S.W. Smith, and M. Lindemann, 22nd National Information Systems Security Conference, October 1999.
- “Validating a High-Performance, Programmable Secure Coprocessor,” S.W. Smith, R. Perez, S.H. Weingart, and V. Austel, 22nd National Information Systems Security Conference, October 1999.
- “Building a High-Performance, Programmable Secure Coprocessor,” S.W. Smith and S.H. Weingart, Research Report RC21102, IBM T.J. Watson Research Center, February 1998.
- “Using a High-Performance, Programmable Secure Coprocessor”, S.W. Smith, E.R. Palmer, and S.H. Weingart, in *FC98: Proceedings of the Second International Conference on Financial Cryptography*, Anguilla, February 1998. Springer-Verlag LNCS, 1998. ISBN 3-540-64951-4
- “Smart Cards in Hostile Environments,” H. Gobioff, S.W. Smith, J.D. Tygar, and B.S. Yee, *Proceedings of the Second USENIX Workshop on Electronic Commerce*, 1996.
- “Secure Coprocessing Research and Application Issues,” S.W. Smith, Los Alamos Unclassified Release LA-UR-96-2805, Los Alamos National Laboratory, August 1996.
- “Secure Coprocessing in Electronic Commerce Applications,” B.S. Yee and J.D. Tygar, in *Proceedings of the First USENIX Workshop on Electronic Commerce*, New York, July 1995.
- “Transaction Security Systems,” D.G. Abraham, G.M. Dolan, G.P. Double, and J.V. Stevens, in *IBM Systems Journal* Vol. 30 No. 2, 1991, G321-0103.
- “Trusting Trusted Hardware: Towards a Formal Model for Programmable Secure Coprocessors,” S.W. Smith and V. Austel, in *Proceedings of the Third USENIX Workshop on Electronic Commerce*,” Boston, August 1998.
- “Using Secure Coprocessors,” B.S. Yee (Ph.D. Thesis), Computer Science Technical Report CMU-CS-94-149, Carnegie-Mellon University, May 1994.
- “Cryptography: It’s Not Just for Electronic Mail Anymore,” J.D. Tygar and B.S. Yee, Computer Science Technical Report, CMU-CS-93-107, Carnegie Mellon University, 1993.
- “Dyad: A System for Using Physically Secure Coprocessors,” J.D. Tygar and B.S. Yee, Harvard-MIT Workshop on Protection of Intellectual Property, April 1993.
- “An Introduction to Citadel—A Secure Crypto Coprocessor for Workstations,” E.R. Palmer, Research Report RC18373, IBM T.J. Watson Research Center, 1992.
- “Introduction to the Citadel Architecture: Security in Physically Exposed Environments,” S.R. White, S.H. Weingart, W.C. Arnold, and E.R. Palmer, Research Report RC16672, IBM T.J. Watson Research Center, 1991.

- “An Evaluation System for the Physical Security of Computing Systems,” S.H. Weingart, S.R. White, W.C. Arnold, and G.P. Double, Sixth Computer Security Applications Conference, 1990.
- “ABYSS: A Trusted Architecture for Software Protection,” S.R. White and L. Comerford, IEEE Security and Privacy, Oakland 1987.
- “Physical Security for the microABYSS System,” S.H. Weingart, IEEE Security and Privacy, Oakland 1987.
- *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*, Bruce Schneier, John Wiley & Sons, Inc. ISBN 0-471-12845-7 or ISBN 0-471-11709-9
- *ANSI X9.31 Public Key Cryptography Using Reversible Algorithms for the Financial Services Industry*
- *IBM Systems Journal* Volume 30 Number 2, 1991, G321-0103
- *IBM Systems Journal* Volume 32 Number 3, 1993, G321-5521
- *IBM Journal of Research and Development* Volume 38 Number 2, 1994, G322-0191
- *USA Federal Information Processing Standard (FIPS):*
  - *Data Encryption Standard,* 46-1-1988
  - *Secure Hash Algorithm,* 180-1, May 31, 1994
  - *Cryptographic Module Security,* 140-1
- *Derived Test Requirements for FIPS PUB 140-1*, W. Havener, R. Medlock, L. Mitchell, and R. Walcott. MITRE Corporation, March 1995.
- *ISO 9796 Digital Signal Standard*
- *Internet Engineering Taskforce RFC 1321*, April 1992, MD5
- *Secure Electronic Transaction Protocol Version 1.0*, May 31, 1997

IBM Research Reports can be obtained from:

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

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

Other IBM Cryptographic Product Publications

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

- *IBM Transaction Security System General Information Manual*, GA34-2137
- *IBM Transaction Security System Basic CCA Cryptographic Services*, SA34-2362
- *IBM Transaction Security System I/O Programming Guide*, SA34-2363
- *IBM Transaction Security System Finance Industry CCA Cryptographic Programming*, SA34-2364

- *IBM Transaction Security System Workstation Cryptographic Support Installation and I/O Guide*, GC31-4509
- *IBM 4755 Cryptographic Adapter Installation Instructions*, GC31-4503
- *IBM Transaction Security System Physical Planning Manual*, GC31-4505
- *IBM Common Cryptographic Architecture Services/400 Installation and Operators Guide, Version 2*, SC41-0102
- *IBM Common Cryptographic Architecture Services/400 Installation and Operators Guide, Version 3*, SC41-0102
- *IBM 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 Cyptographic 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 4758 PCI Cryptographic Coprocessor Custom Software Developer's Toolkit Guide* contains product information that is current with the IBM 4758 PCI Cryptographic Coprocessor announcements made through May, 1998.

Changes made to this first edition in January, 1999 include:

- Chapter 3—Added a new switch to both the VisualAge C++ and Microsoft** Visual C++ compiler options.
- Appendix D—Added a required setting to the coprocessor-side portion of an SCC application under the Preprocessor Category.
- Appendix E—Changed the makefile name *cpqenv.mak* to *cpqenvtk.mak*.

Changes made to this first edition in February, 1999 include:

- Chapter 2—Added a description of the CRSrrrss.CLU file, which is now shipped as part of the Developer's Toolkit.
- Chapter 3—Added information on how to build the host-side portion of an SCC application and simplified the suggested code to incorporate in an application to prevent it from making too much progress before the debugger has a chance to attach. Also added a switch to the VisualAge C++ linker options.
- Appendix D—Removed *...\scctk\cpqenv\nt\msvcmasm* from the list of "Additional include directories" for the host-side portion of an SCC application, added instructions on how to build applications using Microsoft Visual Studio 6.0, and clarified the type of project required for the coprocessor-side portion of an SCC application.
- Appendix E—Added information on how to build the host-side portion of an SCC application.

Changes made to this first edition in March, 1999 include:

- Chapter 2—Added a description of several new CLU files that are now shipped as part of the Developer's Toolkit. Also changed the names of several files.
- Appendix E—Changed the names of the environment variables used with the Developer's Toolkit makefile.

Changes made to this second edition in July, 1999 include:

- Chapter 2—Added installation instructions for the toolkit when shipped on CD-ROM and clarified the instructions on preparing the development platform.
- Chapter 3—Revised the list of required options for the IBM VisualAge and Microsoft** Visual C++ compilers and the ILINK linker, added a note concerning the format of the name of the output file generated by CPQXLT, added a note concerning the behavior of the debugger if the debuggee is blocked in CPYield when the debugger attaches or if the directory containing the debugger does not appear early in the path.
- Chapter 5—Substantially revised the chapter to reflect current practices with respect to preparing applications for release.
- Appendix A—Substantially revised the appendix to reflect current practices with respect to preparing applications for release.
- Appendix D—Added a setting to the coprocessor-side portion of an SCC application under the General Category. Also added the Code Generation Category and a new setting.

Changes made to this second edition in October, 1999 include:

- Chapter 3—Changed the recommended system call to make inside an infinite loop to allow the debugger to attach before the application under debug makes too much progress.
- Function calls in the C runtime are not necessarily ANSI-compliant.

Changes made to this second edition in April, 2000 include:

- Chapter 2—Removed FMTRODSK.EXE and FMTTKCLU.EXE from the list of items included in the *scctk\nt\bin* directory.

Changes made to this third edition in January, 2001 include:

- Chapter 2 and Appendix A—Added a warning that loading TDVrrrss.CLU for model 002 or 023 into a model 001 or 013 can leave the coprocessor in an unusable state. Also added REMBURN2.R2T to the list of files generated by CRUSIGNR.
- Chapter 2—Updated the toolkit directory structure.
- Throughout the manual—Updated file extensions for files generated by CRUSIGNR and CRUPKGR.

Changes made to this third edition in October, 2001 include:

- Throughout the manual - Added information about how to load extensions to the IBM CCA application into a PCI Cryptographic Coprocessor that is installed in an IBM @server zSeries server.
- Appendix C - Updated information about stopping and starting the device driver.

Chapter 1. Introduction

The Developer's Toolkit is a set of libraries, include files, and utility programs that help a developer build, load, and debug applications written in C or assembler for the IBM 4758 PCI Cryptographic Coprocessor. An application that runs within the coprocessor is known as an "agent" or an "SCC application".[1]

The Developer's Toolkit, a commercial compiler and linker, and a PC running Windows NT constitute a complete development environment for the IBM 4758. IBM's CCA Support Program feature is required in order to create a version of an application suitable for distribution. This chapter includes:

- A description of the documentation available to a developer of an SCC 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 SCC 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 SCC application for release

Available Documentation

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

1. *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Overview*, which describes the features of the coprocessor operating system.
2. *IBM 4758 PCI Cryptographic Coprocessor General Information Manual* which provides a basic understanding of IBM's Common Cryptographic Architecture for the IBM 4758.[2]
3. 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 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference*, which describes the function calls supplied by the coprocessor operating system.
- The *IBM 4758 PCI 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.
- The *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System C Runtime Library Reference*, which describes the function calls supported by the full C run-time library supplied by the general version of CP/Q. The customized version of CP/Q that runs in an IBM 4758 does not support the full C run-time library. See "C Run-Time Library Support" on page 3-4 for details.

[1] Secure cryptographic coprocessor (SCC) is an alternate name for the IBM 4758 PCI Cryptographic Coprocessor.

[2] This document will be of particular interest to developers writing user-developed extensions for CCA.

- Developers writing extensions for IBM's CCA application will also need the *IBM 4758 PCI Cryptographic Coprocessor CCA User Defined Extensions Programming Reference*.
- Developers writing CCA extensions that will be run on a PCI Cryptographic Coprocessor installed in an IBM zSeries server will need the *IBM eserver zSeries CCA User Defined Extensions Reference and Guide*.
- *IBM 4758 PCI Cryptographic Coprocessor CCA Support Program Installation Manual* which describes how to configure IBM's CCA application, which in turn is used by some of the tools in the Developer's Toolkit.

Prerequisites

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

1. An IBM 4758 PCI cryptographic coprocessor.[3] Refer to *http://www.ibm.com/security/cryptocards* for ordering information.

   The IBM 4758 should be installed in a host following the instructions in the *IBM 4758 PCI Cryptographic Coprocessor Installation Manual*, which also lists the hardware and software requirements for the host. For application development, the host must be a PC running Windows NT.

2. One of the supported compilers (IBM VisualAge C++[4] or Microsoft Visual C++) 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.

3. The IBM 4758 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.

   The Developer's Toolkit includes a device driver for the IBM 4758 which should be installed on the host following the instructions in chapter 2 of the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Installation Manual*.

4. Developers writing extensions to IBM's CCA application will also need the IBM 4758 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.

   Developers writing CCA extensions that will be run on a PCI Cryptographic Coprocessor installed in an IBM zSeries server will need additional, zSeries-specific enhancements to the UDX toolkit.

Note: Refer to the Custom Programming page of the IBM 4758 product Web site at *http://www.ibm.com/security/cryptocards* for more information about the toolkits. To contact IBM concerning availability of either toolkit, submit a request using the Comments and Questions form located on either the Custom Programming page or the Support page of the product Web site.

[3] Development of extensions to the IBM CCA application that will be run on an IBM zSeries server requires an IBM 4758 Model 002/023 PCI Cryptographic Coprocessor.

[4] IBM offers VisualAge C++ in several packages. Part number 33H4979 is version 3.5 on CD; part number 33H4980 includes documentation. Upgrades are also available.

The developer must obtain and install the following to prepare an application for release:

1. IBM's CCA Support Program for Windows NT (feature 4376) and a function control vector permitted by the applicable import or export regulations (feature 5200, 5201, or 5202). Information on ordering these items can be obtained from *http://www.ibm.com/security/cryptocards*.

   The CCA Support Program should be installed following the instructions in chapter 3 of the *IBM 4758 PCI Cryptographic Coprocessor CCA Support Program Installation Manual*.

Development Overview

As illustrated in Figure 1-1 on page 1-4, an SCC application is compiled and linked in the same manner as a host application, using include and library files customized for the coprocessor environment. The executable is then translated to the format understood by CP/Q++ and is downloaded to the coprocessor.

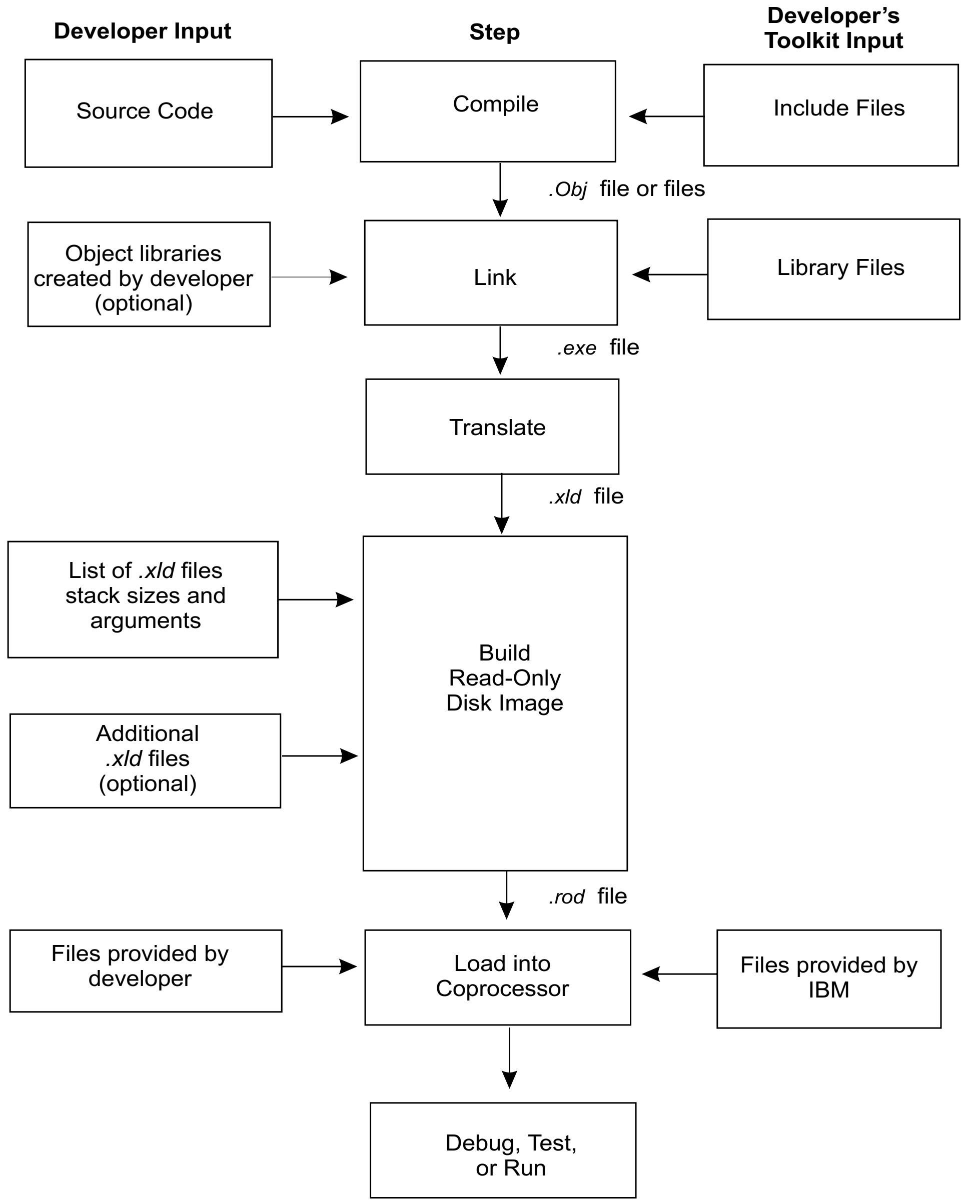

Developer Input
Step
Developer's Toolkit Input
Source Code
Compile
Include Files
.Obj file or files
Object libraries created by developer (optional)
Link
Library Files
.exe file
Translate
.xld file
List of .xld files stack sizes and arguments
Build Read-Only Disk Image
Additional .xld files (optional)
.rod file
Files provided by developer
Load into Coprocessor
Files provided by IBM
Debug, Test, or Run

Figure 1-1. Development Process Overview

The following steps are required to build and load SCC applications:

1. Compile the program.
2. Link the program.
3. Translate the executable file into the format required by CP/Q^{++}.
4. Build a read-only disk image.
5. Load the disk image into the coprocessor.

The Developer's Toolkit includes the tools needed to perform steps 3 through 5.

Development Environment Components

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

Compiler and Linker

Use either IBM VisualAge* C^{++}, Microsoft Visual C^{++}**, or—for assembler language code—Microsoft Assembler**. The linker must be compatible with the compiler; for example, use ILINK with VisualAge C^{++}. These are not shipped with the Developer's Toolkit.[5] Part numbers for IBM VisualAge C++ appear in "Prerequisites" on page 1-2.

Libraries and Include Files

Use the Developer's Toolkit libraries (*.lib*) and include files (*.h/.inc*) in place of the libraries and include files shipped with the compiler and assembler. These files furnish the library entry points and routines that SCC applications use to interface with the CP/Q operating system and the CP/Q++ extensions.

Utilities

Use the following utilities to prepare and load SCC applications:

- **Translator:** A program (CPQXLT.EXE) that translates executable (*.exe*) files into the format (*.xld*) required by CP/Q.
- **Disk Builder:** A program (SCCRODSK.EXE) that packages one or more applications into a read-only disk image.
- **Development Reload Utility (DRUID):** A program (DRUID.EXE[6]) that loads an application into a coprocessor configured as a development platform.
- **Coprocessor Load Utility (CLU):** A program (CSUNCLU.EXE) that verifies and loads digitally signed system software and coprocessor commands into a coprocessor.

CLU Input Files

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

Debugger

The IBM Interactive Code Analysis Tool (ICAT) debugger (ICATCPW.EXE) is a Windows NT program that controls and debugs SCC applications.

[5] Developers writing extensions for IBM's CCA application must use IBM VisualAge C^{++}.

[6] Development Reload Utility for Insecure Development (DRUID)

Coprocessor Operating System

The Developer's Toolkit includes two versions of the CP/Q^{++} embedded operating system:

- A **debug** version (TPRrrrss.CLU) used when coding and debugging an application. It contains a debug probe that runs within the coprocessor and services requests from the ICAT debugger.
- A **production** version (TNPrrrss.CLU) used to test an application in a production level environment. It does not include the debug probe.

Both versions are supplied as signed disk images that can be loaded into the coprocessor by CLU. They include the CP/Q^{++} extensions needed to manage the coprocessor hardware, and can include custom extensions specified by the contract between the developer and IBM.

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 SCC application for release:

- **Signer:** A program (CRUSIGNR.EXE) that generates RSA keypairs and performs other cryptographic operations and that incorporates a read-only disk image into a coprocessor command and digitally signs the command using a developer's private key.
- **Packager:** A program (CRUPKGR.EXE) that combines one or more signed commands into a single file for download to the coprocessor.

CCA Application and Support Program

Signer and Packager use IBM's Common Cryptographic Architecture (CCA) application to generate digital signatures. The CCA application and support program are not shipped with the Developer's Toolkit. See “Prerequisites” on page 1-2 for more information.

Chapter 2. Installation and Setup

The Developer’s Toolkit includes utilities used to build an SCC application and prepare it to be loaded into an IBM 4758 PCI cryptographic coprocessor. This chapter describes how to install the Developer’s Toolkit and the UDX Toolkit (if used), discusses the toolkit’s directory structure and 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 using the xcopy /s/e/r command.

Directories and Files

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

The *DOC* subdirectory contains documentation on use of the toolkit, application programming interfaces (APIs), and other information.

The *SCCTK* subdirectory includes the tools required to create applications for the IBM 4758 models 001 and 013 (in *SCCTK\001*) and the tools required to create applications for the IBM 4758 models 002 and 023 (in *SCCTK\002*). The two subtrees are almost identical and the remainder of this chapter omits the model subdirectory when referring to files and their locations (for example, the chapter refers to the "*scctk\bin\nt*" directory rather than the "*scctk\001\bin\nt*" or "*scctk\002\bin\nt*" directory).

The *UDXTK* subdirectory includes the tools required to create User Defined Extensions to IBM’s Common Cryptographic Architecture (CCA) application. See *IBM 4758 PCI Cryptographic Coprocessor CCA User Defined Extensions Reference and Guide* or *IBM* @server *zSeries CCA User Defined Extensions Reference and Guide* for details.

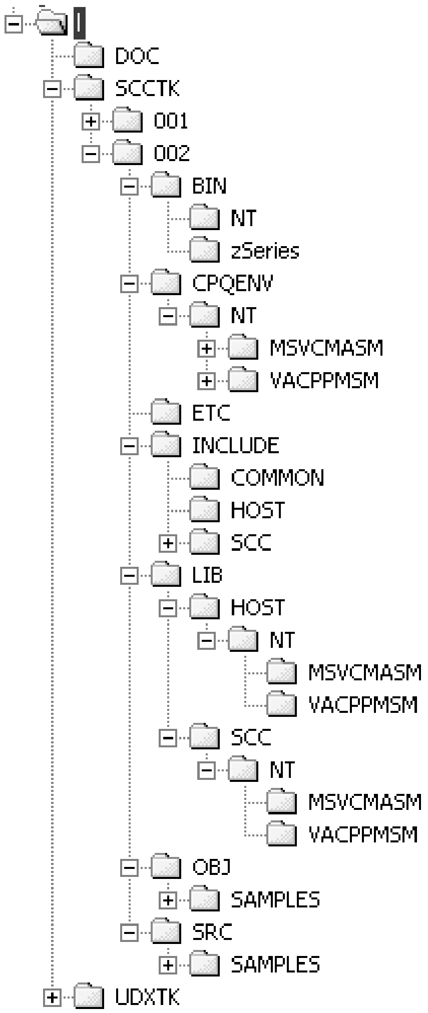


Figure 2-1. Toolkit Directory Structure

- The *scctk\bin\nt* directory contains the following:
  - An InstallShield package containing the device driver and associated DLLs (4758DD.EXE)
  - The translator utility (CPQXLT.EXE)
  - The read-only disk image builder (SCCRODSK.EXE)
  - The device reload utility (DRUID.EXE)
  - The coprocessor load utility (CSUNCLU.EXE)
  - The debugger (ICATCPW.EXE)
  - The signer utility (CRUSIGNR.EXE/CRUZSIGN.EXE)
  - The packager utility (CRUPKGR.EXE/CRUZPKG.EXE)
  - DLLs and command files used by the tools in the directory

  The InstallShield package must be run to place the device driver and other files it contains in the appropriate directories and to make certain entries in the Windows registry. See the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Installation Manual* for more details.

  The InstallShield process places copies of many of the files it contains on a directory specified by the user and adds this directory to the path. If the directory is *not scctk\bin\nt*, the *scctk\bin\nt* directory should be added to the PATH environment variable.

- The *scctk\cpqenv\nt* directories contain include files (*.h* and *.inc*) that define macro variables to customize the include files in the *scctk\include\scc* directory for use with Microsoft Visual C++ (MSVC++) and Microsoft Assembler (MASM) (the *scctk\cpqenv\nt\msvcmasm* directory) or with IBM VisualAge C++ (VACPP) and MASM (the *scctk\cpqenv\nt\vacppmsm* directory).
- The *scctk\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 3-13), including those listed as follows. Many files in this directory have names of the form Txxrrrss, where *rrr* indicates the release of the CCA application the file contains or with which the file is associated, *ss* indicates the revision level of the CCA application, and *xx* distinguishes the file from all others.
  - CR1rrrss.CLU[1], which loads release rrr revision ss of IBM's system software into a coprocessor.[2]

    CR1rrrss.CLU can only be loaded into an IBM 4758 in the factory-fresh state.[3]
  - CE1rrrss.CLU, which updates the system software in a coprocessor.

    CE1rrrss.CLU loads release rrr revision ss of IBM's system software into an IBM 4758 into which system software has previously been loaded.

    Warning

    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 CE1rrrss.CLU need be loaded only once.
  - TDVrrrss.CLU, which prepares a coprocessor for use as a development platform.[4]

    TDVrrrss.CLU can only be loaded into an IBM 4758 that contains release rrr revision ss of IBM's system software.[5]
  - TPRrrrss.CLU and TPSrrrss.CLU, which load into a coprocessor a copy of the operating system (CP/Q++) that allows a coprocessor application to be debugged.

    TPRrrrss.CLU and TPSrrrss.CLU can only be loaded into an IBM 4758 that has been prepared for use as a development platform using TDVrrrss.CLU.[6]

[1] Prior to release 131, CR1rrrss.CLU was named CFCrrrss.CLU.

[2] CR1rrrss.CLU and CE1rrrss.CLU also set the public key associated with segment 1.

[3] In particular, the public key associated with segment 1 must be the key installed during manufacture.

[4] TDVrrrss.CLU sets the public key and owner identifier associated with segment 2. Currently, the owner identifier assigned to segment 2 is 3.

Prior to release 131, TDVrrrss.CLU also incorporated the functions provided by TE3rrrss.CLU and TL3rrrss.CLU.

[5] 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.

[6] In particular, the public key and owner identifier associated with segment 2 and the image names associated with segments 1 and 2 must have the values CR1rrrss.CLU or CE1rrrss.CLU and TDVrrrss.CLU assign them.

TPRrrrss.CLU and TPSrrrss.CLU both incorporate a “debug probe” that allows the ICAT debugger to manipulate a coprocessor application. The copy of the debug probe in TPRrrrss.CLU can communicate with the ICAT debugger using the PCI bus or via the serial port built into the IBM 4758. The copy of the debug probe in TPSrrrss.CLU can only communicate using the PCI bus, thereby placing the serial port at the application’s disposal.

– TNPrrrss.CLU, which replaces the “debug-enabled” version of CP/Q++ with a production-level copy of CP/Q++ release rrr revision ss. This allows a developer to test a coprocessor application in a production-level environment.

  TNPrrrss.CLU can only be loaded into an IBM 4758 that has been prepared for use as a development platform using TDVrrrss.CLU.[6]

– TE3rrrss.CLU, which enables a coprocessor to accept coprocessor applications downloaded by the DRUID utility.[7]

  TE3rrrss.CLU can only be loaded into an IBM 4758 that has been prepared for use as a development platform using TDVrrrss.CLU.[6]

  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 4758 that has been prepared for use as a development platform using TDVrrrss.CLU and which has been prepared to accept downloaded applications using TE3rrrss.CLU.[8]

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

  TR3rrrss.CLU can only be loaded into an IBM 4758 that has been prepared for use as a development platform using TDVrrrss.CLU and which has been prepared to accept downloaded applications using TE3rrrss.CLU.[8]

– S3KCLRPP.DRK and S3KCLRPU.DRK, which contain an RSA keypair and the public key of the keypair, respectively. These files are provided as input to DRUID.

– 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 IBM 4758 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 CR1rrrss.CLU or CE1rrrss.CLU has been loaded.

[7] TE3rrrss.CLU sets the owner identifier associated with segment 3. Currently, the owner identifier assigned to segment 3 is 6.

[8] 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.

TRSrrrss.CLU can only be loaded into an IBM 4758 that has been prepared for use as a development platform using TDVrrrss.CLU.[9]

- CRSrrrss.CLU, which prepares an IBM 4758 into which IBM's CCA program has been loaded for use as a development platform. CRSrrrss.CLU essentially restores the coprocessor to the state it is in immediately after CR1rrrss.CLU or CE1rrrss.CLU has been loaded.

  CRSrrrss.CLU can only be loaded into an IBM 4758 into which IBM's CCA program has been loaded.

- CFFrrrss.CLU[10], which restores a coprocessor that has been used for development to the factory-fresh state.

  Warning

  CFFrrrss.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 restore a coprocessor to its factory-fresh state only if absolutely necessary. Note that once the public key associated with segment 1 has been set using CR1rrrss.CLU or by loading IBM's CCA application, it should not be necessary to restore the coprocessor to its factory-fresh state.

- ESTOWN2.E2T, EMBURN2.L2T, REMBURN2.R2T, and SUROWN2.S2T, which are used to generate a version of an application suitable for release. See Chapter 5, "Packaging and Releasing an SCC Application" on page 5-1 for details.

- The *scctk\include* directories contain include files (*.h* and *.inc*) that replace the standard include files that ship with MSVC++, VACPP, and MASM.

  - *scctk\include\common* contains include files that are used to build both SCC applications and host applications that interact with SCC applications.
  - *scctk\include\host* contains include files that are used only to build host applications.
  - *scctk\include\scc* contains include files that are used only to build SCC applications.

- The *scctk\lib* directories contain library (*.lib*) files that augment or replace the standard library files that ship with MSVC++ or VACPP. The *scctk\lib\host* directories contain library files that are used to build host applications and the *scctk\lib\scc* directories contain library files that are used to build SCC applications. The *msvcmasm* subdirectories are used when building applications with MSVC++ and the *vacppmsm* subdirectories are used when building applications with VACPP.

- The *scctk\obj* directories are empty. The makefiles in the *scctk\src\* directories place object and executable files in this subtree.

- The *scctk\src\samples* directories contain the source for some sample host and SCC applications.

[9] In particular, the public key and owner identifier associated with segment 2 must have the values TDVrrrss.CLU assigns them.

[10] Prior to release 131, CFFrrrss.CLU was named C2Frrrss.CLU.

- The *scctk\src\udx* directory contains files used to create extensions to IBM's CCA application. This directory is not created unless the UDX Toolkit is installed.

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 *scctk\bin\nt*), and that the various system files (*cryptont.sys*, and so on) have been installed.

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

CSUNCLU \logfile-directory\CLU.LOG ST

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

```
*** ROM Status; INIT: INITIALIZED
*** ROM Status; SEG2: RUNNABLE , OWNER2: 03
*** ROM Status; SEG3: RUNNABLE , OWNER3: 06
*** Page 1 Certified: YES
*** Segment 1 Image: CCA 1.3.1 SEGMENT-1 ...
*** Segment 1 Revision: 131
*** Segment 2 Image: CP/Q++ 1.31 ...
*** Segment 2 Revision: 131
*** Segment 3 Image: ...
*** Segment 3 Revision: 1
```

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:

  CSUNCLU \logfile-directory\CLU.LOG PL \scctk\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:

  CSUNCLU \logfile-directory\CLU.LOG PL \scctk\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" on page 2-7.

- **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:

  CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\TDVrrrss.CLU

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

  Warning

  Attempting to load a copy of TDVrrrss.CLU for a model 002 or 023 coprocessor into a model 001 or 013 coprocessor can leave the coprocessor in an unusable state. Assistance from IBM is required to correct this condition.

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

  CSUNCLU \logfile-directory\CLU.LOG PL \scctk\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:

  CSUNCLU \logfile-directory\CLU.LOG PL \scctk\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 4758" on page C-1 for details) and then running the host reverse-then-echo driver, 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.

  The developer then proceeds to load TPRrrrss.CLU or TPSrrrss.CLU as indicated in "Segment 2 Owner ID = 3 and Segment 3 Owner ID = 6".

Segments 2 and 3 RUNNABLE

If the “ROM Status” lines indicate segments 2 and 3 are RUNNABLE, the owner identifiers specified on those lines for segments 2 and 3 dictate how to proceed:

- **Segment 2 Owner ID = 2: Unload Segment 2** - If the owner identifier associated with segment 2 is 2, the developer loads CRSrrrss.CLU into the coprocessor, for example:

  CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\CRSrrrss.CLU

  CRSrrrss relinquishes ownership of segment 2.

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

  If this 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.[11]

- **Segment 2 Owner ID = 3 and Segment 3 Owner ID = 6: Load Debug Kernel into Segment 2** - If the owner identifier associated with segment 2 is 3 and the owner identifier associated with segment 3 is 6, the developer loads TPRrrrss.CLU or TPSrrrss.CLU into the coprocessor, for example:

  CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\TPRrrrss.CLU
  or
  CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\TPSrrrss.CLU

  If the developer needs to run the debugger on a different machine than the one in which the coprocessor is installed (for example, the coprocessor is installed in an RS/6000 or another host that does not run Windows NT), the developer should load TPRrrrss.CLU because the debug kernel in TPRrrrss.CLU can communicate with the debugger via the coprocessor’s serial port.

  If the developer’s application uses the coprocessor’s serial port, the developer should load TPSrrrss.CLU. The debug kernel in TPSrrrss.CLU does not use the serial port, leaving it free for use by the application.

  If neither of the aforementioned considerations applies, the developer may load either TPRrrrss.CLU or TPSrrrss.CLU.

  This completes preparation of the coprocessor for use as a development platform.

- **Other Owner IDs -** If the owner identifiers associated with segment 2 and/or 3 differ from those previously listed, 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.

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.

[11] If CRS12200.CLU is used to relinquish ownership of segment 2, the developer must load CR1rrrss.CLU as indicated in “Segments 2 and 3 UNOWNED.” The contents of segment 1 cannot be used as a guide in this case.

If the “ROM Status” lines indicate segment 2 is RELIABLE_BUT_UNRUNNABLE, 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 2, the developer loads CRSrrrss.CLU into the coprocessor, for example:

```
CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\CRSrrrss.CLU
```

If the segment 2 onwer ID is 3, the developer loads TRSrrrss.CLU into the coprocessor, for example:

```
CSUNCLU \logfile-directory\CLU.LOG PL \scctk\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” above.[11]

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 neither 2 nor 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 2, the developer loads CRSrrrss.CLU into the coprocessor, for example:

```
CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\CRSrrrss.CLU
```

CRSrrrss.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.[12]

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:

```
CSUNCLU \logfile-directory\CLU.LOG PL \scctk\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.[12]

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:

```
CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\TL3rrrss.CLU
```

[12] If CRS12200.CLU is used to relinquish ownership of segment 2, the developer must load CR1rrrss.CLU as indicated in “Segments 2 and 3 UNOWNED.” The contents of segment 1 cannot be used as a guide in this case.

TL3rrrss.CLU sets the public key associated with segment 3 and loads the "reverse-then-echo" application into segment 3. 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 4758" on page C-1 for details) and then running the host reverse-then-echo driver, 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.

Figure 2-2 illustrates the steps involved in preparing an IBM 4758 for use as a development platform.

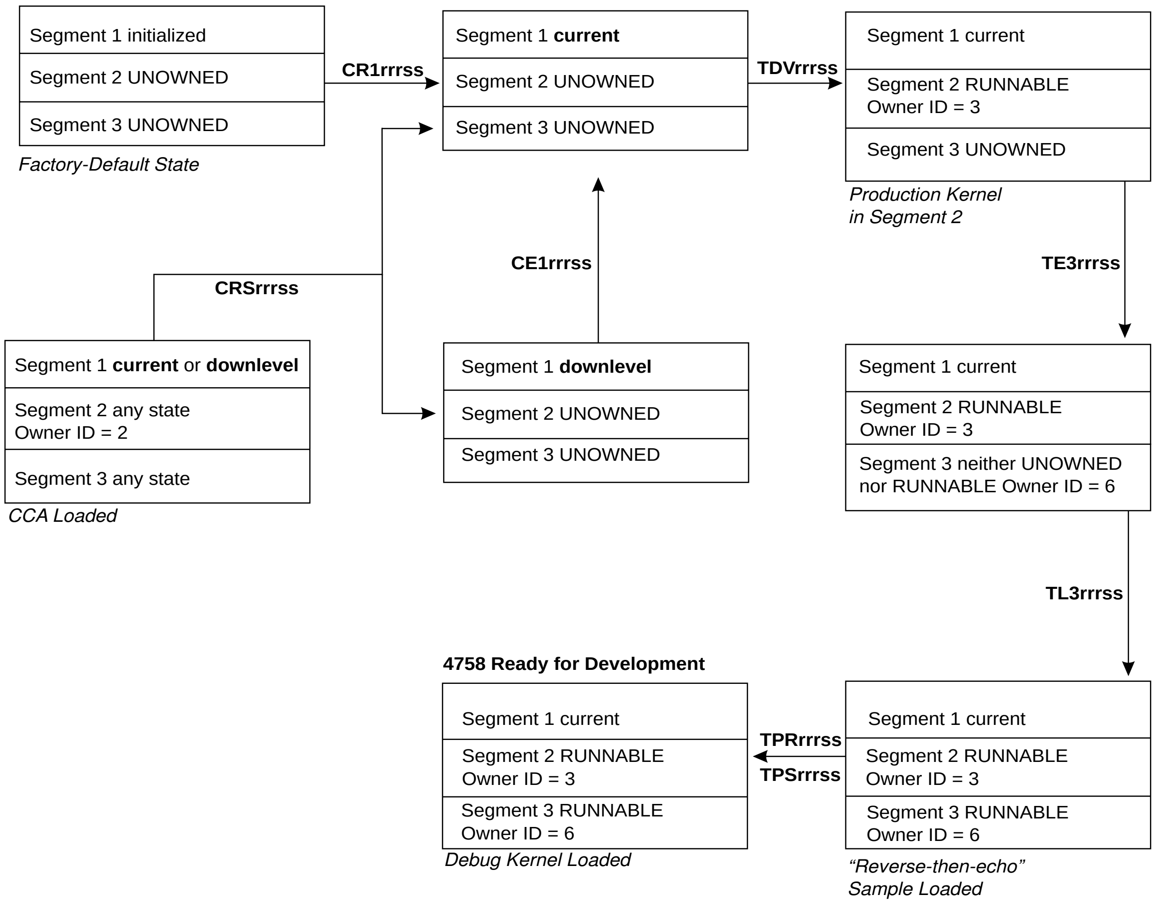

Segment 1 initialized
Segment 2 UNOWNED
Segment 3 UNOWNED
Factory-Default State
CR1rrrss
Segment 1 **current**
Segment 2 UNOWNED
Segment 3 UNOWNED
TDVrrrss
Segment 1 current
Segment 2 RUNNABLE Owner ID = 3
Segment 3 UNOWNED
Production Kernel in Segment 2
TE3rrrss
CE1rrrss
CRSrrrss
Segment 1 **current** or **downlevel**
Segment 2 any state Owner ID = 2
Segment 3 any state
CCA Loaded
Segment 1 **downlevel**
Segment 2 UNOWNED
Segment 3 UNOWNED
Segment 1 current
Segment 2 RUNNABLE Owner ID = 3
Segment 3 neither UNOWNED nor RUNNABLE Owner ID = 6
TL3rrrss
4758 Ready for Development
Segment 1 current
Segment 2 RUNNABLE Owner ID = 3
Segment 3 RUNNABLE Owner ID = 6
Debug Kernel Loaded
TPRrrrss
TPSrrrss
Segment 1 current
Segment 2 RUNNABLE Owner ID = 3
Segment 3 RUNNABLE Owner ID = 6
“Reverse-then-echo” Sample Loaded

Figure 2-2. Development Preparation Process

Chapter 3. Developing and Debugging an SCC Application

This chapter describes how to use the Developer's Toolkit to create the coprocessor-side portion of an SCC application and load it into an IBM 4758 Cryptographic Coprocessor. (The host-side portion of an SCC application may be built in the same manner as any other NT application. The only requirement is to use the appropriate compiler options to ensure the directories listed in "Include File Directory Search Order" on page 3-9 are searched in the proper order.)

This chapter describes:

- Each step in the development process
- Special coding requirements for development
- Unsupported CP/Q base operating system function calls
- Supported and unsupported C run-time library function calls and global variables
- Required option and switch settings for the compiler, assembler, linker, and librarian
- How to convert a compiled SCC application into a version that CP/Q++ can load and execute
- How to build a read-only disk image containing the SCC application
- How to load the disk image 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 *scctk\bin\nt*).

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. MSVC++ supplies *vcvars32.bat* and VACPP supplies *setenv.bat* for this purpose. And the environment variables SCCTK_FS_ROOT and SCCTK_ENV must be set to use the Developer's Toolkit makefiles. See Appendix E, "Building SCC Applications with the Developer's Toolkit Makefiles" on page E-1 for details.

Development Process Road Map

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

1. Compile, assemble, and link
2. Translate
3. Build disk image
4. Load image into the coprocessor

Figure 3-1 on page 3-2 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 1-4; this flowchart simply provides more detail.

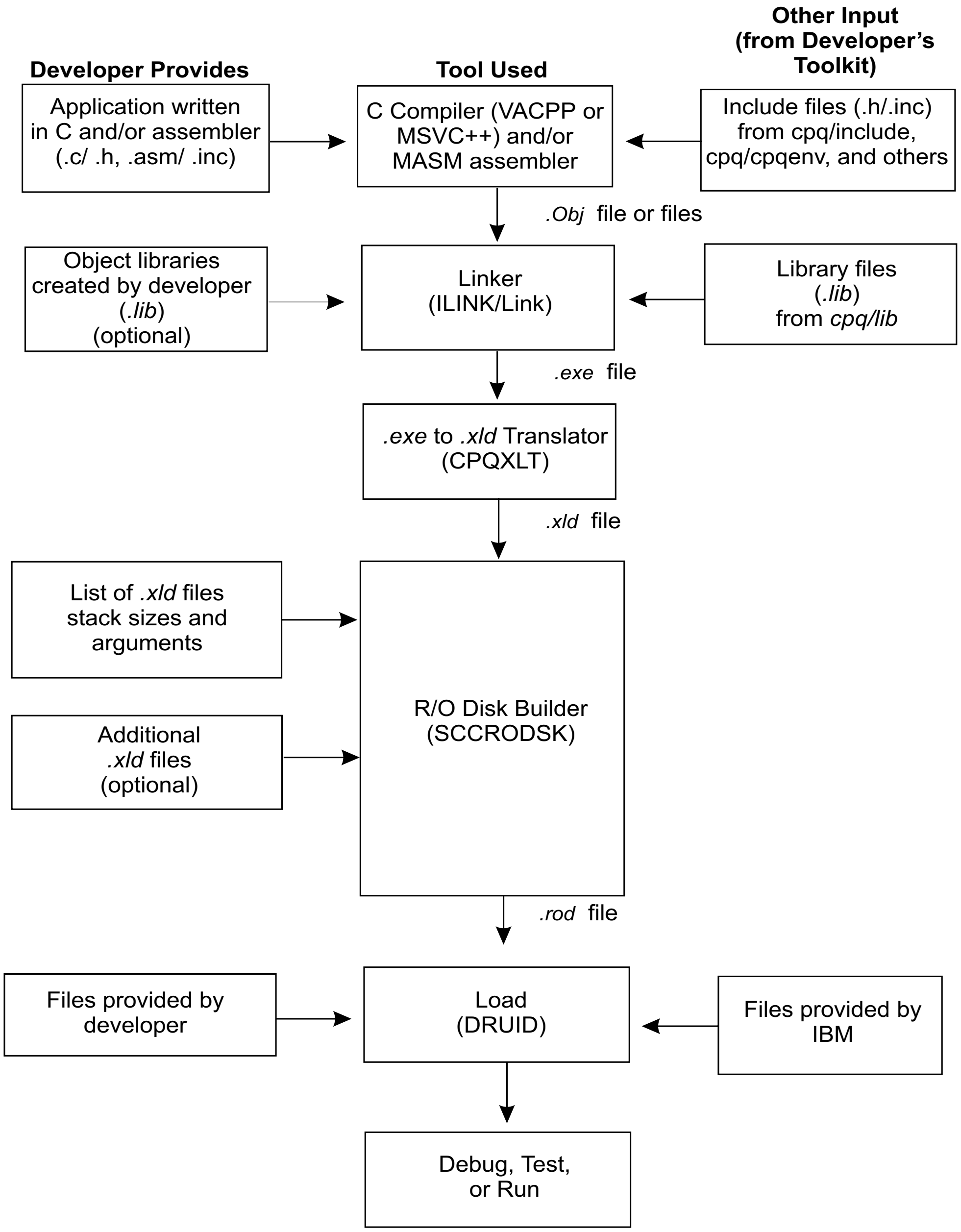

Developer Provides
Tool Used
Other Input (from Developer's Toolkit)
Application written in C and/or assembler (.c/ .h, .asm/ .inc)
C Compiler (VACPP or MSVC++) and/or MASM assembler
Include files (.h/.inc) from cpq/include, cpq/cpqenv, and others
.Obj file or files
Object libraries created by developer (.lib) (optional)
Linker (ILINK/Link)
Library files (.lib) from cpq/lib
.exe file
.exe to .xld Translator (CPQXLT)
.xld file
List of .xld files stack sizes and arguments
Additional .xld files (optional)
R/O Disk Builder (SCCRODSK)
.rod file
Files provided by developer
Load (DRUID)
Files provided by IBM
Debug, Test, or Run

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 SCC application must register with a CP/Q++ 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.[1] During development, a developer may use an arbitrary nonzero value 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 will be loaded and start to run as soon as the coprocessor is rebooted and may run for some time before the debugger places the application under debug and quiesces it. 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:

```
unsigned long i,j;

i=j=0;
for (;;)
{
  CPSleep(1,1);
  i++;
  if (j == 28) /* Make sure optimizer doesn't remove all cope 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, use the debugger’s **Jump to location** function to move the execution point to the statement immediately following the loop, or change the value of j to 28.

Compiling, Assembling, and Linking

The commercial compiler used in development of an SCC application is designed to create applications to run on a workstation under Windows NT rather than on a cryptographic coprocessor under CP/Q++. Consequently, the include files and libraries shipped with the compiler and the defaults for several options are not always appropriate for SCC applications.

This section lists the base operating system and C run-time library function calls that an SCC application may use and those that are not supported.[2] It also lists

[1] Refer to the description of sccSignOn in *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for details.

[2] Most unsupported functions are declared in an include file but are not implemented. That is, a program that invokes an unsupported function may compile, but it will not link.

options that must be specified when compiling, assembling, or linking to ensure that an SCC 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 E, "Building SCC Applications with the Developer's Toolkit Makefiles" on page E-1 for details on their use.

CP/Q Base Operating System Function Support

The Developer's Toolkit supports most of the functions described in *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference*. Exceptions are noted in Table 3-1. Furthermore, an SCC application has no special privileges and consequently cannot invoke certain restricted functions (for example, CPPTrace).

Table 3-1. Unsupported CP/Q Functions

CPSetPreempt	CPSigReturn	CPSigVec
CPSigInt	CPSigSend	
CPSigMask	CPSigStack	

C Run-Time Library Support

The Developer's Toolkit supports most of the library functions and global variables described in the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System C Runtime Library Reference*.[3] However, several functions have been modified and others are not supported at all. Furthermore, there are restrictions on the use of certain intrinsic functions. See "Compiler Options" on page 3-6 for details.

Supported Functions and Global Variables

Table 3-2 on page 3-5 lists the C run-time functions and global variables the Developer's Toolkit supports. Numbers after names refer to the notes following the table.

Note: The functions listed in Table 3-2 on page 3-5 may not be ANSI compliant. See the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System C Runtime Library Reference* for details.

[3] The C run-time library shipped with the Developer's Toolkit includes the cinitnon and exitnon versions of the initialization and exit code.

Table 3-2. Supported C Run-time Functions and Global Variables

_cdapage
_exit (7)
_fullname
_isalnum
_isalpha
_isascii
_iscntrl
_isdigit
_isgraph
_islower
_isprint
_ispunct
_isspace
_isupper
_isxdigit
_mons
_tolower
_toupper
_wdays
abort (7)
abs
asctime (7)
atexit (7)
atof
atoi
atol
bcmp
bcopy
binsort (1)
bsearch (1)
bzero
calloc
cjlvsn
clock
copyenv (7)
ctime (7)
difftime
div
errno (7)
exit (7)
ffs
free
ftime
getenv
getenvall
getenvall2
getopt (7)
gmtime (2) (7)
hsort (1)
index
inssort (1)
isalnum
isalpha
isascii
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
issupv
isupper
isxdigit
itoa
labs
ldiv
localtime (3) (7)
longjmp
malloc
memccpy
memchr
memcmp
memcpy
memicmp
memmove
memset
mktime
msort (1)
printf (4) (7)
putenv
qsort (1) (7)
rand (5) (7)
realloc
rindex
setjmp
sprintf
srand (5) (7)
sscanf (7)
strcat
strchr
strcmp
strcpy
strcspn
strdup
strerror
strftime (7)
stricmp
strlen
strlwr
strncat
strncmp
strncpy
strnicmp
strpbrk
strrchr
strrev
strspn
strstr
strtok (7)
strtol
strtoul
strupr
swab
time (6)
tolower
toupper
va_arg
va_end
va_start
vsprintf

Notes

1. This function takes as an argument the address of a comparison function. The comparison function must use the __cdecl linkage convention.
2. gmtime always sets the tm_isdst field of the output structure to 0.
3. localtime assumes local standard time is 300 minutes behind (west) of Greenwich Mean Time and makes an appropriate adjustment for daylight savings time.
4. The debug version of CP/Q++ forwards output generated by printf to the debugger, which displays the output in the Messages Window. The production version of CP/Q++ treats printf as a (relatively expensive) NOP.
5. rand and srand are ANSI-standard pseudo-random functions. Developers writing SCC applications should use sccGetRandomNumber instead.
6. time assumes the coprocessor time-of-day clock is set to local time (i.e., local standard time with an appropriate adjustment for daylight savings time) and that local standard time is 300 minutes behind (west) of Greenwich Mean Time.
7. This function or global variable is not thread-safe.

Unsupported Functions and Global Variables

Table 3-3 on page 3-6 lists the C run-time functions and global variables the Developer's Toolkit does not support. Numbers after names refer to the notes following the table.

Table 3-3. Unsupported C Functions and Global Variables

_iob
assert
brkpt
close
clearerr (1)
feof (1)
ferror (1)
fileno (1)
fclose
fflush
fgetc
fgetpos
fgets
fopen
fprintf
fputc
fputs
fread
freopen
fscanf
fseek
fsetpos
ftell
fwrite
getc (2)
getchar
gets
getsessid
lseek
open
paws
perror
putc (3)
putchar
puts
raise
read
remove
rename
rewind
scanf
setattr
setbuf
setsessid
setvbuf
signal
system
tell
tmpfile
tmpnam
ungetc
vfprintf
vprintf
write

Notes

1. This function is implemented as a macro and consequently programs that invoke it will compile and link. However, the C run-time library does not define any FILE structures that the function could take as an argument.
2. Use of getc will cause the link to fail with getctext and __fillbuf undefined.
3. Use of putc will cause the link to fail with putctext and __flushbu undefined.

Compiler Options

The Developer's Toolkit supports two compilers: IBM VisualAge C^{++} (VACPP) and Microsoft Visual C^{++} (MSVC^{++}).

Reminder: Although you use a C^{++} compiler to compile SCC applications, the applications must be written in C.

VisualAge C++ (VACPP) Options

When using VACPP to compile executable files, use the following switches with the *icc* command to control the process:

Switch **Function**

/D_SCCTK Define a special symbol to ensure certain *.h* files are read from the proper subdirectory.

/Gn+ Do not put default library information into the object file.

/Gs+ Do not generate stack probes.

/I Use Developer's Toolkit customized settings to search for include files when compiling. See “Include File Directory Search Order” on page 3-9 for a description of the default file locations.

/O Compile without optimization (/O-) when creating an executable suitable for debugging. Optimization may be enabled (/O+) when creating a production version.

/Rn Do not incorporate the VACPP run-time environment into the compiled module.

/Ti Generate debug information (/Ti+) when creating an executable suitable for debugging. Debug information may be omitted (/Ti-) when creating a production version.

/Xi Do not search the path specified by the *include* environment variable.

Note: Developers writing extensions for IBM's CCA application must ensure CCA services are invoked using the __stdcall calling convention. One way to accomplish this is to specify the /Mt option with the icc command.

Notes on Intrinsic Functions: The Developer's Toolkit supports most VACPP intrinsic (automatically inlined) functions; the only unsupported intrinsic function is **_getTIBvalue**.

The following string and memory functions have intrinsic forms:

Table 3-4. VACPP String and Memory Functions with Intrinsic Forms

memcpy	strcat	strncat
memchr	strchr	strncmp
memcmp	strcmp	strncpy
memset	strcopy	strrchr
memmove	strlen	

To use the intrinsic forms of these functions, compile with optimization (/O+) and define _STRING_INTRINSICS_ before including <*string.h*>; define _STRING_INTRINSICS_ in your code with the following syntax:

```
#define _STRING_INTRINSICS_
```

As an alternative, define _STRING_INTRINSICS_ when compiling by using the /D_STRING_INTRINSICS_ compiler option.

Note: Do not define _STRING_INTRINSICS_ unless optimization is enabled.

The nonintrinsic forms of the following functions are not supported:

Table 3-5. VACPP Functions Whose Nonintrinsic Form Is Not Supported

_clear87	_control87	_status87

The following functions are not supported:

Table 3-6. Unsupported VACPP Functions

wcscat	wcscpy	wcsncmp
wcschr	wcslen	wcsncpy
wcscmp	wcsncat	wcsrchr

Note: Routines that invoke **setjmp** must be compiled without optimization.

Microsoft Visual C++ (MSVC++) Options

When using MSVC++ to compile executable files, use the following switches with the *cl* command to control the process:

Switch	Function
/D_SCCTK	Define a special symbol to ensure certain *.h* files are read from the proper subdirectory.
/Gs1000000	Do not generate stack probes. While the value entered does not need to be '1000000', it does need to exceed the size of any stack frame possible.
/I	Use Developer's Toolkit customized settings to search for include files when compiling. See "Include File Directory Search Order" on page 3-9 for a description of the default file locations.
/ML	Select a single-threaded library. This ensures that the compiler does not make any inappropriate assumptions about the contents of certain segment registers.

/O Compile without optimization (/Od) when creating an executable suitable for debugging. Optimization may be enabled (/Ox) when creating a production version.

/X Do not search the path specified by the *include* environment variable.

/Z7 Generate debug information (/Z7) when creating an executable suitable for debugging. Debug information must be incorporated into the executable, not placed in a program database (that is, do not specify /Zi). Debug information may be omitted when creating a production version.

/Zl Do not put default library information into the object file.

Do ***not*** specify the /GZ option.

Notes on Intrinsic Functions: The following MSVC^{++} intrinsic functions are always generated inline:

Table 3-7. MSVC^{++} Intrinisc Functions (Always Inlined)

_disable	_lrotl	_rotl
_enable	_lrotr	_rotr
_inp	_outp	_strset
_inpd	_outpd	
_inpw	_outpw	

Code the appropriate intrinsic compiler directive *#pragma* or enable optimization to use the inline forms of the following intrinsic functions:

Table 3-8. MSVC^{++} Intrinsic Functions (Inlined by Directive or Optimization)

abs	memcpy	strcpy
fabs	memset	strlen
labs	strcat	
memcmp	strcmp	

The following intrinsic functions are inlined if the appropriate optimization level is in effect:

Table 3-9. MSVC^{++} Intrinsic Functions (Inlined by Optimization)

atan	exp	sin
atan2	log	sqrt
cos	log10	tan

Notes:

1. While each function listed in Table 3-9 has a non-intrinsic form, the Developer's Toolkit supports only the non-intrinsic forms of **log** and **log10**.
2. The **_alloca** function is not supported.
3. Routines that invoke **setjmp** must be compiled without optimization.

Include File Directory Search Order

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

1. *scctk\cpqenv\nt\vacppmsm* (if building with VACPP) or *scctk\cpqenv\nt\msvcmasm* (if building with MSVC++)
2. *scctk\include\scc*
3. *scctk\include\common*
4. *udxtk\src\inc*

For example, the following compiler options might be specified to build the coprocessor-side portion of an SCC application with VACPP:

```
/Ic:\scctk\cpqenv\nt\vacppmsm /Ic:\scctk\include\scc /Ic:\scctk\include\common\
/Ic:\udxtk\src\inc
```

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

1. *scctk\include\host*
2. *scctk\include\common*

For example, the following compiler options might be specified to build the host-side portion of an SCC application with MSVC++:

```
/Ic:\scctk\include\host /Ic:\scctk\include\common
```

Assembler Options

The Developer's Toolkit supports the Microsoft** MASM assembler. Use the following switches with the *ml* command to control the process:

Switch	Function
/coff	Generate output as Common Object File Format (COFF), rather than Object Module Format (OMF). This switch should only be used when using the Microsoft Visual C^{++} compiler and the LINK linker.
/Cp	Preserve case of identifiers.
/I	Set the include path. See “Include File Directory Search Order” for a discussion of how to set these options.
/X	Do not search the path specified by the *include* environment variable.
/Zd /Zi	Generate debug information (/Zd and /Zi) when creating an executable suitable for debugging.[4] Debug information may be omitted when creating a production version.

[4] Debug information generated by MASM is not compatible with the format expected by the linker that ships with VisualAge C^{++}. Do not use the */Zd* or */Zi* switches when assembling files that will be linked using that linker.

Linker Options

The compiler used determines which linker must be used to create an executable file from the resulting object (*.obj*) files.

ILINK (VACPP Linker)

ILINK links object files created by the VACPP compiler. Use the following switch with the *ilink* command to control the process:

Switch	Function
/debug	Preserve debug information in the object files (/DEBUG) when creating an executable suitable for debugging. Debug information may be omitted (/NODEBUG) when creating a production version.
/nod	Do not search default libraries listed in the object files.
/noe	Do not search library extended directories.
/nofixed	Preserve relocation information in the executable file.[5]
/pm:vio	Specify a .EXE type. The type specified (PM, VIO, or NOVIO) does not matter, but if this switch is not provided the linker issues a warning message.

The libraries supplied in *scctk\lib\scc\nt\vacppmsm* were compiled with the /Gl+ option. Use the /OPTFUNC linker option to remove unreferenced functions and generate a smaller executable file.

LINK (MSVC^{++} Linker)

LINK links object files created by the MSVC^{++} compiler. Use the following switches with the *link* command to control the process:

Switch	Function
/debug	Preserve debug information in the object files (/DEBUG) when creating an executable suitable for debugging. If this option is specified, /DEBUGTYPE:CV and /PDB:none must also be specified. Debug information may be omitted when creating a production version.
/entry:startup	Specify the program entry point.
/fixed:no	Preserve relocation information in the executable file.
/nodefaultlib	Do not search the default libraries.

The libraries supplied in *scctk\lib\scc\nt\msvcmasm* were compiled with the /Gy option. Use the /OPT:REF linker option to remove unreferenced functions and generate a smaller executable file.

[5] The */nofixed* option is not required with ILINK versions prior to 3.5 or ILINK version 3.5 prior to fixpack WTC357.

Library Files to be Linked with Application

1. scctk\lib\scc\nt\msvcmasm\clib.lib
2. scctk\lib\scc\nt\msvcmasm\smlib.lib
3. scctk\lib\scc\nt\msvcmasm\cpqlib.lib
4. scctk\lib\scc\nt\msvcmasm\scclib.lib

Librarian Options

The compiler type used determines which librarian must be used to create a library (*.lib*) file from one or more object (*.obj*) files. ILIB creates libraries from files generated by VACPP, and LIB creates libraries from files generated by MSVC^{++}.

There are no required option settings for either librarian.

Translating

The Translator Utility (CPQXLT.EXE) translates a fully-compiled executable file into executable file able to run on the CP/Q^{++} operating system embedded within the coprocessor. The utility supports pure 32-bit executable files built from C or assembler source code; it does not support dynamic link libraries or C^{++} programs.

Debug information is translated for Windows NT executables built with the compilers described above, and for executables built with Microsoft-compatible debug information. The compiler, assembler, and linker must have been invoked with the proper options in order to generate debug information (for example, `VACPP /Ti+` or `MSVC++ /Z7`).

Because the entire executable (including any debug information, if present) is incorporated into the read-only disk image that is loaded into the coprocessor, it is recommended that the translation be performed twice, once to create a version of the executable containing debug information for the debugger's use, and a second time to create a version without debug information to be downloaded to the coprocessor.

Syntax

cpqxlt [*input-filename*] [*output-filename*] {*optional switches*}

where *input-filename* is the name of the NT-format executable file to translate (path information must also be provided if the file is not in the current directory), *output-filename* is the name of the CP/Q-format executable file to generate (path information must also be provided if the file is not in the current directory), and the optional switches are as follows:

Switch	Function
/base:address	Sets the address for the first loaded section. (The default is 0).
/nodebug	Suppresses translation of debug information.
/align:factor	Sets the alignment boundary for loaded sections. (The default is a 512-byte boundary).

The name of the output file should be no longer than 8 characters and the extension should be “.XLD”. The ICATCPW debugger may not be able to support debugging at the source level otherwise.

Building Read-Only Disk Images

The Disk Builder Utility (SCCRODSK.EXE) creates a read-only disk 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.

To build a disk image, create an ASCII text file that lists the files to incorporate into the disk image.[6] Each line in the file has the form:

filename [*stacksize*] [*arg1* [*arg2* [...]]]

where *filename* is the name of the file to incorporate into the disk image (path information must also be provided if the file is not in the directory in which the disk builder utility is invoked), *stacksize* is the number of bytes to allocate for the stack when the application the file contains is loaded and run. (The default is 4096 bytes.) Any additional tokens on the line (*arg1*, *arg2*, and so on) are passed to the application as invocation arguments. Blank lines are ignored and lines containing an asterisk in the first column are treated as comments.

| The name of the file to incorporate into the disk image must be no longer than 8
| characters and its extension must be no longer than 3 characters. The name,
| extension, and path must not contain whitespace.

For example, the following line in the input file

```
rte.xld 8192 a b cde "space " 'quote' "    " 1 2 3 4
```

causes rte.xld to be run with an 8K stack. On entry to `main ( ),` `argc` and `argv` have the following values:

```
argc       11
argv[0]    "rte.xld"
argv[1]    "a"
argv[2]    "b"
argv[3]    "cde"
argv[4]    "space"
argv[5]    "\'quote\'"
argv[6]    "       "
argv[7]    "1"
argv[8]    "2"
argv[9]    "3"
argv[10]   "4"
```

After the ASCII text file has been created, invoke the disk builder utility as follows:

sccrodsk *input-filename output-filename*

where *input-filename* is the name of the ASCII text file (path information must also be provided if the file is not in the current directory) and *output-filename* is the name of the disk image file to generate (path information must also be provided if the file is not in the current directory).

[6] At present, only one *.xld* file can be loaded into the coprocessor. Any others listed are incorporated into the disk image but are not loaded.

Downloading and Debugging

Once a file containing the read-only disk image of the application has been generated by SCCRODSK, the file 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:

CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\TL3rrrss.CLU

Syntax

druid [*image_fn pubkey_fn privkey_fn*[*coprocessor_number*]]

where

- *image_fn* is the name of the file containing the read-only disk image to download to the coprocessor. Path information must also be provided if the file is not in the current directory.
- *pubkey_fn* is the name of a file containing the public key to be associated with the application[7] (for example, S3KCLRPU.DRK). Path information must also be provided if the file is not in the current directory.
- *privkey_fn* is the name of a file containing an RSA keypair (for example, S3KCLRPP.DRK). Path information must also be provided if the file is not in the current directory. The public key must match the public key currently associated with the application in the coprocessor.[8]
- More than one coprocessor may be installed in a host. *coprocessor_number* identifies the coprocessor to which the read-only disk image is downloaded. 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

- The coprocessor's serial number[9]
- The current left and right bootcounts (see "Targeting Arguments" on page F-19 for details).

[7] That is, the public key to be associated with segment 3.

[8] That is, the public key currently associated with segment 3.

[9] That is, the value sccGetConfig returns in `pInfo->VPD.AdapterID`. Refer to the *IBM 4758 PCI Cryptographic Coprocessor CP/Q Operating System Application Programming Reference* for details.

- The name, creation date, and size of the image file last downloaded to the coprocessor
- The name of the file containing the public key associated with the application currently loaded in the coprocessor[10]

CP/Q++ will load and run the application after the coprocessor is rebooted. See Appendix C, “How to Reboot the IBM 4758” on page C-1 for a description of how to reboot the coprocessor.

Details on the use of CLU can be found in the *IBM 4758 PCI Cryptographic Coprocessor Custom Software 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 4758 PCI Cryptographic Coprocessor Interactive Code Analysis Tool (ICAT) User’s Guide* for details.

Note: The ICAT debugger may fail to start if an invalid directory (for example, one that does not exist) appears in the search path for executable files before the scctk\bin\nt directory.

[10] That is, the value of *pubkey_fn* supplied when DRUID last downloaded an application to the coprocessor.

Chapter 4. Testing an SCC Application in a Production Environment

The version of CP/Q++ that allows an application to be debugged includes certain components that are not present in the production version of CP/Q++. A developer may find it prudent to test the application running under a production version of CP/Q++ before releasing the application. The TNPrrrss.CLU file shipped with the Developer's Toolkit can be used to replace the debug version of CP/Q++ with a production version:[1]

```
CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\TNPrrrss.CLU
```

The developer should then rebuild the application (***without debug information or any code added for debugging purposes, including any infinite loop added to allow the debugger to attach***), create a read-only disk image, and download it to the coprocessor in the same manner as described in chapter 3.

If further debugging proves necessary, the debug version of CP/Q++ can be reloaded into the coprocessor as follows:

```
CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\TPRrrrss.CLU
```

Use of TNPrrrss.CLU and TPRrrrss.CLU in this manner preserves any state information the application has saved in battery-backed RAM and flash. "Downloading and Debugging" on page 3-13 describes how to clear such state information (if desired) before downloading an application.

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

Chapter 5. Packaging and Releasing an SCC Application

The design for the IBM 4758 PCI Cryptographic Coprocessor was motivated by the need to simultaneously satisfy the following requirements[1]:

1. Code must not be loaded into the coprocessor unless IBM or an agent IBM trusts has authorized the operation.
2. Once loaded into the coprocessor, code must not run or accumulate state unless the environment in which it runs is trustworthy.
3. Agents outside the coprocessor that interact with code running on the coprocessor must be able to verify that the code is legitimate and that the coprocessor is authentic and 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.

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 usually 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 will 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 very 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.

Attribute	Development	Production
Owner	“Generic developer”	Developer-unique identifier
Public Key	Generic (common) key	Developer-generated key

Prior to deployment, a developer must restore the coprocessor used for development to a state suitable for use in production[2] using TRSrrrss.CLU:[3]

```
CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\TRSrrrss.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 appear 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[4] RSA keypairs using CRUSIGNR’s KEYGEN function, for example:[5]

```
CRUSIGNR KEYGEN 2 S3KDEVPP.KEY S3KDEVPU.KEY \scctk\etc\DFT_SKEL.TKN
CRUSIGNR KEYGEN 2 DEVSGNPP.KEY DEVSGNPU.KEY \scctk\etc\DFT_SKEL.TKN
CRUSIGNR KEYGEN 2 DEVPKGPP.KEY DEVPKGPU.KEY \scctk\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 CLU uses to verify that IBM has authorized its use.

The KEYGEN function creates two KEY files, one containing both the private and public keys (for example, S3KDEVPP.KEY) and the other containing just the public key (for example, S3KDEVPU.KEY). The KEYGEN function also creates a file

[2] Or obtain a second IBM 4758.

[3] 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 *scctk\bin\nt*).

[4] 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.

[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 S3DEVPU.KEY\scctk\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.

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 zipfile). The developer also communicates the hash value of each public key file to IBM.[6]

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.
2. The following files generated by CRUSIGNR.[7]
   - ESTOWN2.E2T, which establishes ownership of segment 2.[8] 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.[9] This file is shipped with the IBM 4758 PCI Cryptographic Coprocessor Developer's Toolkit.
   - ESTOWN3.E3T, which establishes ownership of segment 3. IBM assigns the developer[10] 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.
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.

[6] The public key files and hash values must be transmitted separately—and preferably by way of separate channels—to ensure an adversary has not replaced the developer's public key with another. IBM typically provides a form for this purpose that can be returned by way of fax.

[7] See Appendix F, "Using Signer and Packager" on page F-1 for details on the contents of these files.

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

[9] 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.

[10] That is, an OEM or organization within an OEM.

The developer must build a version of the application (APP.ROD) suitable for release. The value of `pAgentID->DeveloperID` in any calls to sccSignOn and the value of `pRequestBlock->AgentID.DeveloperID` in any calls to sccRequest 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 F, “Using Signer and Packager” on page F-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.

The developer uses CRUSIGNR[11] 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
          APP.ROD 2 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 0 0
          x 0 0 65535 0 0
```

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[12], and *oem3* is the owner identifier assigned to the developer. See Appendix F, “Using Signer and Packager” on page F-1 for details.

A user can use CSUNCLU 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

[11] 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.

[12] 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.

saved in nonvolatile memory. To preserve such information, the developer creates a REMBURN3 command instead, for example:[13]

```
CRUSIGNR REMBURN3 MYAPP.R3T
            part version description
            DEVSGNPU.CRT DEVSGNPP.KEY
            APP.ROD 2 title revision
            S3KDEVPU.KEY S3KDEVPP.KEY
            ibm2 oem3
            1 1
            a 0 b 0 c 0 d 0 e 0 0
            x 0 0 65535 0 0
            x 0 0 65535 0 0
            x 0 0 65535 0 0
```

The developer can also use the Packager utility to create a file the user can download to a coprocessor shipped from the factory. For download to a coprocessor which is not installed in an IBM zSeries server, use CRUPKGR to combine the EMBURN3 command with certain files IBM provides.

```
CRUPKGR DEVPKGPU.CRT DEVPKGPP.KEY
            4 ESTOWN2.E2T EMBURN2.L2T ESTOWN3.E3T MYAPP.L3T FRESH.CLU 9
            part version description
```

where *part*, *version*, and *description* supply information that is incorporated into the output file (FRESH.CLU). See Appendix F, “Using Signer and Packager” on page F-1 for details.

A user can download FRESH.CLU to a coprocessor shipped from the factory after IBM’s system software has been installed in segment 1 (using, for example, CR1rrrss.CLU).

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 0 0
            x 0 0 65535 0 0
            x 0 0 65535 0 0
```

Then the developer uses CRUZPKG to combine the SUROWN3, EMBURN3, and REMBURN3 commands with the ESTOWN3.E3T file provided by IBM to create a

[13] 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.

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 F, “Using Signer and Packager” on page F-1 for details.

Place the IQYVP123.UDX file on a diskette in preparation for importing the CCA extensions to the zSeries PCI Cryptographic Coprocessor.[14] (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.)

[14] 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.[1]

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:

   ```
   CSUNCLU \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:

     ```
     CSUNCLU \logfile-directory\CLU.LOG PL \scctk\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.

[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 *scctk\bin\nt*).

- **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:

  CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\CE1rrrss.CLU

  Then load TDVrrrss.CLU as indicated in “Segment 1 Current” below.

- **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:

  CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\TDVrrrss.CLU

  Warning

  Attempting to load a copy of TDVrrrss.CLU for a model 002 or 023 coprocessor into a model 001 or 013 coprocessor can leave the coprocessor in an unusable state. Assistance from IBM is required to correct this condition.

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

  CSUNCLU \logfile-directory\CLU.LOG PL \scctk\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:

  CSUNCLU \logfile-directory\CLU.LOG PL \scctk\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 4758” 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 4.

3. If the owner identifier associated with coprocessor segment 2 is 2, relinquish ownership of segment 2 by loading CRSrrrss.CLU into the coprocessor, for example:

   CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\CRSrrrss.CLU

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

   If this command succeeds, segments 2 and 3 become UNOWNED. Continue with step 2.[2]

[2] If CRS12200.CLU is used to relinquish ownership of segment 2, load CR1rrrss.CLU as indicated in step 2. The contents of segment 1 cannot be used as a guide in this case.

4. If the owner identifier associated with coprocessor segment 2 is 3 and the owner identifier associated with segment 3 is 6 and both segments are RUNNABLE, load into the coprocessor TPRrrrss.CLU or TPSrrrss.CLU, which contain a version of CP/Q++ that supports the debugging of applications, for example:

   CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\TPRrrrss.CLU

   or

   CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\TPSrrrss.CLU

   The developer should load TPSrrrss.CLU if the developer's application needs to use the coprocessor's serial port and load TPRrrrss.CLU otherwise.

   This completes preparation of the coprocessor for use as a development platform. Continue with step 5.

   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:

   CSUNCLU \logfile-directory\CLU.LOG PL \scctk\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 4758" 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 5.

   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:

   CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\TRSrrrss.CLU

   Continue with step 2.

Compiling, Assembling, and Linking

5. Compile and link the application under development. Specify the appropriate options to ensure debugging information is incorporated into the .EXE file produced (APP.EXE[3]).

[3] The developer is free to choose a different file name.

Translating

6. Translate the application to the CP/Q++ executable format:[4]

   CPQXLT APP.EXE APP.XLD

   Translate the application a second time, omitting any debug information:[4]

   CPQXLT APP.EXE DOWNLOAD\APP.XLD /NODEBUG

   The output files generated in the two translations must have the same name (for example, APP.XLD) and so must be placed in separate directories.

Building Read-Only Disk Images

7. Create or modify the text file (RODISKIN.TXT[5]) that lists the names of the executable files to be loaded and run. In our example, if the developer wanted an 8K stack, the contents of RODISKIN.TXT might be:

   ```
   c:\scctk\obj\app\nt\msvcmasm\download\app.xld 8192
   ```

8. Build a read-only disk image that incorporates the application:[6]

   SCCRODSK RODISKIN.TXT APP.ROD

Downloading and Debugging

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

   CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\TL3rrrss.CLU

10. Download the file generated in step 8 to the coprocessor:[7]

    DRUID APP.ROD C:\SCCTK\ETC\S3KCLRPU.DRK C:\SCCTK\ETC\S3KCLRPP.DRK

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

12. Start the debugger and attach to the application:

    ICATCPW

    Refer to the *IBM 4758 PCI 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 5.

Testing an SCC Application in a Production Environment

13. 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 5 through 8 of this procedure. In step 5 do not specify the options that incorporate debugging information in the .EXE file. In step 6, only one translation need be performed.

[4] The developer is free to choose a different file name for the output file although the name must conform to rules governing 8.3-character MS-DOS filenames if source debugging is desired. The first argument is the name of the .EXE file created in step 5.

[5] The developer is free to choose a different file name.

[6] The developer is free to choose a different name for the output file. The first argument is the name of the text file mentioned in the previous footnote.

[7] The first argument is the name of the file created in step 8.

14. Load a production-level copy of CP/Q++ (one that lacks the components that support the debugging of applications) into the development coprocessor using TNPrrrss.CLU:

    ```
    CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\TNPrrrss.CLU
    ```

15. Clear any state saved in nonvolatile memory using the procedure described in step 9.

16. Download the file generated in step 8 to the coprocessor using the procedure described in step 10.

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

    If changes to the application prove necessary, make them and continue with step 13. If additional debugging is required, reload TPRrrrss.CLU as indicated in step 4 and continue with step 5.

Packaging and Releasing an SCC Application

18. Reset the development coprocessor using TRSrrrss.CLU:

    ```
    CSUNCLU \logfile-directory\CLU.LOG PL \scctk\etc\TRSrrrss.CLU
    ```

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

19. 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 4758 PCI Cryptographic Coprocessor CCA Support Program Installation Manual*.

20. Generate three[8] RSA keypairs using CRUSIGNR's KEYGEN function[9]:

    ```
    CRUSIGNR KEYGEN 2 S3KDEVPP.KEY S3KDEVPU.KEY \scctk\etc\DFT_SKEL.TKN
    CRUSIGNR KEYGEN 2 DEVSGNPP.KEY DEVSGNPU.KEY \scctk\etc\DFT_SKEL.TKN
    CRUSIGNR KEYGEN 2 DEVPKGPP.KEY DEVPKGPU.KEY \scctk\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 CLU uses to verify that IBM has authorized its use.

21. 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[10] to ensure an adversary has not replaced the developer's public key file with another.

    IBM returns the following to the developer:

[8] 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.

[9] This version of the KEYGEN command does not encrypt the private keys in the *PP.KEY files, which may not provide the degree of security required. To encrypt the private keys with the CCA master key, specify 0 rather than 2 for the second argument, for example:

```
CRUSIGNR KEYGEN 0 S3KDEVPP.KEY S3DEVPU.KEY\scctk\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.

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

a. Certificates for the CRUSIGNR and CRUPKGR public keys (DEVSGNPU.CRT and DEVPKGPU.CRT, respectively). The developer provides these certificates as input to CRUSIGNR and CRUPKGR, as appropriate.[11]

b. The following files generated by CRUSIGNR.[12] 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.[13] 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.[14] This file is shipped with the IBM 4758 PCI Cryptographic Coprocessor Developer's Toolkit.
   - ESTOWN3.E3T, which establishes ownership of segment 3. IBM assigns the developer[15] 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.

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.

[11] 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.

[12] See Appendix F, “Using Signer and Packager” on page F-1 for details on the contents of these files.

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

[14] 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.

[15] That is, an OEM or organization within an OEM.

22. IBM typically supplies the files listed in step 21 in zipped form.

23. 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 sccSignOn and the value of `pRequestBlock->AgentID.DeveloperID` in any calls to sccRequest to the owner identifier assigned by IBM.

24. Create an EMBURN3 command[16] that incorporates the application, IBM's segment 2 owner ID, the developer's owner ID, and the developer's unique keys:

```
CRUSIGNR EMBURN3 MYAPP.L3T
            part version description
            DEVSGNPU.CRT DEVSGNPP.KEY
            APP.ROD 2 title revision
            S3KDEVPP.KEY ESIGDEV.SIG
            ibm2 oem3
            1 1
            a 0 b 0 c 0 d 0 e 0 0
            x 0 0 65535 0 0
            x 0 0 65535 0 0
```

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[17], and *oem3* is the owner identifier assigned to the developer. See Appendix F, "Using Signer and Packager" on page F-1 for details.

[16] 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.

[17] 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.

A user can use CSUNCLU 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:[18]

```
CRUSIGNR REMBURN3 MYAPP.R3T
            part version description
            DEVSGNPU.CRT DEVSGNPP.KEY
            APP.ROD 2 title revision
            S3KDEVPU.KEY S3KDEVPP.KEY
            ibm2 oem3
            1 1
            a 0 b 0 c 0 d 0 e 0 0
            x 0 0 65535 0 0
            x 0 0 65535 0 0
            x 0 0 65535 0 0
```

25. 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
            DEVSGNPU.CRT DEVSGNPP.KEY
            S3KDEVPP.KEY
            ibm2 oem3
            a 0 b 0 c 0 d 0 e 0 0
            x 0 0 65535 0 0
            x 0 0 65535 0 0
            x 0 0 65535 0 0
```

26. To create a package for download to a coprocessor which is not installed in an IBM zSeries server, combine the task file IBM supplies with the file containing the EMBURN3 command:

```
CRUPKGR DEVPKGPU.CRT DEVPKGPP.KEY
            4 ESTOWN2.E2T EMBURN2.L2T ESTOWN3.E3T MYAPP.L3T FRESH.CLU 9
            part version description
```

where *part*, *version*, and *description* supply information that is incorporated into the output file. See Appendix F, "Using Signer and Packager" on page F-1 for details.

A user can download FRESH.CLU to a coprocessor shipped from the factory after IBM's system software has been installed in segment 1 (using, for example CR1rrrss.CLU).

To create a package which will be imported to and activated on a PCI 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:

[18] 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.

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

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

Place the IQYVP123.UDX file on a diskette in preparation for importing the CCA extensions to the zSeries PCI Cryptographic Coprocessor.[19] (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.)

[19] For more information on how to install the CCA extensions on a zSeries PCI Cryptographic Coprocessor, refer to *zSeries 900 Support Element Operations Guide*.

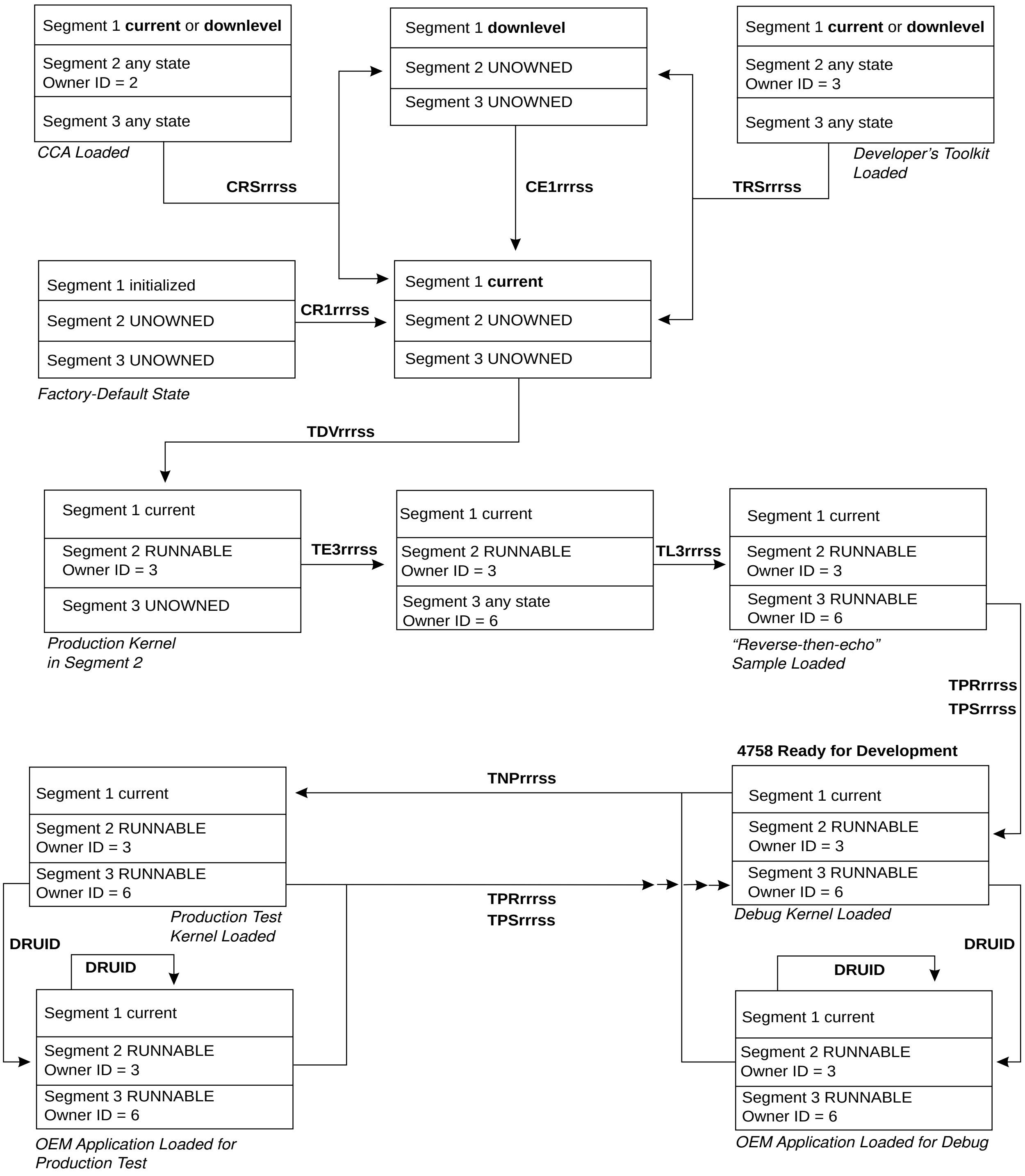


Figure A-1. Overview of the Development Process

Appendix B. Using CLU

The Coprocessor Load Utility (CSUNCLU.EXE) interacts with the coprocessor's ROM-based system software to update software in flash.[1] The Coprocessor Load Utility can also obtain information about the coprocessor, reset the coprocessor, or validate the software in the coprocessor.

Syntax

CSUNCLU *logfilename* {**PL** | **RS** | **SS** | **ST** | **VA**} [*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.

- 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.

[1] The syntax diagram in this appendix assumes the directory that contains the various utilities shipped with the support program is in the search path for executable files (that is, the PATH environment variable includes *pkcs11\bin\nt*).

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 4758

An IBM 4758 can be rebooted in any of several ways:

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

   CSUNCLU \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 Windows NT, this can be accomplished by stopping and restarting the cryptont device in the dialog invoked by Start -> Settings -> Control Panel -> Devices or by issuing the following commands:

   net stop cryptont
   net start cryptont

   On Windows 2000, invoke Start -> Settings -> Control Panel -> System -> Hardware -> Device Manager; the 4758 appears under the "Multifunction Adapters" category.

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 sccOpenAdapter 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 *scctk\bin\nt*).

Appendix D. Building SCC Applications with Microsoft Developer Studio 97 or Microsoft Visual Studio 6.0

This appendix describes how to configure Microsoft Developer Studio 97** or Microsoft Visual Studio 6.0 to ensure the proper compiler and linker options are used to build an SCC application. These instructions apply to Microsoft Developer Studio 97** with Microsoft Visual C++ 5.0 and to Microsoft Visual Studio 6.0 with Microsoft Visual C++ 6.0.

Required Settings for the Host-Side Portion of an SCC Application

Open the Project Settings dialog (Project/Settings...). The required settings under each tab are as follows:

- **C/C++**
  - Precompiled Headers Category -
    - Check "Not using precompiled headers"
  - Add the following paths (adjusted as necessary to account for where the Developer's Toolkit directory tree is located) to "Additional include directories". The paths must be added in the order shown:
    - *...\scctk\include\host*
    - *...\scctk\include\common*
- **Link**
  - Object/library modules: Add *cryptont.lib*
  - Add the following path (adjusted as necessary to account for where the Developer's Toolkit directory tree is located) to "Additional library path".
    - *...\scctk\lib\host\nt\msvcmasm*

Required Settings for the Coprocessor-Side Portion of an SCC Application

The project created to hold the coprocessor-side portion of an SCC application must be a Win32 Console Application.

Open the Project Settings dialog (Project/Settings...). The required settings under each tab are as follows:

- **C/C++** -
  - General Category -
    - Optimizations: Compile without optimization ("Disable (Debug)") when creating an executable suitable for debugging. Optimization may be enabled when creating a production version.
    - Debug info: Select "C7 Compatible"
  - Code Generation Category -
    - Use run-time library: Select a single-threaded library. This ensures the compiler does not make any inappropriate assumptions about the contents of certain segment registers.

- Precompiled Headers Category -
  - Check “Not using precompiled headers”
- Preprocessor Category -
  - Add _SCCTK to the list of items in “Preprocessor definitions”
  - Check “Ignore standard include paths”

    Add the following paths (adjusted as necessary to account for where the Developer’s Toolkit directory tree is located) to “Additional include directories”. The paths must be added in the order shown:
    - *...\scctk\cpqenv\nt\msvcmasm*
    - *...\scctk\include\scc*
    - *...\scctk\include\common*
- Project Options -
  - Add “/Gs1000000”
  - Remove “/GZ” if it is present

- **Link**
  - General Category -
    - Object/library modules: Remove all libraries listed (for example, *kernel32.lib*). Add *clib.lib*, *cpqlib.lib*, *scclib.lib*, and *smlib.lib*
    - Check “Generate debug info” and “Ignore all default libraries”
    - Uncheck “Link incrementally”
  - Customize Category -
    - Uncheck “Link incrementally”
    - Uncheck “ Use program database”
  - Debug Category -
    - Check “Debug info” and select “Microsoft format”
  - Input Category -
    - Check “Ignore all default libraries”
    - Object/library modules: Specify *clib.lib*, *cpqlib.lib*, *scclib.lib*, and *smlib.lib*
    - Add the following path (adjusted as necessary to account for where the Developer’s Toolkit directory tree is located) to “Additional library path”.
      - *...\scctk\lib\scc\nt\msvcmasm*
  - Output Category -
    - Entry-point symbol: Specify “startup”
  - Project Options -
    - Add “/fixed:no”
    - The libraries supplied in *scctk\lib\scc\nt\msvcmasm* were compiled with the /Gy option. Use the /OPT:REF linker option to remove unreferenced functions and generate a smaller executable file.

Appendix E. Building SCC 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 SCC application from a command prompt. (The host-side portion of an SCC application may be built in the same manner as any other NT application. The only requirement is to use the appropriate compiler options to ensure the directories listed in "Include File Directory Search Order" on page 3-9 are searched in the proper order.)

The makefiles are named *cpqenvtk.mak* and are located beneath the *scctk\cpqenv* subdirectory. The *nt\msvcmasm* subdirectory contains a version of *cpqenvtk.mak* tailored for use with Microsoft Visual C++ and the *nt\vacppmsm* subdirectory contains a version of *cpqenvtk.mak* tailored for use with IBM VisualAge C++. The makefiles are used to build the coprocessor-side portion of an SCC application; no makefiles are provided for the host-side portion.

cpqenvtk.mak makes the following assumptions:

- The developer is using the directory structure into which the Developer's Toolkit places files by default (for example, source code is in a directory under *scctk\src*).
- The SCCTK_FS_ROOT environment variable points to the base of the Toolkit directory structure (for example, *c:\scctk*).
- The SCCTK_ENV environment variable points to the directory that contains the appropriate *cpqenvtk.mak* file (for example, *c:\scctk\cpqenv\nt\vacppmsm*).
- The directory that contains the tools shipped with the Toolkit (for example, *c:\scctk\bin\nt*) is in the PATH.

cpqenvtk.mak may be included in other makefiles using a statement of the form

```
!INCLUDE $(SCCTK_ENV)\CPQENVTK.MAK
```

The operation of *cpqenvtk.mak* can be customized by setting the following variables in a makefile that includes *cpqenvtk.mak*. The variables should be set after *cpqenvtk.mak* is included:

- SRC_SUBDIR is the path (relative to *scctk\src*) of the directory that contains the source being compiled (for example, *samples*). This environment variable must be defined.

  cpqenvtk.mak adds `$(SCCTK_FS_ROOT)\src\$(SRC_SUBDIR)` to the include file search path for the compiler and the assembler and places any object and executable files generated during the build in `$(SCCTK_FS_ROOT)\obj\$(SRC_SUBDIR)\nt\msvcmasm` or in `$(CPQ_FS_ROOT)\obj\$(SRC_SUBDIR)\nt\vacppmsm`, as appropriate.

- PAR_SUBDIR is the path (relative to *scctk\src*) of the parent directory of SRC_SUBDIR. This environment variable need not be defined.

  cpqenvtk.mak adds `$(SCCTK_FS_ROOT)\src\$(PAR_SUBDIR)` to the include file search path for the compiler and the assembler.

- SRC_INC is an include directive to place at the beginning of the include search path for the compiler and the assembler. If defined, SRC_INC must include the `/I` flag (for example, `SRC_INC = /I.`). This environment variable need not be defined.
- OPTIMIZE supplies arguments to the compiler to control optimization and the generation of debug information. The default is to generate optimized code without debug information (that is, MSVC++ `/Ox` or VACPP `/O+ /Ti-`). To generate code suitable for debugging, set OPTIMIZE to `/Od /Z7` (MSVC++) or to `/O- /Ti+` (VACPP). This environment variable need not be defined.
- COM_DEF specifies symbols to define (that is, `/D`) or undefine (that is, `/U`) for compilation. This environment variable need not be defined.
- COM_USER specifies additional options to be passed to the compiler. COM_USER appears last on the command line, so any options it specifies override options specified elsewhere. This environment variable need not be defined.
- COM_WARN specifies which messages the compiler will generate (that is, COM_WARN is the `/W` compiler option). This environment variable need not be defined.
- ASM_USER specifies additional options to be passed to the assembler. ASM_USER appears last on the command line, so any options it specifies override options specified elsewhere. This environment variable need not be defined.
- LINK_USER specifies additional options to be passed to the linker. LINK_USER appears last on the command line, so any options it specifies override options specified elsewhere. This environment variable need not be defined.
- XLAT_USER specifies additional options to be passed to the translator (CPQXLT.EXE). XLAT_USER appears last on the command line, so any options it specifies override options specified elsewhere. This environment variable need not be defined.
- LIBR_USER specifies additional options to be passed to the librarian. LIBR_USER appears last on the command line, so any options it specifies override options specified elsewhere. This environment variable need not be defined.

Appendix F. 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 4758 PCI 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

[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.

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 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:[4]

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

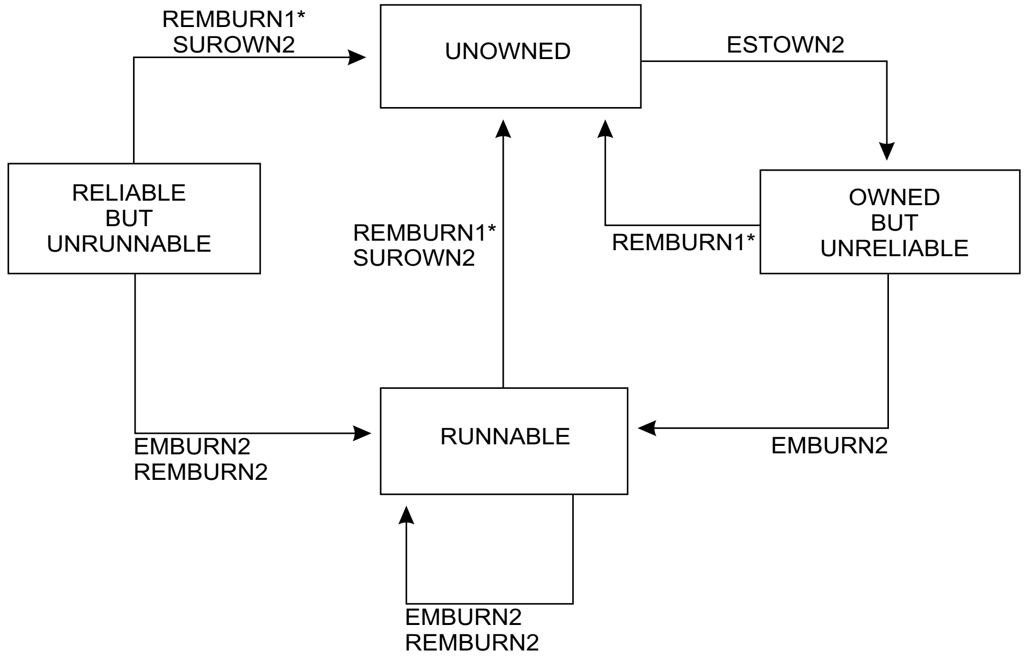
If the coprocessor's tamper-detection circuitry detects an attempt to compromise the physical security of the coprocessor, all data in BBRAM is cleared and Miniboot changes segment 2's state to UNOWNED. Certain unusual errors affecting segment 1 or segment 2 can also cause segment 2's state to change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

[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 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference*.

[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.

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.[5]

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 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[6] 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.

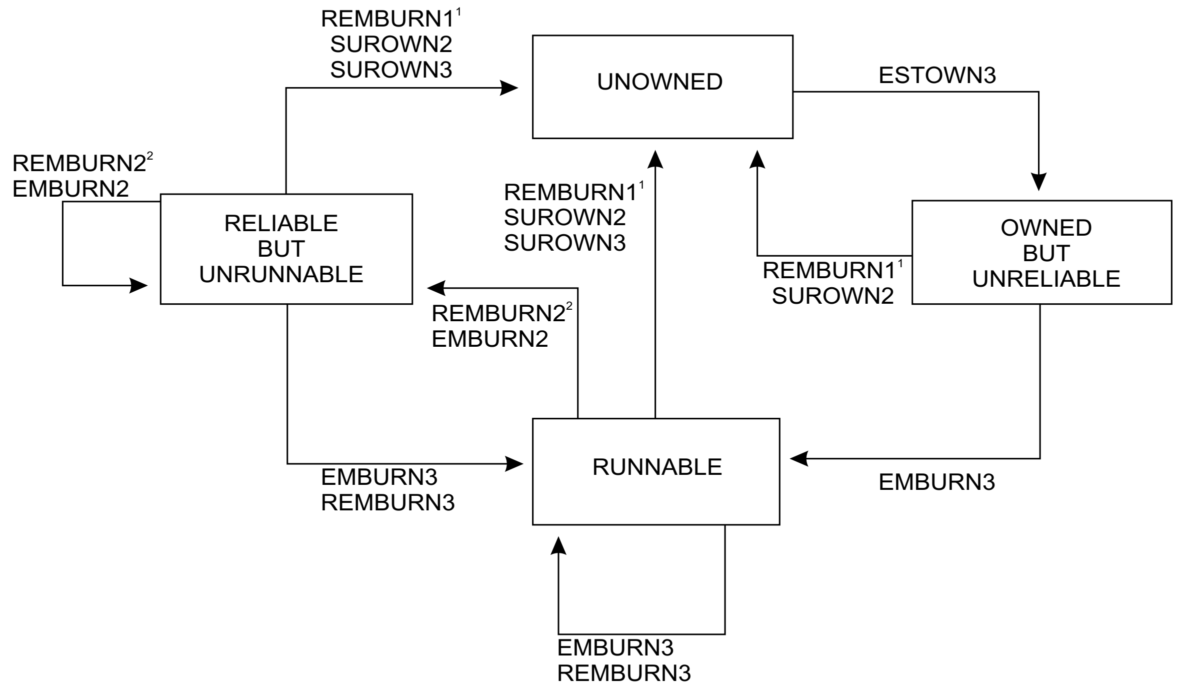


* This transition occurs if the trust arguments associated with segment 2 indicate the new segment 1 is not trusted. See "Trust and Countersignature Arguments" on page F-17 for details.

Figure F-1. State Transitions for Segment 2

[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.

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



[1] This transition occurs if the trust arguments associated with segment 3 indicated the new segment 1 is not trusted. See "Trust and Countersignature Arguments" on page F-17 for details.

[2] This transition occurs if the trust arguments associated with segment 3 indicated the new segment 2 is not trusted. See "Trust and Countersignature Arguments" on page F-17 for details.

Figure F-2. State Transitions for Segment 3

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 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 *scctk\bin\nt*).

Syntax

CRUSIGNR *function* **-F** *parm_file_name* [**-Q**]

CRUSIGNR *function* [*arguments*] [**-Q**]

CRUSIGNR

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 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]

[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 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.

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 F-16 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 F-16 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 F-17 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 2s 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 F-17 for details.
- *type1_target_args* specifies certain conditions that the coprocessor checks before it accepts the new segment 2 information. See “Targeting Arguments” on page F-19 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.

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 F-16 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 F-16 for details.
- *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 F-17 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.

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

- *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 F-17 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 F-17 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 F-19 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.
- *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 F-19 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 F-16 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 F-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 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 F-19 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:

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

- 0 - Use MASTER for the *rule_array* parameter. This causes the coprocessor to encrypt the RSA keypair in *keypair_fn* with the coprocessor CCA master key before returning the keypair.
- 1 - Use XPORT for the *rule_array* parameter. This causes the coprocessor to encrypt the RSA keypair in *keypair_fn* with the key-encrypting key in *transkey_fn* before returning the keypair.
- 2 - Use CLEAR for the *rule_array* parameter. This causes the coprocessor to return the RSA keypair in *keypair_fn* "in the clear" (that is, the file is not encrypted).
- 3 - Use RETAIN for the *rule_array* parameter. This causes the coprocessor to retain the RSA keypair and not write it to the host. Specify 1 as the last argument if the retained key may be cloned and specify 0 if it may not.

Refer to the *IBM 4758 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 F-16 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 F-16 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 F-17 for details.
- *pubkey_fn* is the name of the file that contains the public key to be associated with segment 2.[12] 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.

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

- *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 F-17 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 F-19 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.[13] See “Trust and Countersignature Arguments” on page F-17 for details.

REMBURN3 - Replace Software in Segment 3

Syntax

REMBURN3 *out_fn filedesc_args sigkey_args image_args pubkey_fn privkey_fn 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:

- *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.
- *filedesc_args* provides certain descriptive information that is incorporated into the output file. See “File Description Arguments” on page F-16 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 F-16 for details.
- *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 F-17 for details.

[13] The change to segment 3’s state and the updates of segment 2 are performed automatically.

- *pubkey_fn* is the name of the file that contains the public key to be associated with segment 3.[14] Path information must also be provided if the file is not in the current directory. This key is downloaded to the coprocessor (replacing the key that is already there) and is used to authenticate subsequent commands that affect segment 3.
- *privkey_fn* is the name of a file that contains an RSA keypair. Path information must also be provided if the file is not in the current directory. The public key in this file must be the public key associated with segment 3. CRUSIGNR includes in the output file a hash of the file enciphered using the private key from the *privkey_fn* file. The coprocessor uses the public key associated with segment 3 to validate the hash and rejects the REMBURN3 command if the validation fails.
- *seg2_ownid* is the owner identifier associated with segment 2. The coprocessor compares this value to the owner identifier stored in the coprocessor and rejects the REMBURN3 command if the two identifiers are not equal.
- *seg3_ownid* is the owner identifier associated with segment 3. The coprocessor compares this value to the owner identifier stored in the coprocessor and rejects the REMBURN3 command if the two identifiers are not equal.
- *trust1_fl* 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 F-17 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. See "Trust and Countersignature Arguments" on page F-17 for details.
- *type3_target_args* specifies certain conditions that the coprocessor checks before it accepts the new segment 3 information. See "Targeting Arguments" on page F-19 for details.

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.[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:

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

[15] This also changes segment 3's state to UNOWNED.

- *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 F-16 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 F-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 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 F-19 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.
- *filedesc_args* provides certain descriptive information that is incorporated into the output file. See “File Description Arguments” on page F-16 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 F-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 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 F-19 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:

partnumber ECnumber 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 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[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:

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 4758 the code is intended to execute. Recognized values are 1 (for code that targets the 4758 Model 001 and Model 013) and 2 (for code that targets the 4758 Model 002 and Model 023).
- *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; 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 F-19 for details.

Trust and Countersignature Arguments

Recall that one of the primary design goals for the IBM 4758 PCI 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

[17] The public key is compiled into CLU.

caused by an EMBURN command are always untrusted.) See “Coprocessor Memory Segments and Security” on page F-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.

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.[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[18] 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

{**NoCSig2** | *privkey_fn type2_target_args*} {**NoCSig3** | *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 F-19 or details.

[18] See Figure F-1 on page F-3 and Figure F-2 on page F-4.

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

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

where

- *RTCid* and *RTCid_mask* specify a range of permitted values for the serial number incorporated in the coprocessor chip that implements the real-time clock and the battery-backed RAM.[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.

  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.[20] 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

[19] That is, the value sccGetConfig returns in `pInfo->AdapterID`. Refer to the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference* for details.

[20] That is, the value sccGetConfig returns in `pInfo->VPD.sn`. Refer to the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference* for details.

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

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

- *VPDpartno* and *VPDpartno_mask* specify a range of permitted values for the coprocessor's IBM part number.[21] 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.[22] Each of these arguments is a string and may contain as many as seven characters. The arguments should have the same length.

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

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

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

- *VPDflags* and *VPDflags_mask* specify a range of permitted values for the coprocessor's VPD flags.[23] Each of these arguments is a string and may

[21] That is, the value sccGetConfig returns in `pInfo->VPD.pn`. Refer to the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference* for details.

[22] That is, the value sccGetConfig returns in `pInfo->VPD.ec`. Refer to the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference* for details.

[23] That is, the value sccGetConfig returns in the last sixteen bytes of `pInfo->VPD.reserved`. Refer to the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference* for details.

contain as many as 32 characters. The arguments should have the same length.

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

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

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

- *bootcount_fl, bootcount_left,* and *bootcount_right* are used as follows: each time the coprocessor boots, it increments one of two counters. The "left count" is a 16-bit number kept in EEPROM that is zero when the coprocessor leaves the factory and is incremented each time the coprocessor boots in a zeroized state (that is, each time the coprocessor is revived after having cleared memory upon detecting an attempt to compromise the coprocessor's security). The "right count" is a 32-bit number that is zero when the coprocessor leaves the factory and is incremented each time the coprocessor is booted in a nonzeroized state.[24] It is set to zero if the coprocessor detects an attempt to compromise the coprocessor's security.[25] *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*.

[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.

- *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 F-17 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 hexidecimal number. Each byte in the hexadecimal number is set to 0xFF if the corresponding character in *segflags_mask* is ASCII 1 and is set to 0x00 otherwise.

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

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

  – *revision_min* and *revision_max* specify a range of permitted values for the segment's revision level (as specified in the EMBURN or REMBURN command that loaded the segment into the coprocessor - see "Image File Arguments" on page F-17 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 and *hash* is either 0 or a string containing 20 characters.

    If *hash_fl* is 1, *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 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, *hash* must also be 0. 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 for *hash_fl* and 0 for *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 -F *parm_file_name* [**-Q**]

CRUPKGR [*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 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

[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 *scctk\bin\nt*).

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 F-16 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*.
- *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 F-16 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.[27]

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 **/?**.

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).

[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 *scctk\002\bin\zSeries*).

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 IQYVP*xxx*.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.

Appendix G. 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:

@server
z/OS
zSeries
IBM
VisualAge

Microsoft, Microsoft Assembler, Microsoft Developer Studio 97, Microsoft Visual C++, Windows, and Windows NT are trademarks or registered trademarks of Microsoft Corporation.

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

List of Abbreviations and Acronyms

API	application program interface
ASCII	American National Standard Code for Information Interchange
CCA	Common Cryptographic Architecture
CLU	Coprocessor Load Utility
CP/Q	Control Program/Q
FIPS	Federal Information Processing Standard
IBM	International Business Machines
ICAT	Interactive Code Analysis Tool
I/O	input/output
IPL	initial program load
ISO	International Organization for Standardization
MD5	message digest 5 (hashing algorithm)
PCI	peripheral component interconnect
PDF	portable document format
RSA	Rivest-Shamir-Adleman (algorithm)
SCC	secure cryptographic coprocessor
TOD	time-of-day (clock)
UART	universal asynchronous receiver/transmitters
VPD	vital product data

Glossary

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

- The *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42 Street, New York, New York 10036. Definitions are identified by the symbol (A) following the definition.
- The *Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) following the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) following the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

A

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

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

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

adapter. Synonym for *expansion card*.

agent. (1) An application that runs within the IBM 4758 PCI 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 4758 PCI Cryptographic Coprocessor uses BBRAM to store persistent data for SCC applications, as well as the coprocessor device key.

BBRAM. Battery-backed random access memory.

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

C

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

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

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.

Comm_Mgr. Communications Manager.

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

Common Cryptographic Architecture (CCA) API. The application program interface used to call Common Cryptographic Architecture functions; it is described in the *IBM 4758 CCA Basic Services Reference and Guide*, &frefform..

Communications Manager (Comm_Mgr). A CP/Q^{++} extension for the IBM 4758 PCI Cryptographic Coprocessor that manages communication among the host device driver, SCC applications, and CP/Q^{++}. It handles the receipt and delivery of request headers, and the inbound and outbound data buffers.

Control Program/Q (CP/Q). The operating system embedded within the IBM 4758 PCI Cryptographic Coprocessor. The version of CP/Q used by the coprocessor—including extensions to support cryptographic and security-related functions—is known as CP/Q^{++}.

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

Coprocessor Load Utility (CLU). A program used to load validated code into the IBM 4758 PCI Cryptographic Coprocessor.

CP/Q. Control Program/Q.

Cryptographic Coprocessor (IBM 4758). An expansion card that provides a comprehensive set of cryptographic functions to a workstation.

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

cryptography. (1) The transformation of data to conceal its meaning. (2) In computer security, the principles, means, and methods used to 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 CP/Q^{++} extension that manages the IBM 4758 PCI Cryptographic Coprocessor DES processing hardware.

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

DES_Mgr. Data Encryption Standard Manager.

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

E

encipher. (1) To scramble data or convert it to a secret code that masks its meaning. (2) Contrast with *decipher*.

enciphered data. (1) Data whose meaning is concealed from unauthorized users or observers. (2) See also *ciphertext.*

expansion board. Synonym for *expansion card*.

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

expansion slot. One of several receptacles in a PC or RS/6000 machine into which a user can install an expansion card.

F

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

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

FIPS. Federal Information Processing Standard

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

H

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

host. As regards to the IBM 4758 PCI Cryptographic Coprocessor, the workstation into which the coprocessor is installed.

I

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 4758 PCI 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 4758 PCI Cryptographic Coprocessor. This latch can be used, for example, to detect when the cover of the coprocessor host workstation has been opened. The intrusion latch does not trigger the destruction of data stored within the coprocessor.

I/O. Input/output.

IPL. Initial program load.

ISO. International Organization for Standardization.

J

jumper. A wire that joins two unconnected circuits.

K

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

M

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

miniboot. Software within the IBM 4758 PCI Cryptographic Coprocessor designed to initialize the CP/Q^{++} operating system and to control updates to flash memory.

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

N

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

NIST. National Institute of Science and Technology.

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

NT. See *Windows NT*.

P

passphrase. In computer security, a string of characters known to the computer system and to a user; the user must specify it to gain full or limited access to the system and to the data stored therein.

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

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

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 CP/Q^{++} extension that manages the IBM 4758 PCI Cryptographic Coprocessor PKA processing hardware.

R

Random Number Generator Manager (RNG_Mgr). A CP/Q^{++} extension that manages the IBM 4758 PCI Cryptographic Coprocessor hardware-based random number generator.

reduced instruction set computer (RISC). A computer that processes data quickly by using only a small, simplified instruction set.

return code. (1) A code used to influence the execution of succeeding instructions. (A) (2) A value returned to a program to indicate the results of an operation requested by that program.

RNG_Mgr. Random Number Generator Manager.

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

S

SCC. Secure cryptographic coprocessor.

SCC_Mgr. Secure Cryptographic Coprocessor Manager.

secure cryptographic coprocessor (SCC). An alternate name for the IBM 4758 PCI Cryptographic Coprocessor. The abbreviation “SCC” is used within the product software code.

secure cryptographic coprocessor (SCC) application. (1) An application that runs within the IBM 4758 PCI Cryptographic Coprocessor. (2) Synonym for *agent*.

Secure Cryptographic Coprocessor Manager (SCC_Mgr). A CP/Q^{++} extension that provides high-level management of all agents running within a IBM 4758 PCI Cryptographic Coprocessor. As the “traffic cop”, the SCC_Mgr identifies agents and controls the delivery of their messages and data.

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

SMIT. System Management Interface Tool.

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

System Management Interface Tool (SMIT). An AIX utility program used to maintain the system in good working order and to modify the system to meet changing requirements.

T

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

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

TOD clock. Time-of-day clock.

U

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

V

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

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

VPD. Vital product data.

W

Windows NT. A Microsoft operating system for personal computers.

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

Numerics

IBM 4758. IBM 4758 PCI Cryptographic Coprocessor.

Index