



A New Direction in Mixed-Signal

Express DX

Software Development Kit

User Guide

No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language in any form by any means without the written permission of Exar Corporation.

Licensing and Government Use

Any Exar software ("Licensed Programs") based on Hifn Technology described in this document is furnished under a license and may be used and copied only in accordance with the terms of such license and with the inclusion of this copyright notice. Distribution of this document or any copies thereof and the ability to transfer title or ownership of this document's contents are subject to the terms of such license.

Such Licensed Programs and their documentation may contain public open-source software that would be licensed under open-source licenses. Refer to the applicable product release notes for open-source licenses and proprietary notices. Use, duplication, disclosure, and acquisition by the U.S. Government of such Licensed Programs is subject to the terms and definitions of their applicable license.

Disclaimer

Exar reserves the right to make changes to its products, including the contents of this document, or to discontinue any product or service without notice. Exar advises its customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied upon is current. Every effort has been made to keep the information in this document current and accurate as of the date of this document's publication or revision.

Limited Warranty

Exar warrants Products based on the Hifn Technology, including cards, against defects in materials and workmanship for a period of twelve (12) months from the delivery date. Exar's sole liability shall be limited to either, replacing, repairing or issuing credit, at its option, for the Product if it has been paid for. Exar will not be liable under this provision unless: (a) Exar is promptly notified in writing upon discovery of claimed defects by Buyer; (b) The claimed defective Product is returned to Exar, insurance and transportation charges prepaid, by Buyer; (c) The claimed defective Product is received within twelve (12) months from the delivery date; and (d) Exar's examination of the Product discloses to its satisfaction that the alleged defect was not caused by misuse, neglect, improper installation, repair, alteration, accident or other hazard. THIS WARRANTY DOES NOT COVER PRODUCT DAMAGE WHICH RESULTS FROM ACCIDENT, MISUSE, ABUSE, IMPROPER LINE VOLTAGE, FIRE, FLOOD, LIGHTNING OR OTHER ACTS OF GOD OR DAMAGE RESULTING FROM ANY MODIFICATIONS, REPAIRS OR ALTERATIONS PERFORMED OTHER THAN BY EXAR OR EXAR'S AUTHORIZED AGENT OR RESULTING FROM FAILURE TO STRICTLY COMPLY WITH EXAR'S WRITTEN OPERATING AND MAINTENANCE INSTRUCTIONS. BUYER ACKNOWLEDGES THAT THE PRODUCT ARE HIGHLY SENSITIVE ELECTRONIC PRODUCT REQUIRING SPECIAL HANDLING AND THAT THIS WARRANTY DOES NOT APPLY TO IMPROPERLY HANDLED PRODUCT. PRODUCT MANUFACTURED TO MEET BUYER'S SPECIFIC PERFORMANCE SPECIFICATIONS ACCEPTED BY EXAR ARE WARRANTED ONLY TO PERFORM IN CONFORMITY WITH SUCH SPECIFICATIONS, AND ARE WARRANTED ONLY AGAINST DEFECTS NOT RELATED TO SUCH SPECIFICATIONS IN ACCORDANCE WITH THE TERMS AND CONDITIONS SET FORTH HEREIN ABOVE.

Life Support Policy

Exar's Product are not authorized for use as critical components in life support devices or systems. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury or death to human life. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness. Buyer agrees to indemnify, defend and hold Exar harmless for any cost, loss, liability, or expense (including without limitation attorneys' fees and other costs of litigation or threatened litigation) arising out of violation of the above prohibition by Buyer or any person or entity receiving Exar's Product through Buyer.

Patent Infringement - Indemnification

Exar agrees, at its own expense, to defend Buyer from and against any claim, suit or proceeding, and to pay all judgments and costs finally awarded against Buyer by reason of claim, suit or proceeding insofar as it is based upon an allegation that the Product as furnished by Exar infringes any United States letter patent, provided that Exar is notified promptly of such claim in writing and is given authority and full and proper information and assistance (at Exar's expense) for defense of same. In case such Product are finally constituted an infringement and the use of Product is enjoined, Exar shall at its sole discretion and at its own expense: (1) procure for Buyer the right to continue using the Product; (2) replace or modify the same so that it becomes non-infringing; or (3) remove such Product and grant Buyer a credit for the depreciated value of the same.

Buyer shall have the right to employ separate counsel in any claim, suit or proceeding and to participate in the defense thereof, but the fees and expenses of Buyer's counsel shall not be borne by Exar unless: (1) Exar specifically so agrees; or (2) Exar, after written request and without cause, does not assume such defense. Exar shall not be liable to indemnify Buyer for any settlement effected without Exar's written consent, unless Exar failed, after notice and without cause, to defend such claim, suit or proceeding.

The indemnification shall not apply and Buyer shall indemnify Exar and hold it harmless from all liability or expense (including costs of suit and attorney's fees) if the infringement arises from, or is based upon Exar's

compliance with particular requirements of Buyer or Buyer's customer that differ from Exar's standard specifications (Custom Product) for the Product, or modifications or alterations of the Product, or a combination of the Product with other items not furnished or manufactured by Exar.

Buyer agrees that Exar shall not be liable for any collateral, incidental or consequential damages arising out of patent infringement.

The foregoing states the entire liability of Exar for patent infringement.

Motorola

The use of this product in stateful compression protocols (for example, PPP or multi-history applications) with certain configurations may require a license from Motorola. In such cases, a license agreement for the right to use Motorola patents (US05,245,614, US05,130,993) may be obtained directly from Motorola.

Patents

May include one or more of the following United States patents: 4,930,142; 4,996,690; 4,701,745; 5,003,307; 5,016,009; 5,126,739; 5,146,221; 5,414,425; 5,414,850; 5,463,390; 5,506,580; 5,532,694; 6,320,846; 6,816,459; 6,651,099; 6,665,725; 6,771,646; 6,789,116; 6,954,789; 6,839,751; 7,299,282; 7,260,558. Other patents pending.

Trademarks

Hi/fn[®], MeterFlow[®], MeterWorks[®], and LZS[®], are registered trademarks of Exar Corporation. Hifn[™], Hifn Technology, FlowThrough[™], BitWackr, and the Hifn logo are trademarks of Hi/fn, Inc. All other trademarks and trade names are the property of their respective holders.

IBM, IBM Logo, and IBM PowerPC are trademarks of International Business Machines Corporation in the United States, or other countries.

Microsoft, Windows, Windows XP, Windows Vista, Windows Server 2003, Windows Server 2008 and the Windows logo are trademarks of Microsoft Corporation in the United States, and/or other countries.

Exporting

This product may only be exported from the United States in accordance with applicable Export Administration Regulations. Diversion contrary to United States laws is prohibited.

Exar Confidential

If you have signed a Exar Confidential Disclosure Agreement that includes this document as part of its subject matter, please use this document in accordance with the terms of the agreement. If not, please destroy the document.

Table of Contents

List of Figures	10
List of Tables	11
Preface	12
Abbreviations	15
1 Introduction	16
1.1 Software Architecture	16
1.2 Software Modules	17
1.2.1 Operating System	17
1.2.2 Driver Module	18
1.2.2.1 Service Offering Infrastructure (SOI)	18
1.2.2.2 API Layer	18
1.2.2.3 Service Core	19
1.2.2.4 Software Library	19
1.2.3 Applications	19
2 Features	20
2.1 Symmetric Key Algorithms	20
2.1.1 Compression/Decompression	20
2.1.2 Encryption/Decryption	20
2.1.3 Hash/MAC	20
2.2 Public Key (PK) Algorithms	21
2.3 Random Number Generator (RNG)	22
2.4 Support for User Mode 32-bit Applications on a 64-bit OS	22

3	Session Structure	23
3.1	Terminology	23
3.2	Raw Acceleration Sessions	23
3.2.1	Raw Acceleration Session Model	24
3.2.2	Stateful Block Operations	25
3.3	Data Offload Sessions	26
3.3.1	Session Model	26
3.3.2	Segment Chaining	27
3.3.3	Hash Operations	28
4	System Considerations	30
4.1	Application Considerations	30
4.1.1	Synchronous/Asynchronous Mode	30
4.1.2	Kernel/User Mode	30
4.1.3	Single/Multi-Threaded Applications	30
4.1.4	CRC and Metadata	30
4.1.5	Single/Dual Command Ring	31
4.2	Memory Considerations	31
4.2.1	Scatter/Gather Memory Scheme	31
4.2.2	Large Memory Pool Scheme	33
4.2.3	Driver Memory Allocation	33
4.3	API Buffer Requirements	34
4.3.1	Alignment Requirements	34
4.3.2	Expansion Requirements	36
4.4	Performance Considerations	36
5	Driver Module	38
5.1	Initialization Sequence	38
5.2	Driver Configuration File	39
5.2.1	Host Initialization Settings	39
5.2.2	Command Structure Settings	44
5.2.3	Log file Settings	45
5.2.4	Temperature Sensor Settings	46

5.3	Driver Features	47
5.3.1	OS Abstraction Layer.	47
5.3.2	Service Core	47
5.3.2.1	Device manager	47
5.3.2.2	Error Recovery	47
5.3.2.3	Key Manager	47
5.3.2.4	Load Balancer	48
5.3.2.5	Notification Manager	48
5.3.2.6	DMA Manager.	48
5.3.2.7	PK.	48
5.3.2.8	POST.	48
5.3.2.9	Packet Processor.	48
5.3.2.10	Register Access	48
5.3.2.11	RNG	48
5.3.2.12	Session Manager	48
5.3.2.13	User Mode Context Manager	49
6	Operation	50
6.1	Raw Acceleration API Processing Steps.	50
6.1.1	Initialize the SDK	50
6.1.2	Retrieve the Hardware Information	51
6.1.3	Create Keys	51
6.1.3.1	Create Symmetric Keys	51
6.1.3.2	Create Public Keys	52
6.1.3.3	Create a Session.	52
6.1.3.4	Submit Data to the Session	53
6.1.3.5	Submit Data for PK Operation	54
6.1.3.6	Close the Session	55
6.1.4	Destroy Keys	55
6.1.4.1	Destroy Symmetric Keys	55
6.1.4.2	Destroy Public Key	55
6.1.5	Uninitialize the SDK	56
6.1.6	Raw Acceleration Session Data Transform Flow	56
6.2	Data Offload API Processing Steps	59
6.2.1	Initialize the Resources	59
6.2.2	Retrieve the Hardware Information	59

6.2.3	Set the Symmetric Key(s)	60
6.2.4	Establish a Session and Perform a Data Operation	60
6.2.5	Release the Resources.	64
6.2.6	Data Offload Transform Data Flow	64
7	Application Programs	66
7.1	demo Application	66
7.1.1	Initiators	66
7.1.2	CPU Load Calculator	66
7.1.3	Thread Pool	67
7.1.4	RNG	67
7.1.5	DRBG	67
7.1.6	PK Performance	67
7.1.7	Packet Processor Performance	68
7.1.8	demo Configuration File.	70
7.1.8.1	Test Configuration Settings	70
7.1.8.2	Global Configuration Settings	71
7.1.8.3	Raw Acceleration and Data Offload Configuration Settings	72
7.1.8.4	Public Key Configuration Settings	75
7.1.8.5	RNG Configuration Settings	76
7.1.8.6	DRBG Configuration Settings	76
7.2	sdemo Application	77
7.2.1	sdemo Configuration Files	77
7.2.1.1	File Settings	77
7.2.1.2	Transform Settings	77
7.2.2	sdemo Key File	80
7.2.3	sdemo IVAAD File	80
7.3	example Application	81
7.3.1	Raw Session example Application	81
7.3.1.1	Synchronous Mode	81
7.3.1.2	Asynchronous mode	81
7.3.2	Data Offload example Application	82
7.3.2.1	Synchronous mode	82
7.3.2.2	Asynchronous mode	83
7.3.3	PK example Application	83

7.4	Debugging Tools	84
7.4.1	Monitor Tool	84
7.4.2	Status Tool	84
7.4.3	Diagnostic Tool	84
8	Error Handling	86
8.1	Definition of Error Status Codes	86
8.1.1	Error Category	89
8.1.2	Error Flags	90
8.2	Failover	96
8.3	Single Command Error Handling	96
8.3.1	Packet Processing Command Error Handling	96
8.3.2	Public Key Command Error Handling	97
8.4	Hardware Timeout Error Handling	97
8.5	Overheated Condition Error Handling	98
8.6	PCIe Error Handling	99
8.6.1	Register Access Error Detection	102
8.6.2	Handling Uncorrectable PCIe Errors	102
8.7	Data Corruption Error Handling	102
	Appendix A: Usage and Standards Compliance of the 820x Random Number Generator	104
A.1	Overview	104
A.2	Hardware Implementation	104
A.3	Software Implementation	106
A.3.1	Retrieve a Nondeterministic RNG Value from a 820x Device	106
A.3.2	RNG Test	107
A.3.3	Monitoring the RNG Statistics	108
A.4	Standards Requirements and Compliance	109
A.4.1	FIPS Requirements	109
A.4.2	NIST Requirements	109
A.4.3	NIST Compliance	110

A.5	References	110
-----	----------------------	-----

Appendix B: Exported Software Algorithms 111

B.1	eLZS	111
B.1.1	eLZS_Compress()	112
B.1.2	eLZS_Compress_Description()	115
B.1.3	eLZS-Decompress()	116
B.2	DRE_swHashSha256()	118

I Document Revision History 121

List of Figures

Figure 1-1. High Level 820x SDK System Model	16
Figure 1-2. Software Architecture	17
Figure 3-1. Visual Representation of a Raw Acceleration Session	24
Figure 3-2. Raw Acceleration Session Model	25
Figure 3-3. Visual Representation of a Data Offload Session.	26
Figure 3-4. Data Offload Session Model	27
Figure 3-5. Visual Illustration of a Data Offload Hash Session.	29
Figure 4-1. Scatter/Gather Illustration.	32
Figure 6-1. Raw Encode Session Data Operation Flow	57
Figure 6-2. Raw Decode Session Data Operation Flow	58
Figure 6-3. Data Offload Transform Data Flow Diagram	65
Figure 7-1. demo Application Flow Diagram	66
Figure 7-2. PK Performance Flow Diagram	68
Figure 7-3. Packet Processor Performance Flow Diagram	69
Figure 7-4. Operations in Synchronous and Asynchronous Packet Processor Thread	70
Figure 8-1. Single Packet Processing Command Failure Error Processing	96
Figure 8-2. Single Public Key Command Failure Error Processing	97
Figure 8-3. Hardware Timeout Error Processing	98
Figure 8-4. Hardware Overheated Error Processing	99
Figure 8-5. PCIe Error Detection Flow	100
Figure 8-6. PCIe Error Recovery Flow	101
Figure A-1. RNG High Level Block Diagram	105
Figure A-2. Flow to Retrieve RNG Number from a 820x Device	107

List of Tables

Table 4-1. Buffer Requirements of User Mode Data Offload & Raw Acceleration APIs . . .	35
Table 4-2. Buffer Requirements of Kernel Mode Data Offload & Raw Acceleration APIs . .	35
Table 8-1. Error Status Fields	86
Table 8-2. Error Code Field Description	88
Table 8-3. Combined Error Status and Flag Conditions	91
Table B-1. eLZS Compression Flag Definitions	112
Table B-2. Calgary Corpus Performance Test Results.	113
Table B-3. Canterbury Corpus Performance Test Results	114
Table B-4. eLZS Decompression Flag Definitions.	116
Table B-5. swHashSha256 Flag Definitions.	118

Preface

About This Document

Welcome to the Express DX Software Development Kit (SDK) User Guide. This document describes the operation and features of the Express DX SDK version 1.3.0L release.

The Express DX SDK operates with the following Exar devices or cards and their specific drivers:

- DX 1825, 1835, 1845 card (DX 18xx)
- DX 1710, 1720, 1730, 1740 card (DX 17xx)
- 8201, 8202, 8203, 8204 processor

The term "DX card" will be used in this document to refer to any of the cards listed above. Likewise, the term "820x" will be used to refer to any of the processors listed above. If any particular usage is required for a unique platform, it will be specifically noted. Also, please note that in the context of this document, references to the 820x device and the DX card are interchangeable.

Audience

This document is intended for:

- Project managers
- System engineers
- Hardware and software development engineers
- Marketing and product managers

Prerequisite

Before proceeding, you should generally understand:

- Advanced Encryption Standard (AES), Triple Data Encryption Standard (3DES) and their modes of operation
- Cryptographic hash functions
- The functionality of the DX card
- Software and hardware of the target system
- C programming language

Document Organization

This document is organized as follows:

Chapter 1, "Introduction" provides an overview of the Express DX SDK.

Chapter 2, "Features" gives an overview of the operations supported by the Express DX SDK and describes the concept of segment chaining.

Chapter 3, "Session Structure" introduces the concept of a session as it applies to the Express DX SDK.

Chapter 4, "System Considerations" gives important information on application settings, memory allocation and performance tuning.

Chapter 5, "Driver Module" describes the driver initialization sequence and the driver features.

Chapter 6, "Operation" provides a step-by-step example of the data processing on the Exar Express DX card using the Express DX SDK and its API.

Chapter 7, "Application Programs" describes the application programs provided by Exar.

Chapter 8, "Error Handling" describes error handling and the error status codes that are returned by the API functions.

"Appendix A: Usage and Standards Compliance of the 820x Random Number Generator" provides a more detailed description of the 820x hardware RNG functionality and compliance.

"Appendix B: Exported Software Algorithms" documents the exported eLZS and hash software functions.

Related Documents

The following documents can be used as a reference to this document.

Express DX SDK Release Notes, RN-0129

Express DX SDK Getting Started Guide, UG-0208

Data Offload API 2.0 Reference Guide, UG-0213

Raw Acceleration API 1.4 Reference Guide, UG-0189

Express DX SDK 1.3.0L Raw Acceleration API Performance Application Note, AN-0206

Express DX SDK 1.3.0L Data Offload API Performance Application Note, AN-0205

8201, 8202, 8203, 8204 Acceleration Processor Data Sheet, DS-0157

8201, 8202, 8203, 8204 Hardware Design User Guide, UG-0211

DX 17xx Secure Data Reduction Card Data Sheet, DS-0162

DX 1825, 1825, 1845 Secure Data Reduction Card Data Sheet, DS-0164

Express DX 1710, 1720, 1730, 1740 Secure Data Reduction Card & 8201, 8202, 8203, 8204 Acceleration Processor Errata, ER-0015

Customer Support

For technical support about this product, please contact your local Exar sales office, representative, or distributor.

For general information about Exar and Exar products refer to: www.exar.com

Abbreviations

Term	Definition
3DES	Triple DES
AAD	Additional Authenticated Data
AES	Advanced Encryption Standard
API	Application Programming Interface
CBC	Cipher Block Chaining encryption mode
CTR	Counter encryption mode
DES	Data Encryption Standard
DIF	Data Integrity Field
ECB	Electronic Codebook encryption mode
eLZS	Enhanced Lempel-Ziv-Stac Compression
GCM	Galois Counter Mode
HIV	Hash Initialization Vector
HMAC	Hash Message Authentication Code
IV	Initial Vector
LZS	Lempel-Ziv-Stac Compression
OSAL	Operating System Abstraction Layer
PK	Public Key
PP	Packet Processor
RNG	Random Number Generator
SHA	Secure Hash Algorithm
XTS	XEX-based Tweaked CodeBook mode (TCB) with CipherText Stealing (CTS), or XEX-TCB-CTS

1 Introduction

Figure 1-1 shows a high level system model of how the Exar Express DX SDK interfaces to the user application and the Exar 820x device(s).

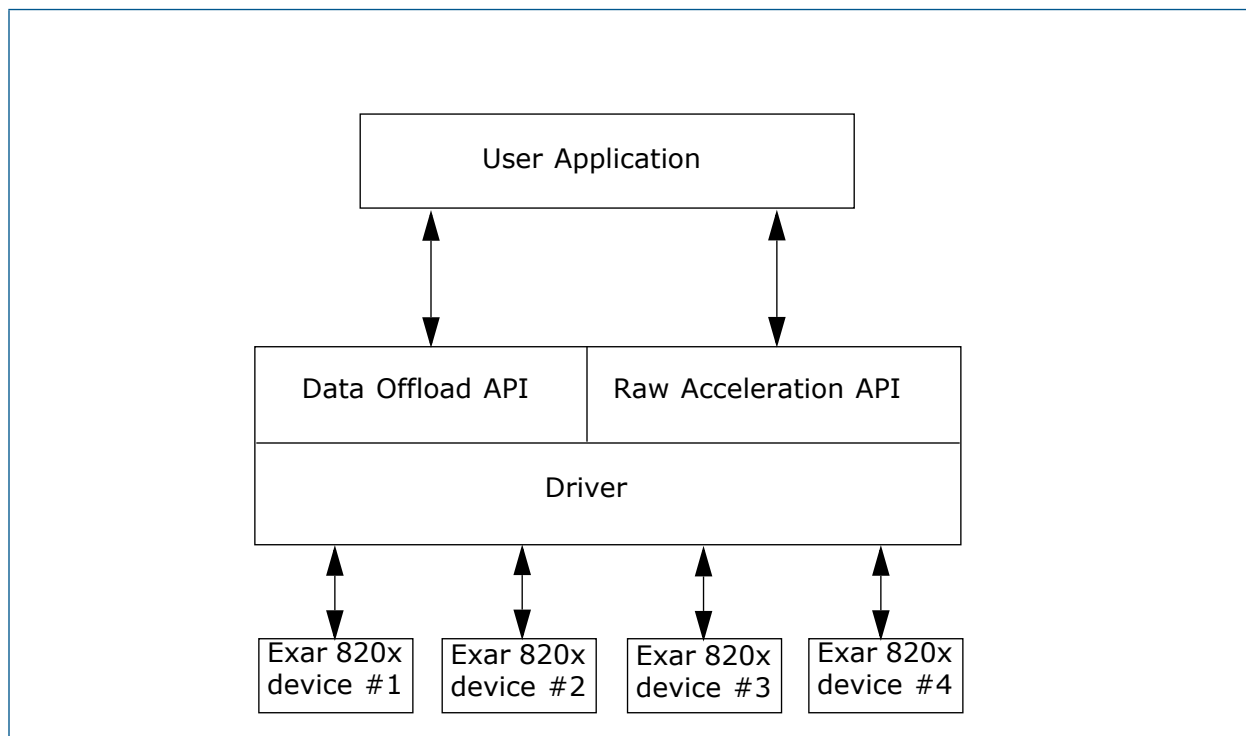


Figure 1-1. High Level 820x SDK System Model

1.1 Software Architecture

Figure 1-2 illustrates the architecture of the Express DX SDK. The software architecture comprises three modules that are each discussed in more detail in the following sections.

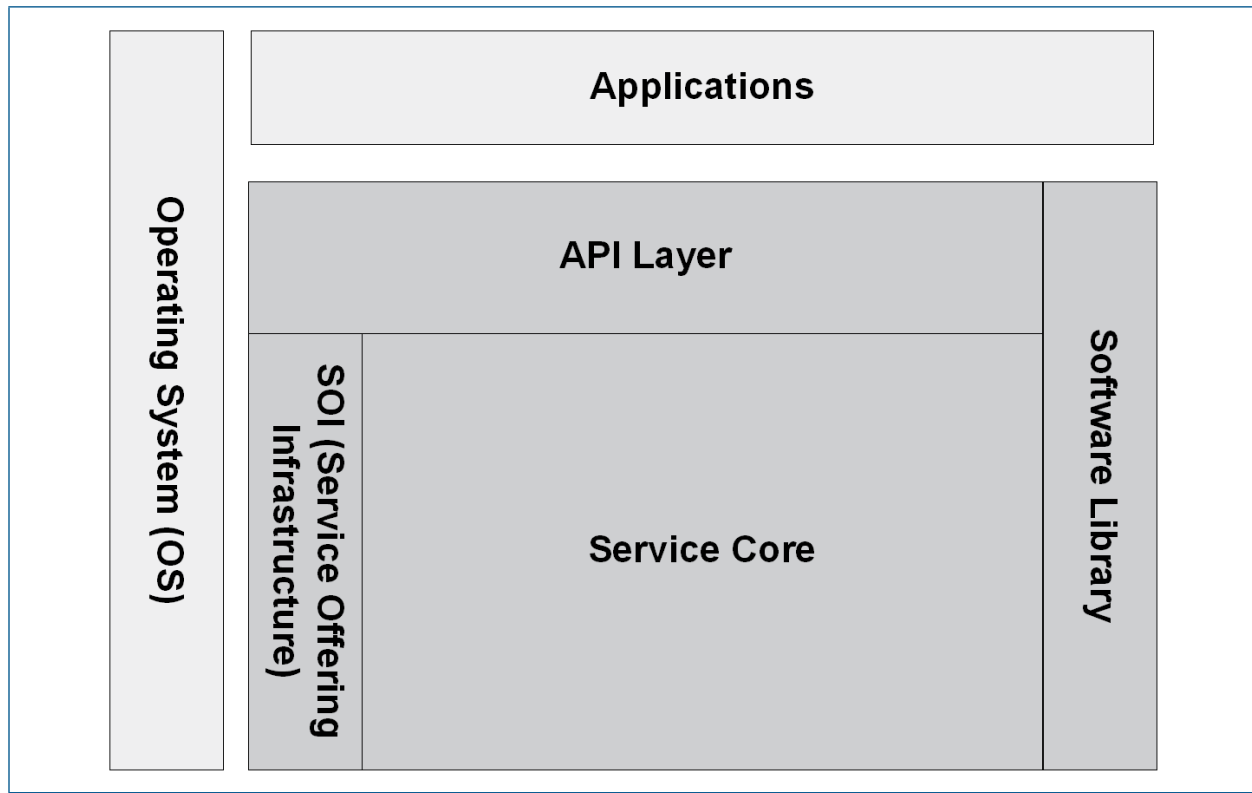


Figure 1-2. Software Architecture

1.2 Software Modules

1.2.1 Operating System

This is the fundamental module of the Express DX SDK. The supported distributions are:

- Red Hat Enterprise Linux 4 Update 6 (2.6.9-67 for i386 32-bit and x86_64 64-bit)
- Red Hat Enterprise Linux 5 Update 1 (2.6.18-53 for i386 32-bit and x86_64 64-bit)
- Red Hat Enterprise Linux 5 (2.6.24.3 for i386 32-bit and x86_64 64-bit)
- Red Hat Enterprise Linux 6 (2.6.32-71 for i386 32 bit and x86_64 64 bit)
- SUSE Linux Enterprise 9 SP 4 (Kernel-bigsmpt-2.6.5-7-308 for i386 32-bit and x86_64 64-bit)
- SUSE Linux Enterprise 10 (Kernel-bigsmpt-2.6.16-21-0.8 for i386 32-bit and x86_64 64-bit)
- SUSE Linux Enterprise 11 (Kernel 2.6.27.19-5 for x86_64 64-bit and i386 32-bit)
- SUSE Linux Enterprise 11 SP1 (Kernel 2.6.32.12-0.7 for x86_64 64-bit and i386 32-bit)

- Linux kernel 2.6.33.2 (RHEL 5 upgrade)
- Real-time Linux kernel 2.6.33 with 2.6.33-rt8 patch (Cent OS5.5 upgrade for x86_64 64-bit)
- Linux kernel 2.6.39 for i386 32-bit and x86_64 64-bit
- CentOS 6.2, kernel 2.6.32-220 for i386 32-bit and x86_64 64-bit
- CentOS 6.3, kernel 2.6.32-279 for i386 32-bit and x86_64 64-bit

The supported hardware platforms are:

- i386 platform
- x86_64 platform

1.2.2 Driver Module

This is the basic driver module for the Express DX SDK that implements all the operations on the 820x device. The Driver module is built on the following sub-modules:

- Service Offering Infrastructure (SOI)
- API Layer
- Service Core
- Software Library

1.2.2.1 Service Offering Infrastructure (SOI)

The Service Offering Infrastructure (SOI) provides the fundamental services for the other modules. It is composed of the OS Abstraction Layer (OSAL), log and file parser components.

OS Abstraction Layer

An OS abstraction layer provides abstracted interfaces for different operating systems. Currently, the OSAL provides several wrappers for Linux APIs. The OSAL APIs can be called by other sub-modules within the SDK.

Log

The Log module allows other modules to log information to a specified log file. The log file is created when loading the driver. The user can edit the driver configuration file to configure the log file's reporting level or the console's print level.

File Parser

This module provides a mechanism to parse the driver configuration file.

1.2.2.2 API Layer

Exar Express DX SDK provides two APIs to interface to the user application.

- Raw Acceleration (Raw) API

The Raw Acceleration API leverages all functionality of Exar 820x device, including compression, encryption, hashing, RNG and PK operations. The Raw Acceleration API should be used for all new designs. Please refer to the *Raw Acceleration API Reference Guide*, UG-0189, for detailed syntax and usage.

- Data Offload (DO) API

The Data Offload API provides compatibility with the Exar Express DR SDK product line for compression, encryption, and decryption operations. The Data Offload API should be used for applications that are migrating from Exar's DR SDK to the DX SDK. Please refer to the *Data Offload API Reference Guide*, UG-0213, for detailed syntax and usage.

1.2.2.3 Service Core

The Service Core module provides the basic operations to manage sessions, keys and other resources to facilitate software simulation and hardware acceleration operations.

The service core retrieves various operations from the API layer, translates those operations into hardware commands, submits the commands to the hardware, retrieves the hardware finished commands and returns those finished operations to the API layer. It also manages the load balancing, session context and key pool.

The Service Core works with the Software Library to provide software support for various operations, such as compression, hash and encryption, and PK operations if the hardware is not available for data operations.

1.2.2.4 Software Library

The Software library will execute the operations, such as compression, hash and encryption and public key operations in software. The Software library may be used to simulate the hardware if an 820x device errored or is recovering from an error, or if there are no operable 820x/DX17xx/DX18xx devices in the system.

1.2.3 Applications

The Express DX SDK includes applications to demonstrate the capabilities of the 820x: demo, example, sdemo, dx_monitor, dx_status, and dx_diag.

The *demo* application can be used to demonstrate the functionality of the Exar 820x device and its API-based system performance. It exercises the Raw Acceleration API, the Data Offload API, or both.

The *example* application can be used as a coding reference for software developers.

The *sdemo* application is a simplified version of the demo application that can be used to quickly verify the encode and decode operations.

The *dx_monitor*, *dx_diag* and *dx_status* applications are debugging tools that can be used to verify the DX card's or 820x processor's status and performance.

2 Features

This section describes the features supported by the 820x SDK.

2.1 Symmetric Key Algorithms

2.1.1 Compression/Decompression

The following compression and decompression operations are supported:

- Stateless LZS
- Stateful LZS decompression
- Stateless eLZS with support for anti-expansion
- Stateless GZIP (Deflate, RFC1951)

2.1.2 Encryption/Decryption

The 820x SDK supports the cipher algorithms listed below:

- Advanced Encryption Standard
 - AES-ECB-128, AES-ECB-192, AES-ECB-256
 - AES-CBC-128, AES-CBC-192, AES-CBC-256
 - AES-CTR-128, AES-CTR-192, AES-CTR-256
 - AES-GCM-128, AES-GCM-192, AES-V-256
 - AES-XTS-512, AES-XTS-256
- 3DES-CBC
- DES (supported via 3DES API with $K1=K2=K3$)
- ARC4 (software only implementation)

Stateful encryption for CBC-mode transforms is supported by storing the current IV in the Host-based session state.

2.1.3 Hash/MAC

A hash of a segment of data returns only one hash value for the entire segment; a hash of the same segment of data at the slice level returns a hash value for each slice.

The hashing algorithms supported are:

- HASH-SHA1, HASH-SHA256
- MD5

- HMAC-MD5
- HMAC-SHA1, HMAC-SHA256
- SSLMAC-MD5
- SSLMAC-SHA1, SSLMAC-SHA256
- AES-GMAC-128, AES-GMAC-192, AES-GMAC-256
- AES-XCBC-128, AES-XCBC-192, AES-XCBC-256

The following hash algorithms are only supported by the software library:

- HASH-SHA384, HMAC-SHA384

The DX SDK supports slice and multi-size hash, dual algorithm hash, and stateful hash.

2.2 Public Key (PK) Algorithms

The Public Key algorithms supported are:

- Diffie-Hellman
 - DH shared secret generation
 - Supports key sizes 512 through 8192 bits in multiples of 64 bits
- Rivest-Shamir-Adleman
 - RSA encryption and decryption
 - RSA sign and verify
 - Supports key sizes 512 through 8192 bits in multiples of 64 bits
 - Supports PKCS #1 v2.1: RSA Cryptography Standard
- Digital Signature Algorithm
 - DSA sign and verify
 - Supports FIPS186-3
 - Supports L and N key length pairs of (512, 160), (768, 160), (1024, 160), (2048, 160), (2048, 224), (2048, 256), (3072, 256), and (3072, 384)
- Elliptic curve Diffie Hellman (ECDH)
 - ECDH public key, shared key generation
 - Supports all EC curves with the form $E/Zp: y^2 = x^3 - 3x + b$
However, the hardware is optimized for FIPS 256, 384, and 521 bit curves.
- Elliptic Curve DSA (ECDSA)
 - ECDSA sign and verify
 - Supports all FIPS curves

However, the hardware is optimized for FIPS 256, 384, and 521 bit curves

- EC Point Verify verifies whether a point is on a specified curve
- EC Point Multiply used to multiply over a specified curve
- Miller-Rabin Primality Test used to determine whether a large number is prime

2.3 Random Number Generator (RNG)

The 820x contains a raw hardware random number generator. The Raw Acceleration API provides methods to retrieve random numbers from the raw hardware RNG and pseudo-random numbers using a Deterministic Random Bit Generator module.

2.4 Support for User Mode 32-bit Applications on a 64-bit OS

Express DX SDK version 1.1.x added support for 32-bit user mode applications running on a 64-bit OS. Refer to the *Express DX SDK Getting Started Guide*, UG-0208, for more information.

3 Session Structure

This section describes the terminology and structure of a Raw Acceleration session and a Data Offload session.

A session is an information exchange between the application and the Express DX SDK. A session is created to submit source data to the API and retrieve transformed data from the service core, and to define the specifics of the data transform. Raw Acceleration and Data Offload sessions have different structures because each API is targeted to specific application environments.

3.1 Terminology

Object	An unstructured data binary large object to be processed.
Unit	A stream of data bytes that is self-contained with respect to the algorithm and the mode of operation.
Session	A session is an information exchange between the application and the Express DX SDK. It is represented by a sequence of data with configurable operations and attributes for those operations. Data Offload and Raw Acceleration sessions are unique.
Service core	Exar's Express DX SDK processing core that can be supported by hardware acceleration.
Block	This term only applies to a Raw Acceleration session. A block is an operational unit which translates source data to destination data for encryption, compression, and hash.
Segment	This term only applies to a Data Offload session. An object can be broken down into a single segment or multiple segments. A segment corresponds to the data to be processed in a command. Examples of segments include a record in a file, a record on tape, or a block on disk.
Slice	This term only applies to a Data Offload session. A segment can be further divided into multiple slices with fixed or variable slice sizes.
Range	This term only applies to a Data Offload session. A contiguous set of segments in a session that form a processing unit for the algorithm of interest.

3.2 Raw Acceleration Sessions

As shown in [Figure 3-1](#), a Raw Acceleration session is self-contained. The algorithm parameters are configured for each packet of data submitted to that session. A Raw Acceleration session may be configured to perform compression, padding, symmetric-key encryption, and/or hash/MAC.

Unlike a Data Offload session, a Raw session will NOT buffer the submitted packets of data. After the packet has been processed, the packet information is no longer available to the SDK.

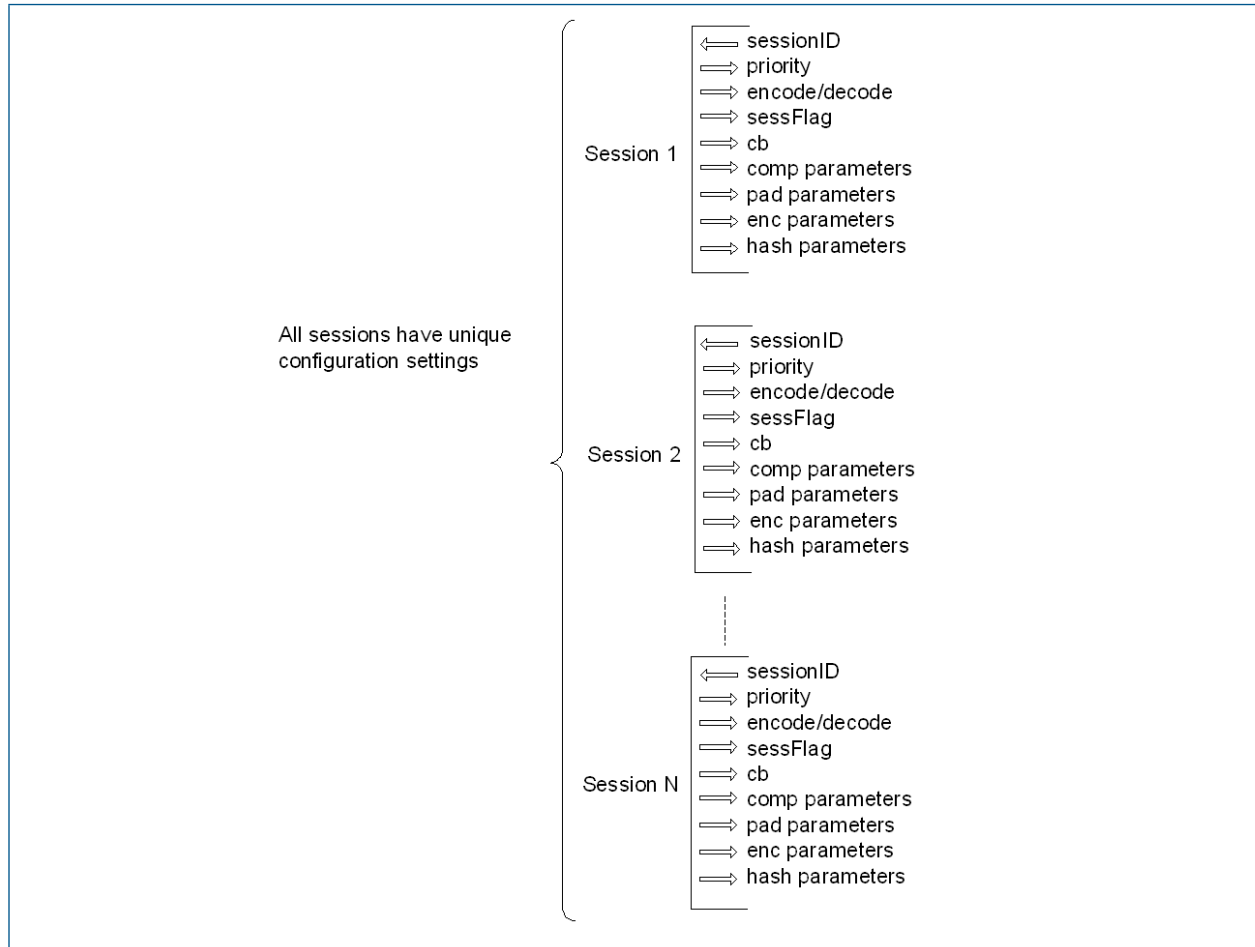


Figure 3-1. Visual Representation of a Raw Acceleration Session

3.2.1 Raw Acceleration Session Model

A Raw Acceleration session is modeled after a block processing operation as illustrated in [Figure 3-2](#). The user submits a block along with the location of the source data and destination buffer to the session to perform a data operation. When the operation completes, the transformed data is written to the destination buffer and returned to the user. Blocks may be submitted synchronously or asynchronously to a Raw Acceleration session.

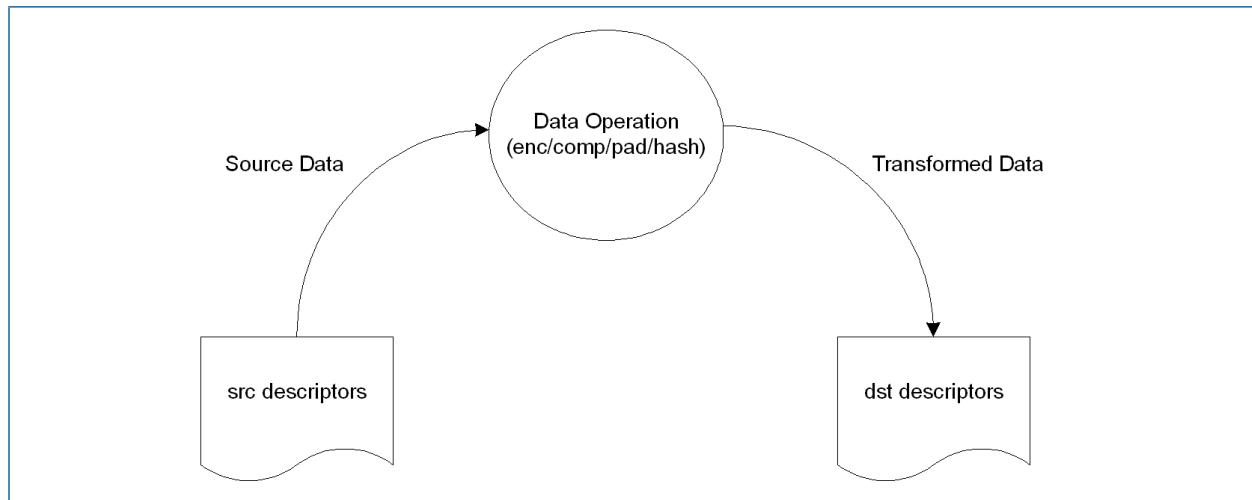


Figure 3-2. Raw Acceleration Session Model

3.2.2 Stateful Block Operations

The Raw Acceleration API allows stateful block-based operations similar to segment chaining for sessions with the Data Offload API. A stateful block operation is achieved by setting the Boolean parameter `stateful` to TRUE in the compression, encryption, or Hash/MAC session management descriptors.

Both synchronous and asynchronous data operation structures contain a flag parameter that must be set to identify the last block in a stateful session.

When in stateful mode, the SDK will automatically save the relevant session information across commands. For stateful LZS decompression, the decompression history is saved. For stateful encryption operations, the initialization vector (IV) is saved across the blocks in a stateful session. For stateful hash operations, an intermediate hash value (IHV) is saved across the blocks.

For example, in a stateful encryption session (CBC mode), the IV is set once by the host when the session is opened. This IV is applied to the very first command for that session, and subsequent commands will use the last CBC block of the previous command for the IV.

3.3 Data Offload Sessions

As shown in [Figure 3-1](#), a Data Offload session contains a set of algorithm parameters that are common to all segments of data submitted for that session. A Data Offload session may be configured to perform compression, padding, symmetric-key encryption, and/or hash/MAC.

Unlike a Raw Acceleration session, submitted packets of data to a Data Offload session are buffered by the driver. After the segment has been processed, the segment information is still available to the SDK.

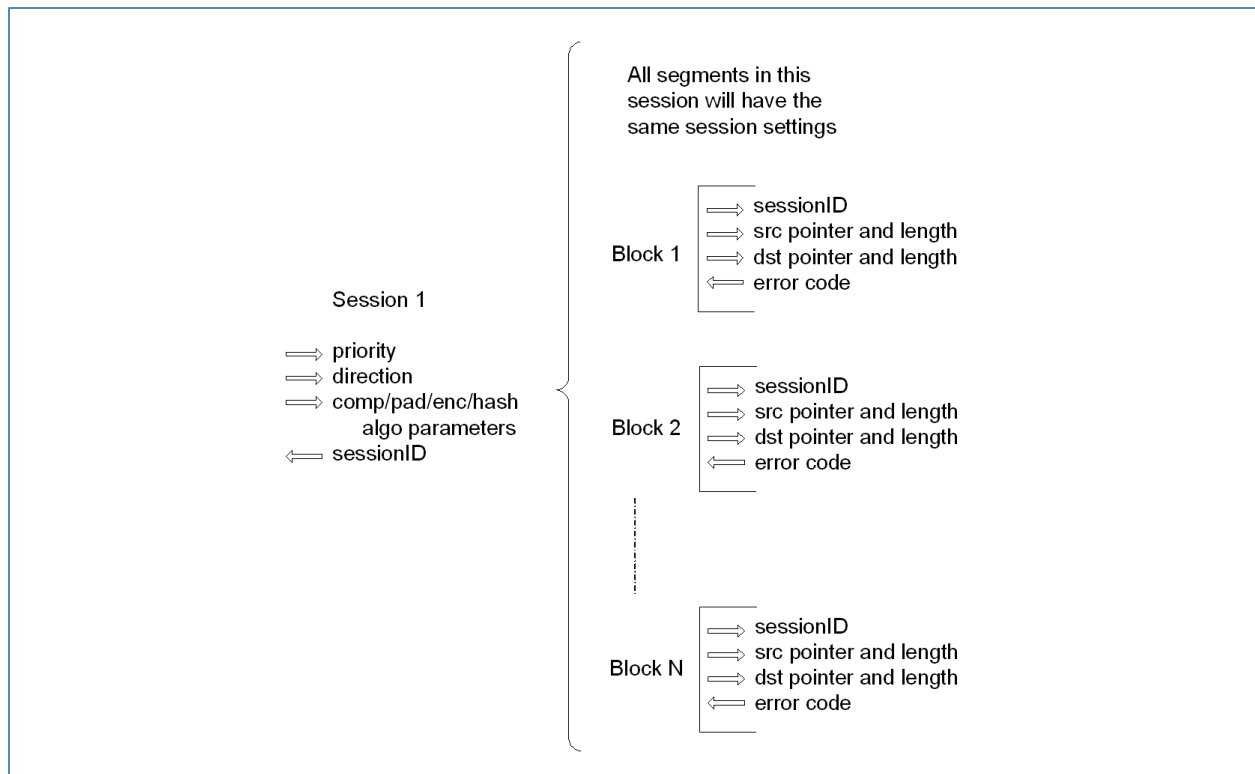


Figure 3-3. Visual Representation of a Data Offload Session

3.3.1 Session Model

A Data Offload session can be thought of as a pipe of variable length. A session "pipe" has a write end and a read end. The application submits data to be transformed to the write end of the session, and retrieves transformed data from the read end. For Data Offload sessions, the submitted data is buffered beyond the end of the read pointer, allowing historical data to be read by using a "rewind" operation to move the read pointer. This extension of the pipe supports single-submittal/multiple-retrieval style operations.

[Figure 3-4](#) compares the same submitted data using two different operations. The first operation illustrates a normal data processing example, with the source data entering at the write end of the pipe and transformed data exiting at the read end of the pipe. The second operation illustrates the same data process that is followed by a rewind operation. The read end pointer is pushed back into the historical data buffer by the rewind operation.

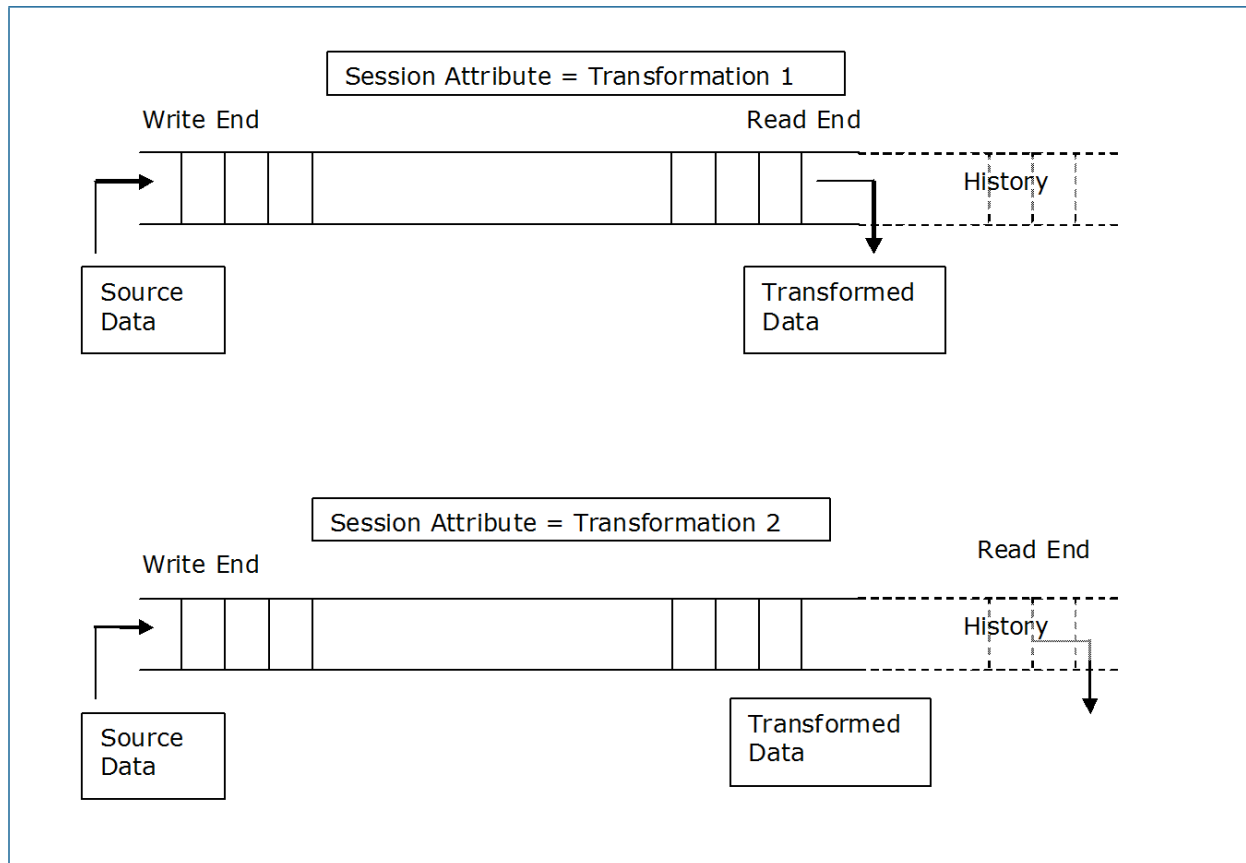


Figure 3-4. Data Offload Session Model

The Data Offload API keeps the user written source data in its session segment chain, so the user application must not release their source buffer until the buffer clean callback function is invoked by the SDK. The parameter `segID` in the Data Operation API function `DRE_doSessWrite()` is automatically incremented after each segment is consumed. The segments associated with the `segID` will not be freed by the SDK. This is not a memory leak, the allocated memory regions are simply occupied. The user application may call the function `DRE_doSessClean()` to free the buffered segments in the allocated memory regions.

3.3.2 Segment Chaining

The Data Offload API allows stateful session-based operations similar to the Raw Acceleration API. A stateful session is achieved by chaining multiple segments together to form a single data unit for the algorithm. When an object is submitted to a session in multiple segments, the segment chaining feature effectively upgrades the segment level operations to object level operations.

For functions that set the compression, hash or encryption attributes of a session, segment chaining is enabled or disabled by specifying the `segChaining` flag.

If segment chaining is disabled, each segment is considered a complete data unit, and as a result, the initialization values applicable to the algorithm are applied to every segment. The application must change the initialization values of the session before submitting a new segment with different initialization values.

If segment chaining is enabled, the applicable initialization values are applied only to the first segment of the chained segments. The chained segments are collectively referred to as a data range. A new range starts when a session is created, or when `DRE_doSessReadLast()`, `DRE_doSessRewind()` or both for that session are called.

If segment chaining is enabled, the application must use the special API function `DRE_doSessReadLast()` to read the last entry in the data range to indicate the last segment in the chain.

3.3.3 Hash Operations

Figure 3-5 illustrates a hash operation session.

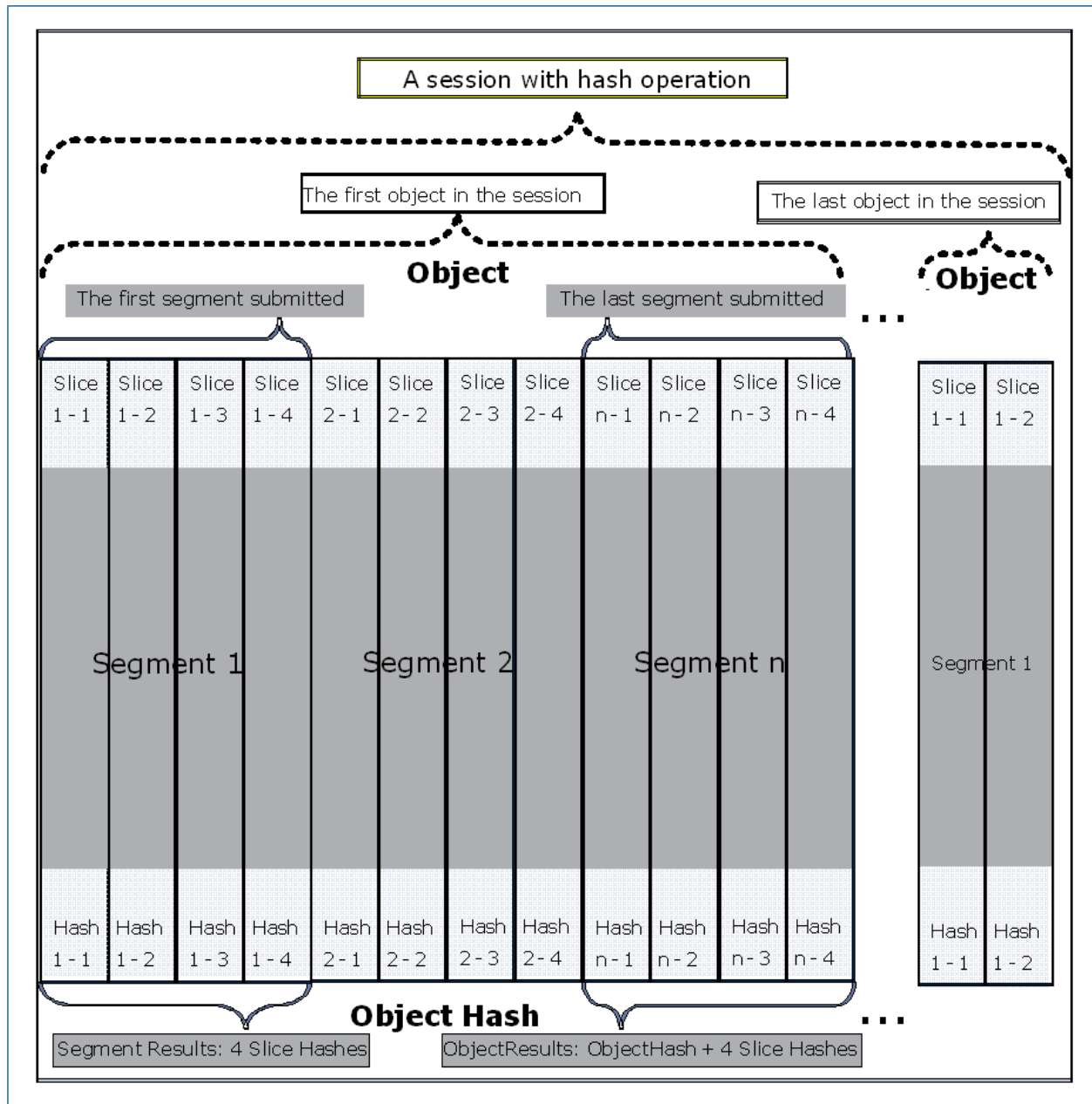


Figure 3-5. Visual Illustration of a Data Offload Hash Session

4 System Considerations

This chapter discusses system issues that should be considered before installing and using the Express DX SDK and its applications.

This chapter uses terms and concepts defined in [Chapter 3, "Session Structure"](#). [Chapter 3](#) should be read prior to this chapter.

4.1 Application Considerations

4.1.1 Synchronous/Asynchronous Mode

For all symmetric key algorithm related operations, the Exar 820x SDK provides both synchronous and asynchronous modes for data processing. In synchronous mode, the API call will not return until the corresponding data processing operations have been completed by the SDK. In asynchronous mode, the API call will return immediately after the data processing operation is accepted by the SDK, and the user will be notified of data processing completion via a callback function.

[Section 4.4](#) describes the performance considerations for synchronous and asynchronous modes.

Please refer to the *Raw Acceleration API Reference Guide*, UG-0189, for a detailed description of the Raw Acceleration API synchronous and asynchronous modes. Please refer to the *Data Offload API Reference Guide*, UG-0213, for a detailed description of the Data Offload API synchronous and asynchronous modes.

4.1.2 Kernel/User Mode

Your system environment will determine if the Express DX SDK will operate in user mode or kernel mode. The Express DX SDK has been designed to link with applications running in either user mode or kernel mode.

4.1.3 Single/Multi-Threaded Applications

Both the Data Offload and Raw Acceleration APIs support single and multi-threaded applications.

The decision to use single or multi-threaded sessions will be based primarily on the system architecture. Multi-threaded sessions will typically yield improved performance.

4.1.4 CRC and Metadata

The metadata feature is only used for backward compatibility with the DR SDK. The DR SDK does not include a CRC when compressing source data, however, the DX SDK does if that feature is enabled. If the DX SDK is used to decompress data that was compressed using the DR SDK, the metadata feature should be enabled in the DX SDK *driver.cfg* file.

4.1.5 Single/Dual Command Ring

Commands for data operation are entered into command rings for the DX SDK to process. The 820x supports a single command ring or dual command rings, however, the DX SDK always utilizes dual command ring mode. In this mode, the SDK maintains two command pointer rings and two result rings in host memory. The SDK will place all high priority commands in one command ring and the low priority commands into the other command ring. Priorities are assigned on a per-session basis, so the priority of a particular command is determined by the session used by that command when it is submitted. The hardware fetches commands from both command rings using a round-robin scheme, so the higher priority commands will, on average, complete sooner than the low priority commands. Note that this priority scheme assumes that there will more low priority commands than high priority commands.

Applications that don't use a priority scheme can simply use a single command ring and specify all sessions as either "high" or "low". There is no hardware performance penalty for using one command ring.

4.2 Memory Considerations

4.2.1 Scatter/Gather Memory Scheme

The 820x and Express DX SDK API support a scatter/gather capability for accessing data in memory. As shown in [Figure 4-1](#), data buffers consist of fragments that are scattered throughout the Host memory area, where each fragment is a contiguous memory region.

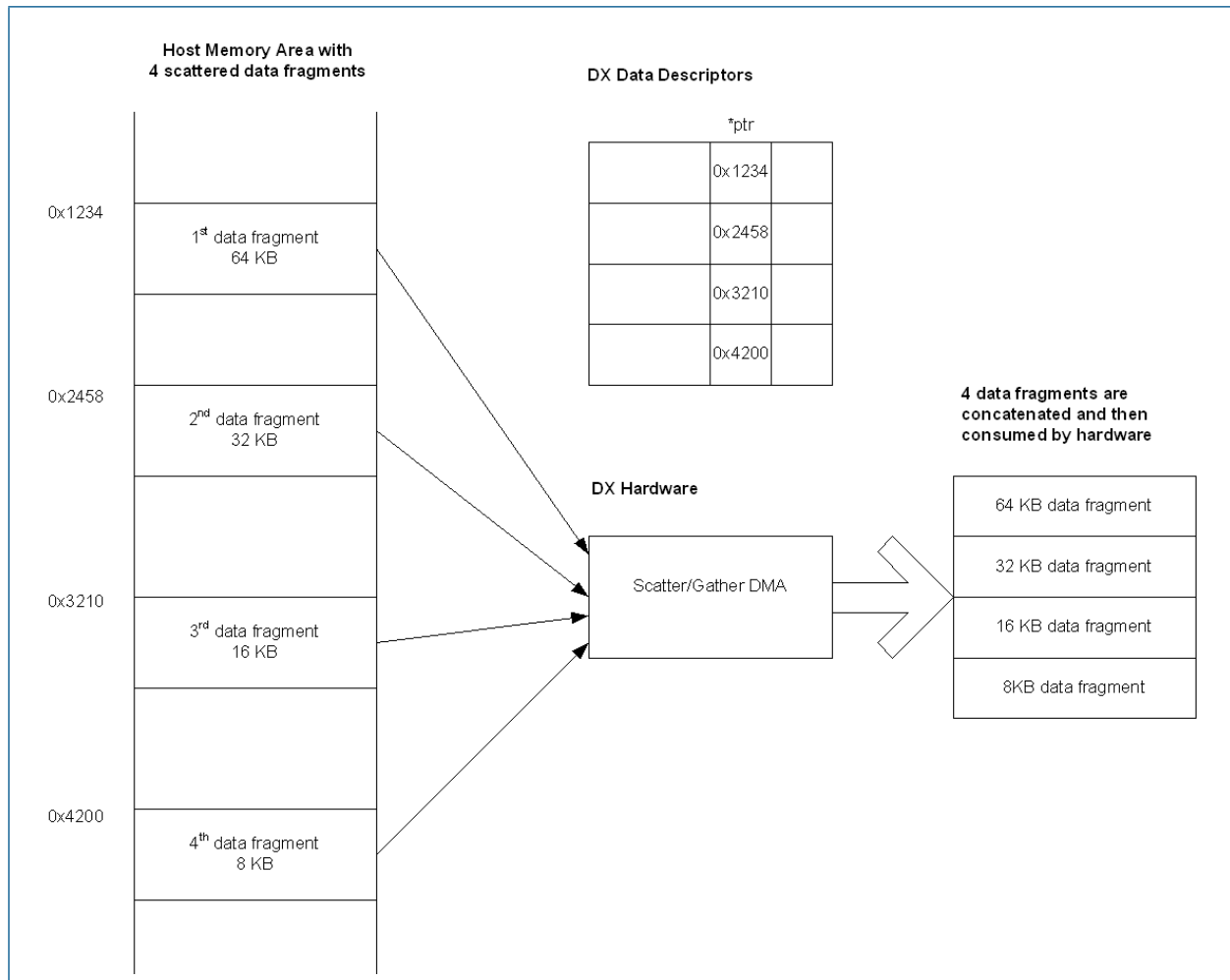


Figure 4-1. Scatter/Gather Illustration

The user submits buffers containing data to process (source buffers) and buffers to hold processed data (destination buffers) via the SDK's various command submission API calls. For user-mode applications, in the typical case where the user does not provide a physical address via the `pAddr` member of the `DRE_DataDesc` structure, the SDK will automatically determine the virtual to physical page mappings for each virtual memory buffer and pass these pages on to the hardware as fragments to process.

On a per-command basis, the 820x DMA controller stores the starting addresses of all the source and destination memory fragments. During a data move operation, the DMA controller pulls the starting address of the first fragment and when that segment of data has completed, it automatically feeds the starting address of the next fragment. The scatter/gather memory scheme does not require a large contiguous block of memory from the operating system.

For both user and kernel mode applications, the total source or destination buffer size (aggregate size of all source or destination fragments) per command must be less than 300MB. Submitted commands whose buffer sizes exceed 300MB may fail due to memory restrictions.

The SDK API provides a set of data descriptors to describe how the data buffers are stored in memory. Users set the beginning address and size of each memory fragment that will be gathered by the hardware DMA controller. The data descriptors themselves are contiguous in the 820x command structure.

Refer to the *820x Data Sheet*, DS-0157, for more details on the command structure and data descriptors.

4.2.2 Large Memory Pool Scheme

Some legacy applications may implement a memory pool to store data in host memory. A memory pool is a large section of physically contiguous memory.

Previous generations of Hifn Technology Look-Aside security/compression devices required such a memory pool for Linux, calling it a “Reserved Memory” scheme. This involved booting the Linux kernel and configuring it with less physical memory than was actually present in the system. The driver would then take the physically contiguous memory just beyond the kernel's region and use it for ring and buffer storage.

With the 820x device family and SDK, this scheme is no longer required. The SDK handles issues related to non-contiguous memory (ie. user-space pages mapped to non-contiguous physical memory) by utilizing the scatter-gather capabilities of the device.

4.2.3 Driver Memory Allocation

The DX SDK enhances performance by using a pre-allocation mechanism. As a result, the user must monitor the total DMA coherent memory available in the system during driver load time, especially when using multiple 820x devices. The driver load may fail if there is not enough DMA coherent memory in the system. The total DMA coherent memory can be calculated using the following formula:

$$\begin{aligned} \text{Memory required for each DMA command (MemCmd)} &= \\ & (2^{\text{ceil}(\log((\text{data_desc_per_cmd} * 32 + 48 + \text{PAGESIZE} - 1)/\text{PAGESIZE}))}) * \text{PAGESIZE} \\ & \text{where } \text{ceil}(x) \text{ means the smallest integer not less than } x \\ \text{Memory required for the metadata of each command:} & \\ & \quad 1\text{K for Raw Acceleration API} \\ & \quad 2\text{K for Data Offload API} \\ \text{Memory required for each card/device (MemCrd)} &= \\ & (\text{MemCmd} + \text{Metadata}) * (\text{cmds_per_ring} + \text{cpu_core_num} * 32) \\ \text{Memory required for the memory pool (MemPool)} &= \\ & 6.4\text{M} + 2^{(\log(\text{data_desc_per_cmd} * 28))} * (64 + 2048) \\ \text{Total memory required} &= \\ & \text{MemCrd} * (\text{number of 820x devices}) + \text{MemPool} \end{aligned}$$

The parameters used in these formulas are defined in the driver configuration file. PAGESIZE is a operating system dependent parameter that sets the size of each physical page in memory.

A rough estimate for the amount of required memory can be obtained using the guideline that each 820x device requires 20-30 MB of memory. Given this rule of thumb, a single DX 1740 card would required 20-30 MB of memory and a DX 1845 card would require 80-120 MB of memory.

4.3 API Buffer Requirements

There are three types of data buffers used in the Exar 820x SDK: source buffers, destination buffers and hash buffers.

The array of source buffers is specified with the parameter `src` in the Raw Acceleration functions `DRE_rawSessSubmitSync()`, `DRE_rawSessSubmitAsync()`, and `DRE_rawSessCmdBundleSubmitAsync()`, or in the Data Offload function `DRE_doSessWrite()`.

The array of destination buffers is specified with the parameter `dst` in the Raw Acceleration functions `DRE_rawSessSubmitSync()`, `DRE_rawSessSubmitAsync()` and `DRE_rawSessCmdBundleSubmitAsync()`, or in the Data Offload functions `DRE_doSessRead()` and `DRE_doSessReadLast()`.

The hash buffer is specified with the parameter `hash` in the Raw Acceleration functions `DRE_rawSessSubmitSync()`, `DRE_rawSessSubmitAsync()` and `DRE_rawSessCmdBundleSubmitAsync()`, or with the parameter `authenticationTag` in the Data Offload function `DRE_doSessWrite()`, with the parameter `hash` in the Data Offload functions `DRE_doSessRead()` and `DRE_doSessReadLast()`.

Results from a Hash operation are stored in the hash buffer. The hash buffer, if needed, is described by the first destination descriptor in the command structure. It must be located in physically contiguous memory, and its length must be 32-byte aligned.

The source buffers and destination buffers support scatter/gather, but the hash buffer does not.

4.3.1 Alignment Requirements

[Table 4-1](#) lists the special buffer requirements for user mode applications. [Table 4-2](#) lists the special buffer requirements for kernel mode applications. Alignment for the destination buffer's address and length is not required for Express DX SDK 1.2.0L and later. However, aligning the buffers as suggested will improve performance.

The 820x hardware requires that the destination buffer's address and length must be 8-byte aligned. If either the address or length is not 8-byte aligned, the DX SDK will reallocate a buffer whose address is 32-byte aligned, and the user specified length is padded to the next 8-byte aligned size. For example, if the user specifies a 50 byte length destination buffer, the SDK will allocate a 56 byte destination buffer whose address is 32-byte aligned.

The 820x hardware requires that the hash buffer's address and length must be 32-byte aligned. If the address alignment is not 32-byte aligned, the DX SDK will reallocate a buffer whose address is 32-byte aligned, and the user specified length is left unchanged. The *Raw Acceleration API Reference Guide*, UG-0189, specifies the hash buffer lengths that must be set by the user for all hash algorithms.

Table 4-1. Buffer Requirements of User Mode Data Offload & Raw Acceleration APIs

Buffer type	Scatter/gather supported?	Address Alignment	Length Alignment	Allocated From
Source	Yes	N/A	N/A	User heap (i.e. malloc) or stack
Destination	Yes	8-byte aligned See Note 1.	8-byte aligned See Note 1.	User heap (i.e. malloc) or stack
Hash	No	32-byte aligned See Note 1.	32-byte aligned See Note 2.	User heap (i.e. malloc) or stack
Notes: 1. Alignment for the destination and hash buffer's address and length is not required for Express DX SDK 1.2.0L and later. However, aligning these buffers as suggested will improve performance. 2. The hash buffer length must be set by the user according to the algorithm. Refer to the <i>Raw Acceleration API Reference Guide</i> , UG-0189, for details.				

Table 4-2. Buffer Requirements of Kernel Mode Data Offload & Raw Acceleration APIs

Buffer type	Scatter/gather supported?	Address Alignment	Length Alignment	Allocated From
Source	Yes	N/A	N/A	kmalloc(), __get_free_pages(), vmalloc(), or kernel stack. See Note 1.
Destination	Yes	8-byte aligned See Note 2.	8-byte aligned See Note 2.	kmalloc(), __get_free_pages(), vmalloc(), or kernel stack. See Note 1.
Hash	No	32-byte aligned See Note 2.	32-byte aligned See Note 3.	kmalloc(), __get_free_pages(), vmalloc(), or kernel stack. See Note 1.
Notes: 1. Buffers allocated using vmalloc() require page table translation and therefore experience degraded performance. This is especially true for small blocks/packets. 2. Alignment for the destination and hash buffer's address and length is not required for Express DX SDK 1.2.0L and later. However, aligning these buffers as suggested will improve performance. 3. The hash buffer length must be set by the user according to the algorithm. Refer to the <i>Raw Acceleration API Reference Guide</i> , UG-0189, for details.				

4.3.2 Expansion Requirements

In addition to the alignment requirements, some conditions may require that the source and destination buffer sizes be larger than anticipated. LZS and eLZS compression may actually cause data expansion if the data is highly random. For example, consider a 1M-byte source data buffer. After compression, the destination data typically becomes smaller than 1Mbyte but may expand to more than 1 Mbyte. If using LZS compression, the maximum number of bytes that should be added to the original source buffer size to accommodate worst case expansion can be evaluated by the following expression:

$$\text{Maximum additional bytes} = 1 / 8 * (\text{source buffer size}) + 16$$

If using eLZS compression, the maximum number of bytes that should be added to the original source buffer size to accommodate worst case expansion can be evaluated by the following expression:

$$\text{Maximum additional bytes} = 4 * (\text{source buffer size}) / 2048 + 4$$

Enabling CRC may also require special buffer size considerations, depending on the algorithm and API being used. When CRC is enabled for a particular encode command, the 820x device will compute and append a four byte CRC to the source data stream. Applications using a block cipher mode of encryption such as AES-CBC must be aware that the data to be fed to the encryption engine must be multiples of the 16 byte AES block size and adjust their buffer sizes appropriately. For example, an application that performs AES-CBC encryption on 4K buffers but does not enable CRC is able to feed in the 4K buffer without modification. Later, when CRC is enabled and the same 4K is passed to the 820x device, an error will be returned because the new plaintext size is now 4K+4 bytes. The source data must therefore be padded with 12 bytes to achieve the proper AES block size granularity. Applications that use the Raw Acceleration API can simply enable and configure the 820x device's pad engine to achieve the proper padding. Applications that use the Data Offload API, which does not expose the pad engine, must pad the data manually. For example, for a 4K source buffer, since the hardware adds 4 bytes for the CRC in the pipeline prior to the encryption engine, the buffer would need to be padded with 12 bytes before being submitted to the hardware so the data to be encrypted by the encryption engine is 16-byte aligned (4096+12+4=4112).

Users should carefully read the appropriate API section in the Release Notes for source/destination buffer restrictions that may require adjustments to the source and destination buffer sizes.

4.4 Performance Considerations

For optimum performance, it is crucial to always keep the command ring(s) fully loaded with commands. In a system where synchronous calls are made to the DX SDK, using multiple threads will achieve the best command processing throughput. In a system where asynchronous calls are made to the DX SDK, then using one or more threads with multiple calls will achieve the best command processing throughput.

In general, due to command processing overheads and user/kernel context switching, small block/packet performance is better in kernel mode than it is in user mode. To alleviate overhead from context switching, user mode applications may utilize the command bundle function introduced in DX SDK version 1.3.0L and the Raw Acceleration API version 1.4.

The driver configuration file parameter settings will affect the system performance. Please refer to [Section 5.2](#) for more information.

5 Driver Module

The Driver module initializes the Exar 820x device and provides services to the applications.

After reading this document, refer to the *Express DX SDK Getting Started Guide*, UG-0208, for instructions on how to compile and install the Driver module.

5.1 Initialization Sequence

The driver performs the following sequence to initialize the Exar 820x device.

1. Initialize the SOI layer.
2. Parse the driver configuration file.
3. Initialize the Service Core Module.
4. Probe for Exar 820x devices. If found, the driver will then perform the remaining steps for each device found in the system. If no devices are found, and the user has enabled software processing ("failover=1") in the driver, all data operations will be sent to the software library for processing. If no 820x devices are found and the user has not configured the driver to use software processing, the driver will still load but data operations will fail.
5. Map the 820x register's memory space. Each probed 820x/DX17xx/DX18xx will be inserted into a card list. Note that an Express DX 1845 card will be seen as four 8204 devices or four DX1740 "cards" from the SDK's point of view. Likewise, an Express DX 1835 card will be seen as three 8204 devices or three DX1740 "cards", and an Express DX 1825 card will be seen as two 8204 devices or two DX1740 "cards".
6. Reset the 820x device.
7. Allocate memory for the command rings and result rings. The number of the rings and the sizes of the rings are defined by the parameters in the driver configuration file.
8. Configure the hardware DMA rings.
9. Load the public key firmware into the hardware if public key operations are enabled (pk_enable = 1 in the driver configuration file).
10. Configure the hardware for PK (if enabled) and RNG operations.
11. Register the interrupt service routine to the OS if necessary.

After the driver is successfully installed, the 820x devices are ready to process commands.

After uninstalling, the previously allocated physical memory will be released to the system. The driver will unregister the interrupt service routine. The hardware cannot provide service until the driver is installed again.

5.2 Driver Configuration File

A driver configuration file is used to configure the hardware related features. The configuration file parameters should be modified to reflect the system before loading the driver.

5.2.1 Host Initialization Settings

This section lists the driver configuration parameters that should be set by the host during initialization. The settings for these parameters would typically not be changed while running data operations.

Note

Changing any of the host initialization settings will require the driver to be reloaded.

notification_mode

The parameter *notification_mode* sets the method that the 820x device should use to notify the host driver of completed packet processing (PP) commands and how the host driver processes the completed commands. (Polling mode is always used for Public Key commands.) The *notification_mode* options are:

- 0 = Interrupt mode using a kernel thread
- 1 = Polling mode
- 2 = Interrupt mode using a workqueue
- 3 = Interrupt mode using a tasklet (DPC)

For options 0, 2 and 3, the 820x device will interrupt the host driver when a command completes. For option 0, a kernel thread is spawned for each 820x device to handle the interrupt and then notifies the user when the command has completed. For option 2, a workqueue is used to handle the interrupt and then notify the user when the command has completed. For option 3, a tasklet is issued to handle the interrupt and then notify the user when the command has completed.

If interrupt mode using a kernel thread is selected (option 0), the SDK uses the Linux New API (also referred to as NAPI) interrupt mechanism to optimize the overall system performance. In the NAPI method, polling is used to mitigate excessive interrupts during heavy processing loads by locking a polling thread to a single CPU.

If polling mode is selected (option 1), the SDK uses a single thread for each 820x device to check if a hardware PP command has completed. This polling thread will check the hardware result ring every 3ms when the SDK is idle, and at a shorter interval when the SDK is busy. The user application must be tolerant of a 3ms latency for the first command in polling mode.

In most cases, interrupt mode provides the best system performance. However, there are some cases in which polling mode may boost performance.

1. If the host computer has many CPUs, a single polling thread can be locked to a single CPU and achieve efficient polling because it cannot be preempted by another thread.
2. If the host computer has a single CPU, using interrupt mode with a user mode API may result in interrupt flooding.

The default value for *notification_mode* is interrupt mode using a kernel thread (option 0).

failover

The parameter *failover* enables or disables the software library. The two options are: enable software library and disable software library.

This parameter applies in the unlikely event of all 820x devices or DX cards failing. If failover is enabled, commands that are submitted to the failed card will continue to be processed in software, albeit at reduced performance. The return status code (bit 24) will be set if the software library is engaged. See [Section 8.2, "Failover"](#) for more information.

The default value for failover is software library disabled for DX SDK versions prior to 1.2.1L, and is enabled for versions 1.2.1L and later.

real_time_verification

The parameter *real_time_verification* enables or disables real time verification of the 820x GZIP, LZS, Hash and encryption engines. The two options are: enable real time verification and disable real time verification. CRC must also be enabled in order for the compression engine to perform real time verification.

The default value for *real_time_verification* is real time verification disabled for DX SDK versions prior to 1.2.1L, and is enabled for versions 1.2.1L and later.

load_balance_algorithm

The parameter *load_balance_algorithm* sets the load balancing algorithm that the SDK will utilize for a system containing multiple cards/devices. The two options are: round-robin and queue depth.

In the round-robin scheme, the SDK sequentially assigns the data operation commands to all devices in the system. When the last device in the sequence is reached, the SDK assigns the next command to the first device and so on. With this load-balancing scheme, no consideration is given to the number of outstanding commands in each device's command ring.

In the queue (command ring) depth scheme, the SDK considers each device's current command queue depth before assigning the next command. For example, if a system has two devices and device A's queue has 10 commands to process, and device B's queue has 20 commands to process, then the SDK will assign the next command to device A.

The round-robin load balancing algorithm will typically yield the best performance if all the cards/devices in the system have the same processing capacity and each command's packet/block size is approximately the same size. In contrast, the queue depth load balancing algorithm will typically yield the best performance if the cards/devices in the system have different processing capacities and/or the mix of commands to process consist of varying packet/block sizes.

Note, however, that due to the extra decision making logic, the queue depth approach has a larger per-packet overhead than round-robin. Exar's own internal testing has shown that small packet (64-byte) performance is better with the round-robin algorithm than with the queue depth algorithm.

Ultimately, the operating environment and application specifics will determine the best load-balancing algorithm to use, and customers are encouraged to experiment with both settings during the performance tuning phase of device integration.

The default value for *load_balance_algorithm* is round-robin.

xts_dif_format

The parameter *xts_dif_format* specifies the XTS_DIF format. The two options are: T10/03-310r0 DIF standard, and T10/08-044r1 SBC-3 DIF standard.

There is no performance impact for the *xts_dif_format* settings.

The default value for *xts_dif_format* is T10/03-310r0.

metadata_enabled

The parameter *metadata_enabled* indicates whether the metadata feature is enabled or disabled. The two options are: enable metadata and disable metadata.

NOTE: This field should only be enabled for ENGINE_CRC sessions using the Data Offload API. Please see [Section 4.1.4](#) for more information about metadata.

The metadata feature should be enabled for backwards compatibility to the DR SDK. There should be no noticeable performance impact for enabling metadata.

The default value for *metadata_enabled* is to disable metadata.

max_key_num

The parameter *max_key_num* sets the maximum number of symmetric and PK keys that may be used by the SDK. The valid values for this parameter are any whole number between 7 and 32M. Note that although the DX SDK supports up to 32M keys, some systems may not support this value.

The SDK dynamically allocates the symmetric and PK keys as entries in a key table. Each entry in the key table points to a key structure, with each entry consuming 4 bytes on a 32-bit platform and 8 bytes on 64-bit platform. There is no performance impact for setting *max_key_num*, however a larger key table requires additional host memory. The maximum size of the symmetric key structure that is used for encryption and MAC operations is 128 bytes.

The default value for *max_key_num* is 4096.

max_session_num

The parameter *max_session_num* sets the maximum number of Raw and Data Offload sessions. The minimum value for this parameter is 1 and the maximum value is 32M.

The SDK dynamically allocates the session numbers as entries in a table. Each entry in the table points to a session structure, with each entry consuming 4 bytes on a 32-bit platform and 8 bytes on 64-bit platform. There is no performance impact for setting *max_session_num*, however a larger table requires additional host memory. The maximum size of a single Raw session structure is 1K bytes and is 2K bytes for a Data Offload session structure. Certain sessions, such as sessions that specify stateful decompression, require additional memory space.

The default value for *max_session_num* is 4096.

pp_statistics_enable

The parameter *pp_statistics_enable* indicates whether to enable or disable statistics gathering. The two options are: enable statistics and disable statistics.

NOTE: Enabling this field will significantly degrade small packet performance on some platforms.

For more information about statistics gathering, please refer to either the *Raw Acceleration API Reference Guide*, UG-0189, or the *Data Offload API Reference Guide*, UG-0213.

The default value for *pp_statistics_enable* is disable statistics gathering.

pp_malloc_mem_threshold

The Express DX SDK uses the *pp_malloc_mem_threshold* parameter to control small packet performance for user-space applications. When the packet size is less than or equal to *pp_malloc_mem_threshold*, the SDK will allocate a buffer in kernel-space and copy in the user-space buffer. Otherwise, the SDK will lock the user-space page(s) and operate on the buffer directly from kernel-space.

Copying the buffer from user-space to kernel-space eliminates the page-lock overhead and increases small packet performance.

Users should tune this parameter to obtain the best results for their particular system environment.

The default value for *pp_malloc_mem_threshold* is 1800.

msi_interrupt_enable

The parameter *msi_interrupt_enable* sets the device interrupt mode. The two options are: MSI interrupt mode, and legacy interrupt mode.

In legacy interrupt mode, the interrupt from the SDK to the host is shared with all other types of interrupts. In MSI mode, the SDK interrupt is not shared. The performance difference between these two modes is negligible.

The default value for *msi_interrupt_enable* is legacy interrupt mode.

pk_enable

The parameter *pk_enable* controls the clock to the Public Key module. The two options are: enable the PK module and disable the PK module.

Enabling the PK module increases power significantly. Applications that do not use the Public Key module should disable this parameter to conserve power.

The default value for *pk_enable* is enabled.

pcie_error_recovery_enable

The parameter *pcie_error_recovery_enable* indicates whether the PCIe error handling feature is enabled or disabled. The two options are: enable PCIe error handling and disable PCIe error handling. Enabling this field will significantly degrade small packet performance, therefore, *pcie_error_recovery_enable* is typically enabled for storage applications and disabled for network applications.

The default value for *pcie_error_recovery_enable* is enabled.

pcie_error_interval_threshold

The parameter *pcie_error_interval_threshold* sets the PCIe error handling interval. If more than one uncorrectable PCIe error occurs within that interval, the DX SDK will take that card offline. To force the DX SDK to attempt to recover every PCIe error, set *pcie_error_interval_threshold* to zero.

The default value for *pcie_error_interval_threshold* is 8 hours.

rng_bit_rate

The parameter *rng_bit_rate* sets the number of clock cycles between bit samples in the serial-to-parallel conversion whitening LFSR inside the 820x RNG engine. Refer to [Appendix A](#) for detailed information about the RNG implementation.

The valid values for *rng_bit_rate* are:

- 0: 125 Mbps (continuous)
- 1: 62.5 Mbps
- 2: 41.7 Mbps
- 3: 31.25 Mbps
- 4: 25 Mbps
- 5: 20.8 Mbps
- 6: 17.9 Mbps
- 7: 15.63 Mbps

The default value for *rng_bit_rate* is 7 (15.63 Mbps). The default value is expected to be suitable for most applications.

rng_sample_interval

The parameter *rng_sample_interval* sets the number of clock cycles between capturing the 32-bit output of the seed generator into the RNG buffer. Refer to [Appendix A](#) for detailed information about the RNG implementation.

The valid values for *rng_sample_interval* are:

- 0: 32 * (rng_bit_rate+1)
- 1: 64 * (rng_bit_rate+1)

- 2: 128 * (rng_bit_rate+1)
- 3: 256 * (rng_bit_rate+1)

The default value for *rng_sample_interval* is 2. The default value is expected to be suitable for most applications.

5.2.2 Command Structure Settings

cmds_per_ring

The parameter *cmds_per_ring* sets the maximum number of commands in a command ring. The SDK uses dual command ring mode, so this parameter sets the maximum number of commands for both command rings (the value will be applied to both rings).

For optimum command processing throughput, the command ring should always be full. Selecting too small of a value for *cmds_per_ring* may limit the performance because the hardware will not run at peak performance unless the command rings are kept full. On the other hand, selecting too large a value for *cmds_per_ring* may unnecessarily consume too much host memory (refer to [Section 4.2.1](#)).

The default setting for *cmds_per_ring* should be satisfactory for most applications. If DRE_C_CMD_NODE_IN_USE or DRE_C_NOT_ENOUGH_RESOURCE errors occur too frequently, it is an indication that the value for *cmds_per_ring* is set too small.

The value of *cmds_per_ring* is not sensitive to large block environments. Typically, this option can be set to 128 for large block environments and above 2K for small block environments. Customers should tune *cmds_per_ring* for their particular application environment.

The valid values for *cmds_per_ring* are:

32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384

The default value for *cmds_per_ring* is 2048.

data_desc_per_cmd

Note

The definition for the parameter *data_desc_per_cmd* was modified for DX SDK versions 1.3.0L and later.

The parameter *data_desc_per_cmd* sets the maximum number of source and destination descriptors for each command structure pre-allocated by the SDK. The valid values for this parameter are any even whole number larger than 32. The value must be even because the hardware requires an equal number of source and destination pairs per command.

The DX SDK and driver will pre-allocate memory for source and destination descriptors up to the size defined by *data_desc_per_cmd*. For commands that require additional source or destination descriptors, the DX SDK will dynamically allocate the required memory.

The setting for *data_desc_per_cmd* and the dynamic memory allocation for larger commands will not affect performance. The default value in the driver configuration file will be suitable for most applications.

Larger values for *data_desc_per_cmd* will allow for more fragments in a scatter-gather scheme, and/or large user mode buffers that will likely be subject to more discontinuous physical page mappings.

The default value for *data_desc_per_cmd* is 64.

5.2.3 Log file Settings

log_file_name

The parameter *log_file_name* sets the path and file name for the log file. The value must be a string type. For example, `"/root/home/abc/d.log"`, uses the absolute file path and sets the log filename to d.log. If this string is empty (`"log_file_name ="`), the log data is appended to the end of the existing or default log file and path.

The default value for *log_file_name* is *dresys.log*. The default path is relative to the current working directory when the driver was loaded. If the driver was loaded using udev, the current working directory will be the root path `"/"`, and as a result, the path and file name will be `"/dresys.log"`.

log_print_level

The parameter *log_print_level* sets the print level for the log output.

There are five defined log output levels:

- 0: DRE_LOG_EMG (least verbose)
- 1: DRE_LOG_ERR
- 2: DRE_LOG_WARNING
- 3: DRE_LOG_INFO
- 4: DRE_LOG_TRACE (most verbose)

The log levels are inclusive of the lower levels. For example, a *log_print_level* value of DRE_LOG_INFO, will direct all DRE_LOG_EMG, DRE_LOG_ERR, DRE_LOG_WARNING and DRE_ERR_INFO messages to be saved to the log file.

The default value for *log_print_level* is DRE_LOG_ERR.

log_redirection

The parameter *log_redirection* indicates where the log information will print. The two options are: print only to the log file, and print to the log file and to the display console.

The default value for *log_redirection* is to print to a log file and to the console.

log_file_size

The parameter *log_file_size* sets the file size for the log file. Entries in the log are written using a circular buffer technique, with the newest log entries wrapping around and overwriting the oldest entries.

Note that the actual log file size may be larger than the value configured for *log_file_size* because the DX SDK adds an additional 10KB to the configured log file size for expansion.

The default value for *log_file_size* is zero which sets the log size to unlimited for backward compatibility.

5.2.4 Temperature Sensor Settings

temp_over_enable

The parameter *temp_over_enable* enables or disables over-temperature protection in the 820x device(s). The two options are: enable over-temperature protection and disable over-temperature protection.

This parameter is used in conjunction with the parameters *normal_temp*, and *over_heat_temp*.

The default value for *temp_over_enable* is enable over-temperature protection.

normal_temp

The parameter *normal_temp* sets the temperature at which the 820x can operate normally. If the 820x temperature exceeds the normal temperature and then returns to a temperature less than or equal to the value for *normal_temp*, the 820x will continue to operate.

This parameter is only valid if the parameter *temp_over_enable* is set to enable. The valid values for *normal_temp* are 0 - 125 °C.

The default value for *normal_temp* is 105 °C.

over_heat_temp

The parameter *over_heat_temp* sets the temperature that if exceeded will cause the 820x device to not operate correctly. If the 820x exceeds this temperature, the 820x will enter into an overheated state and will no longer process commands.

This parameter is only valid if the parameter *temp_over_enable* is set to enable. The valid values for *over_heat_temp* are 0 - 125 °C. The value for *over_heat_temp* must be greater than the value for *normal_temp*.

The default value for *normal_temp* is 115 °C. The default value will be applicable for most system environments.

5.3 Driver Features

This section describes the Driver features in more detail than the general description of the driver components in [Section 1.2.2, "Driver Module"](#).

5.3.1 OS Abstraction Layer

As its name implies, the OS Abstraction Layer (OSAL) defines a generic linkage between the SDK and various OS-specific functionality. The SDK provides an out-of-the-box OSAL implementation specific to the supported Linux distributions. Support for other operating systems requires porting of the OSAL routines.

Examples of the items that require porting to various OS platforms are:

- Mutual Exclusion Semaphore Wrapper
- Virtual/Physical Address Translation APIs
- Delaying Execution APIs
- Memory Operation APIs
- File Operation APIs
- User Mode Thread Operation APIs
- Timer Function Wrapper
- IO access Wrapper
- WorkItem queue Wrapper

5.3.2 Service Core

The Service Core contains several functional modules to assist the API Layer and internal modules.

5.3.2.1 Device manager

The device management handles the initialization of a given device when it is probed. It also serves as a repository for information about the device.

5.3.2.2 Error Recovery

The Error Recovery handles those hardware errors which require the SDK to reset the 820x, i.e. PCIe related errors.

5.3.2.3 Key Manager

The Key Manager provides centralized key storage and index management for both symmetric keys and public keys. The maximum number of keys allowed to be created in the key manager may be configured in the driver configuration file.

5.3.2.4 Load Balancer

The Load Balancer provides the load balancing service for data processing.

5.3.2.5 Notification Manager

The Notification Manager notifies other sub-modules within the SDK when certain operations have been completed by the hardware. It supports both interrupt and polling modes. The user can configure the SDK to use either mode during driver load time using the driver configuration file.

5.3.2.6 DMA Manager

The DMA Manager is responsible for managing the hardware DMA rings.

5.3.2.7 PK

The PK sub-module leverages the 820x PK hardware to provide public key related operations to other modules.

5.3.2.8 POST

The POST sub-module provides a power-on-self-test to test the hardware functionality during initialization.

5.3.2.9 Packet Processor

All the data to be processed by a specific session is represented by a packet within the SDK. The Packet Processor is responsible for managing those packets.

5.3.2.10 Register Access

The Register Access sub-module provides register access services for other sub-modules within the SDK.

5.3.2.11 RNG

The RNG sub-module leverages the hardware to provide users with the hardware RNG values.

5.3.2.12 Session Manager

The Session Manager is responsible for creating, deleting, maintaining, and setting up the various types of sessions within the SDK.

5.3.2.13 User Mode Context Manager

The User mode context manager is responsible for handling operations from user mode applications.

6 Operation

The Express DX SDK package insulates the user application program from the low-level management and use of the 820x device. The Express DX SDK automatically and transparently identifies the 820x hardware and its capabilities. Operations that can execute using Exar's fast hardware compression, decompression, hash, encryption and decryption will do so. If the device or card does not support that function in hardware, the operation will execute in software.

User application software interfaces to the Express DX SDK through either the Raw Acceleration API or the Data Offload API. For detailed syntax and format of the Raw Acceleration and Data Offload APIs, please refer to the *Exar Raw Acceleration API Reference Guide*, UG-0189, or the *Data Offload API Reference Guide*, UG-0213.

This chapter describes the processing steps, transform data flow, and buffer requirements for the Raw Acceleration and Data Offload APIs.

The driver configuration file should be reviewed and modified according to the user's system environment before any data is processed.

6.1 Raw Acceleration API Processing Steps

The basic processing steps to use the Raw Acceleration API are described below.

- Step 1** **Initialize the SDK**
- Step 2** **Retrieve the hardware information**
- Step 3** **Create symmetric or public Keys**
- Step 4** **For symmetric key data operations, create a session**
- Step 5** **For symmetric key data operations, submit data to the session; for PK key operations, submit data to the relevant PK keys for processing**
- Step 6** **For symmetric key data operations, close the session**
- Step 7** **Destroy symmetric or public keys**
- Step 8** **Uninitialize the SDK**

6.1.1 Initialize the SDK

The application should first initialize the resources used by the SDK. To do so, the application should call the API function `DRE_apiSysInit()`. This function should be called only once at the start of every application process. Once initialized, the resources must be released at the exit of the application by calling the API function `DRE_apiSysExit()`.

The pseudo code may look like:

```
status=DRE_apiSysInit();
if(DRE_IS_RESULT_ERR(status))
{
    //error
    ... ..
}
```

6.1.2 Retrieve the Hardware Information

After successfully initializing the SDK, the application should call the API function `DRE_cardInfoGet()` to retrieve the following hardware information:

- Driver version
- Device version
- Card version
- Card model number
- Number of 820x devices in the system
- PCIe Vendor ID
- PCIe Device ID
- PCIe Revision ID
- PCIe Class Code
- Subsystem Device ID
- PCIe Subsystem Vendor ID
- PCIe Link Width
- PCIe bus ID
- Status of the card
- Serial number of the card
- Number of unconfirmed/confirmed hardware errors

The pseudo code may look like:

```
DRE_cardInfo cardInfo[DRE_MAX_CARD_NUM];
DRE_u32b numCards;

status = DRE_cardInfoGet(&numCards, cardInfo);
if (DRE_IS_RESULT_ERR(status))
{
    //error
    .....
}
```

6.1.3 Create Keys

6.1.3.1 Create Symmetric Keys

A symmetric key must be created for any Raw sessions that will use an algorithm that requires an encryption or HMAC key.

To create a symmetric key, the application should call the API function `DRE_symKeySet()`. This function should be called whenever a new symmetric encryption key is required. A key index will be returned to the application that should be used for data operations.

The pseudo code may look like:

```
DRE_u64b keyId;
DRE_u08b keyString[16]=
{
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f
};

status=DRE_rawSymKeySet(keyString, sizeof(keyString), &keyId, 0, 0);
if(DRE_IS_RESULT_ERR(status))
{
    //error
    .....
}
```

6.1.3.2 Create Public Keys

All public key operations except `MODMUL` (modular multiply) and `MODEXP` (modular exponentiation) require that the application create public keys.

To create a public key, the application should call the API function `DRE_pkKeyCreateKey()`. This function should be called whenever a new public key is required. A key index will be returned to the application that should be used for the data operations.

The pseudo code may look like:

```
DRE_u64b keyId;
DRE_pkKey pkKey;

//set RSA public PK key
memset(&pkKey, 0, sizeof(pkKey));
pkKey.type=DRE_PK_RSA_PUB;
pkKey.param.RsaPubParam.m.nLen= 1024/BITS_IN_WORDS;
pkKey.param.RsaPubParam.m.pData=RSA_PUB_KEY_1024_M;
pkKey.param.RsaPubParam.e.nLen = 1;
pkKey.param.RsaPubParam.e.pData = RSA_PUB_KEY_1024_E;

status= DRE_pkKeyCreateKey(&pkKey, &keyId);
if(DRE_IS_RESULT_ERR(status))
{
    //error
    .....
}
```

6.1.3.3 Create a Session

Before any packet of data can be processed, a session must be created.

To create a Raw session, the application should call the API function `DRE_rawSessOpen()`. A session handle will be returned to the user application if the call is successful. The user application will use this returned session handle to process packets and eventually to close the Raw session once the processing is complete.

The pseudo code may look like:

```
DRE_rawSessHandle handle;
DRE_rawSessCompParam comp;
DRE_rawSessPadParam pad;
DRE_rawSessEncParam enc;
DRE_rawSessHashParam hash

//set up the algorithm parameters
... ..

status= DRE_rawSessOpen(&handle,
                        0,
                        DRE_TRUE,
                        0,
                        NULL,
                        &comp,
                        &pad,
                        &enc,
                        &hash);
if (DRE_IS_RESULT_ERR(status))
{
    //error
    .....
}
```

During the Raw session creation, the user application may specify a callback function for that session. If a Raw session is created with a callback function, (i.e. parameter `cb` of `DRE_rawSessOpen()` is NOT NULL), the data submitted to that session must use the asynchronous mode API. If a Raw session is created with a NULL callback function, the application can only call the synchronous mode API for this session. For a detailed discussion of synchronous and asynchronous modes, please refer to [Section 4.1.1](#).

6.1.3.4 Submit Data to the Session

The user application may submit a packet of data to a successfully opened Raw session for algorithm processing. The user application may submit the data in synchronous mode using `DRE_rawSessSubmitSync()`, or in asynchronous mode using `DRE_rawSessSubmitAsync()` or `DRE_rawSessCmdBundleSubmitAsync()`.

In synchronous mode, `DRE_rawSessSubmitSync()` will return only after the SDK/hardware has finished processing the packet of data. In asynchronous mode, `DRE_rawSessSubmitAsync()` will return immediately after the packet of data has been accepted by the SDK, and once the data has finished processing, the callback function specified during the corresponding Raw session creation will be called to notify the user application of the command completion.

In asynchronous mode, `DRE_rawSessCmdBundleSubmitAsync()` wraps multiple commands into a bundle that may then be submitted using a single call. All commands in a bundled session must use the same transform. The results for each command in the bundle are returned individually and may be out of order.

The pseudo code may look like:

```
DRE_rawSyncOpData synOpData;
DRE_dataDesc src, dst;

//set up the opData
.....
//set the src & dst data descriptors
... ..

status=DRE_rawSessSubmitSync(handle,
                             &synOpData,
                             &src, 1
                             &dst, 1
                             NULL);

if(DRE_IS_RESULT_ERR(status))
{
//error
.....
}
```

The data submittal functions will return with the command execution status and if applicable, any error codes. The user application should confirm the status of all returned commands. Refer to ["Error Handling"](#) for more information.

6.1.3.5 Submit Data for PK Operation

The Public Key data should be submitted using the function `DRE_pkExecCmd()`. This function is only supported in asynchronous mode, so the user must specify the callback function during submission. Through the first argument in `DRE_pkExecCmd()`, the PK operation can be directed either to a specific card/device in the system, or the SDK's internal load-balancing algorithm can be used via a special "virtual" card/device identifier.

The pseudo code may look like:

```
Void pk_callback(DRE_u32b cardNum, void *callbackID, DRE_u32 status)
{
    if(DRE_IS_RESULT_ERR(status))
    {
        //error happens
        .....
    }else {
//success
.....
    }
    return;
}

DRE_PKArgs pkArg;
//set up pkArg
```

```
.....
status=DRE_pkExecCmd(0,
                    pk_callback,
                    &pkArg,
                    &pkArg);
if(DRE_IS_RESULT_ERR(status))
{
//error
.....
}
```

The data submittal functions will return with the command execution status and if applicable, any error codes. The user application should confirm the status of all returned commands. Refer to ["Error Handling"](#) for more information.

6.1.3.6 Close the Session

After the user application completes the Raw session, it should be closed using the function `DRE_rawSessClose()` to free the resources used by the session. The user should pass the session handle to `DRE_rawSessClose()` that was returned by `DRE_rawSessOpen()`.

The pseudo code may look like:

```
status=DRE_rawSessClose(handle);
if(DRE_IS_RESULT_ERR(status))
{
//error
.....
}
```

6.1.4 Destroy Keys

6.1.4.1 Destroy Symmetric Keys

The symmetric key created by `DRE_symKeySet()` should be destroyed using the function `DRE_symKeyDestroy()` when the user no longer requires that key.

The pseudo code may look like:

```
status= DRE_symKeyDestroy(keyId);
if(DRE_IS_RESULT_ERR(status))
{
//error
.....
}
```

6.1.4.2 Destroy Public Key

The public key created by `DRE_pkKeyCreateKey()` should be destroyed by the function `DRE_pkKeyDestroy()` when the user no longer requires that key.

The pseudo code may look like:

```
status= DRE_pkKeyDestroy(keyId);  
if(DRE_IS_RESULT_ERR(status))  
{  
  //error  
  .....  
}
```

6.1.5 Uninitialize the SDK

Finally, the user application should release the resources which were allocated during the SDK initialization by calling the function `DRE_apiSysExit()`. This function should be called only once at the end of every application process.

The pseudo code may look like:

```
DRE_apiSysExit();
```

6.1.6 Raw Acceleration Session Data Transform Flow

In a Raw Acceleration session, the four data transforms, namely compression/decompression, padding/de-padding, encryption/decryption, hash (MAC), may be chained.

For encode sessions, the order of data transformation is compression -> padding -> encryption, as shown in [Figure 6-1](#). The hash/MAC position is configurable and may be at any position. If a pure hash (non-MAC) algorithm is required, the hash position must be at the very beginning (see `HASH_BEFORE_CMP (a)` in [Figure 6-1](#)).

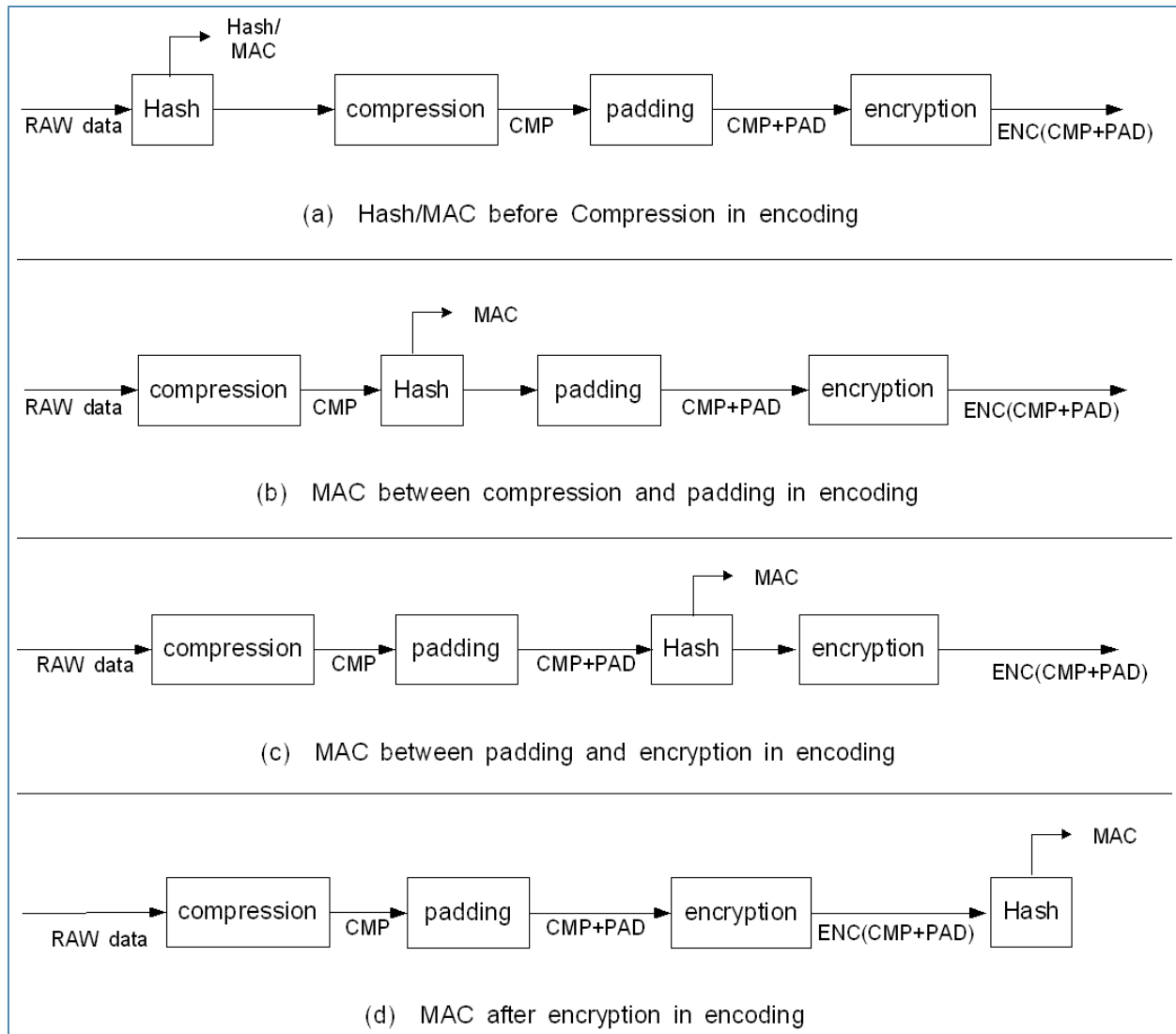


Figure 6-1. Raw Encode Session Data Operation Flow

For decode sessions, the order of data transformation is decryption -> pad stripping -> decompression. The MAC verification position is configurable and may be at any position. Please note that decode sessions do not support pure hash (non-MAC) algorithms.

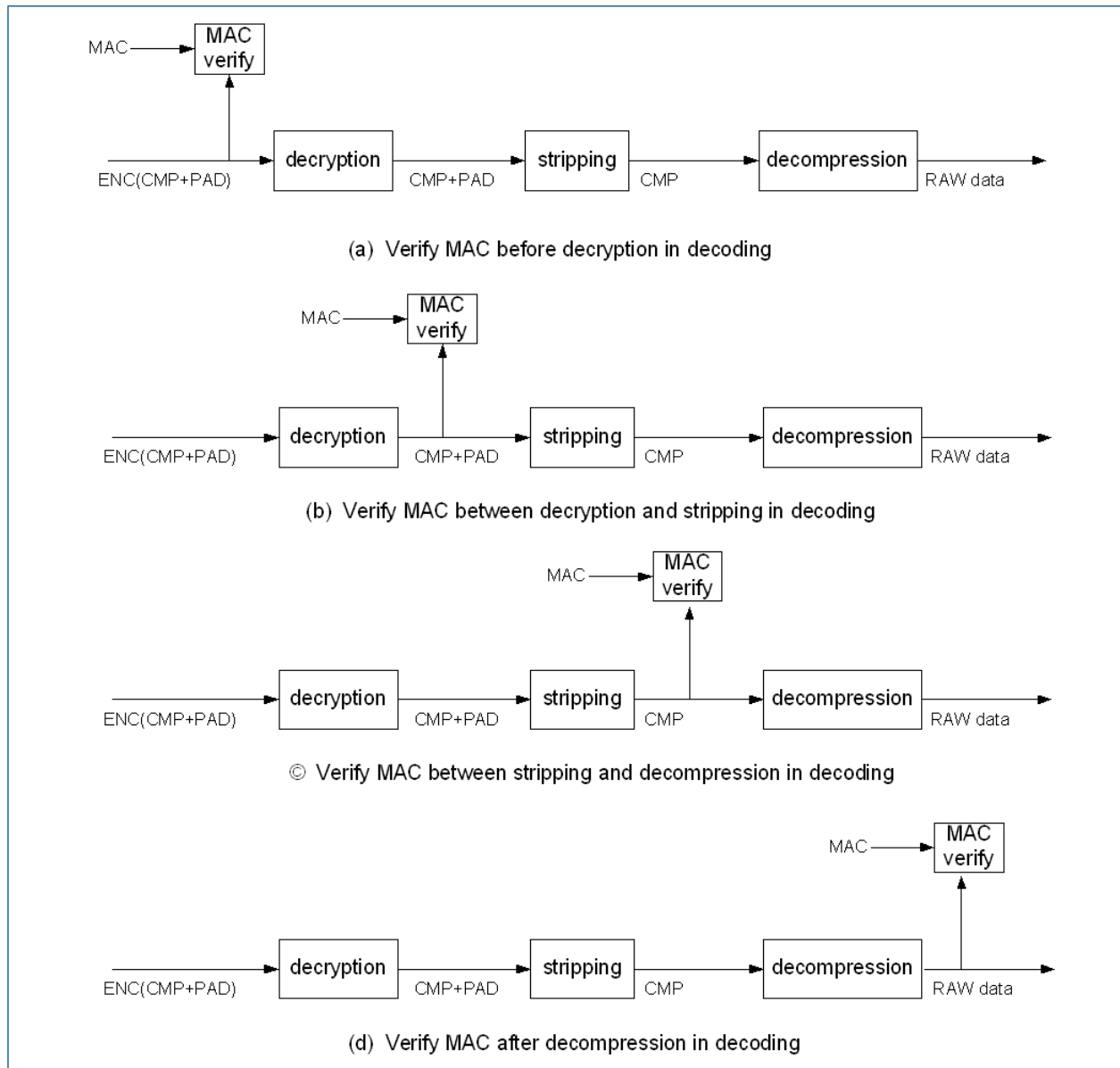


Figure 6-2. Raw Decode Session Data Operation Flow

6.2 Data Offload API Processing Steps

The basic processing steps to use the Data Offload API are described below.

- Step 1** **Initialize the resources**
- Step 2** **Retrieve the hardware information**
- Step 3** **Set the encryption key (if applicable)**
- Step 4** **Establish and operate a session**
- Step 5** **Release the resources**

The following sections provide detailed descriptions of each processing step. Note that the pseudo code examples ignore the return values of the functions.

6.2.1 Initialize the Resources

The application should first initialize the resources used by the SDK. To do so, the application should call the API function `DRE_apiSysInit()`. This function should be called only once at the start of every application process. Once initialized, the resources must be released at the exit of the application by calling the API function `DRE_apiSysExit()`.

The pseudo code may look like:

```
DRE_apiSysInit ();
```

6.2.2 Retrieve the Hardware Information

After initializing the resources, the application should call the API function `DRE_cardInfoGet()` to retrieve the following information from the Exar Express DX cards.

- Driver version
- Device version
- Card version
- Card model number
- Number of 820x devices in the system
- PCIe Vendor ID
- PCIe Device ID
- PCIe Revision ID
- PCIe Class Code
- Subsystem Device ID
- PCIe Subsystem Vendor ID
- PCIe Link Width
- PCIe bus ID

- Status of the card
- Serial number of the card
- Number of unconfirmed/confirmed hardware errors

The pseudo code may look like:

```
DRE_cardInfo cardInfo[DRE_MAX_CARD_NUM];
DRE_s32b numCards = DRE_MAX_CARD_NUM;

...
DRE_cardInfoGet(&numCards, cardInfo);
```

6.2.3 Set the Symmetric Key(s)

If the application will implement a symmetric encryption key, it should first ensure the key is properly set before performing session operations.

To write a key, the application should call the API function `DRE_symKeySet()`. This function should be called whenever a new key is created. A key index will be returned to the application to use in a session operation. The key index must be released when the key is no longer used by calling the API function `DRE_symKeyDestroy()`.

The pseudo code may look like:

```
DRE_u64b aesCbcKeyIndex;
DRE_u08b keyString[] =
{
    0xc9,0x55,0xc3,0xe9,0xc6,0x5d,0x6a,0x64
    ,0x89,0xc3,0xa0,0xa0,0x91,0x7b,0xfb,0xc5
    ,0x0
};

DRE_symKeySet(DRE_AES,
DRE_CBC,
keyString,
strlen(keyString),
& aesCbcKeyIndex,
0,
DRE_FALSE);
```

6.2.4 Establish a Session and Perform a Data Operation

Before executing operations, the application should first create a session. The Session Management API provides functions to create and close a session, as well as set the session operation to compression, decompression, encryption, decryption, or hashing.

To simplify the discussion, an example of an application using these API functions is given below. This example illustrates an initial hash operation with segment chaining enabled, that upon analysis from the application software, is later compressed and encrypted. (Exar's Example application sample code also demonstrates compression/encryption/hash operation with chaining.)

1. The application creates a session by calling `DRE_doSessOpen()`, using the read callback pointer `rcb` and the write callback pointer `wcb` to set the synchronous/asynchronous mode for this session.

The pseudo code to prepare the four segments for a source/destination/hash buffer session operation may look like:

```
DRE_doSessHandle sessHandle;

DRE_dataDesc src[4];
DRE_dataDesc dst[4];
DRE_dataDesc hash[4];
DRE_u32b dstLen[4];
DRE_u32b segId[4];

DRE_u08b iv[] =
{
    0x50, 0xa1, 0xdb, 0x1c, 0x8f, 0x01, 0xa7, 0x7c
    , 0xd2, 0x69, 0x32, 0x70, 0x91, 0x51, 0xc5, 0xd7
};
DRE_modeParam modeParam,
modeParam.cbc.iv=iv;
```

The pseudo code to create a session with asynchronous mode applied may look like:

```
DRE_doSessOpen(&sessHandle, 0, wcb, rcb, doSessBufferFreeCallback);
```

The pseudo code to set the encode mode for the session may look like:

```
DRE_doSessSetEncodeDecode (sessHandle, DRE_TRUE, 0);
```

2. The application sets the session to perform hashing by calling `DRE_doSessSetHash()`. Segment chaining is enabled for this session.

The pseudo code to set the hash operation, enable file and slice hash and segment chaining may look like:

```
DRE_doSessSetHash(sessHandle,
    DRE_SHA1,
    NULL,
    DRE_HASH_NONE,
    NULL,
    -1,
    DRE_TRUE);
```

3. The application submits source data comprised of four segments to the session by calling `DRE_doSessWrite()` four times, and saves all segment IDs returned by `DRE_doSessWrite()`.

The pseudo code to write four segments to the session may look like:

```
DRE_doSessWrite(sessHandle,
    &src[0],
    1,
    NULL,
    &segId[0],
    NULL);
```

```
DRE_doSessWrite(sessHandle,  
                &src[1],  
                1,  
                NULL,  
                &segId[1],  
                NULL);
```

```
DRE_doSessWrite(sessHandle,  
                &src[2],  
                1,  
                NULL,  
                &segId[2],  
                NULL);
```

```
DRE_doSessWrite(sessHandle,  
                &src[3],  
                1,  
                NULL,  
                &segId[3],  
                NULL);
```

4. The application retrieves the hash results of the first two segments by calling DRE_doSessRead() twice. The application calls DRE_doSessReadLast() on the 3rd segment, designating it as the last segment of the chain.

The pseudo code to read the results from segments 0 and 1 using DRE_doSessRead() and for segment 2 using DRE_doSessReadLast() may look like:

```
DRE_doSessRead(sessionHandle,  
               &dst[0],  
               1,  
               &hash[0],  
               &dstLen[0],  
               NULL,  
               NULL);
```

```
DRE_doSessRead(sessionHandle,  
               &dst[1],  
               1,  
               &hash[1],  
               &dstLen[1],  
               NULL,  
               NULL);
```

```
DRE_doSessReadLast(sessionHandle,  
                   &dst[2],  
                   1,  
                   &hash[2],  
                   &dstLen[2],  
                   NULL,  
                   NULL);
```

5. The application examines the hash results, and decides to skip the first two segments but compress and encrypt the remaining data.
6. The application rewinds the session to the third segment by calling `DRE_doSessRewind()`. The first two segments are released by calling `DRE_doSessClean()`.
7. To clear any previous session settings, the application should first call `DRE_doSessSetNone()`. Then the application calls `DRE_doSessSetEncodeDecode()` to set the direction to "encode". Finally, the application calls `DRE_doSessSetComp()` and `DRE_doSessSetEncr()` to set the session for compression and encryption.

The pseudo code to set compression and encryption for the remaining data may look like:

```
DRE_doSessRewind(sessionHandle, segId[2]);

DRE_doSessClean (sessionHandle, segId[1]);

DRE_doSessSetNone (sessionHandle);

DRE_doSessSetEncodeDecode (sessHandle, DRE_TRUE, 0);

DRE_doSessSetComp (sessionHandle,
    DRE_ELZS,
    NULL,
    DRE_COMP_MODE_NONE,
    NULL,
    DRE_FALSE);

DRE_doSessSetEncr (sessionHandle,
    aesCbcKeyIndex,
    NULL,
    modeParam,
    DRE_FALSE);
```

8. The application retrieves the encrypted compression results by calling `DRE_doSessRead()` repeatedly.

```
DRE_doSessRead(sessionHandle,
    &dst[2],
    1,
    NULL,
    &dstLen[2],
    NULL,
    NULL);

DRE_doSessRead(sessionHandle,
    &dst[3],
    1,
    NULL,
    &dstLen[3],
    NULL,
    NULL);
```

```
NULL);
```

9. The application should close the session after all operations are finished.

```
DRE_doSessClose(sessionHandle);
```

6.2.5 Release the Resources

The application should release the resources used by the software at the end of the session. To do so, the application should call the API function `DRE_apiSysExit ()`. This function should be called only once at the end of every application process.

The pseudo code may look like:

```
DRE_apiSysExit ();
```

6.2.6 Data Offload Transform Data Flow

Figure 6-3 illustrates the Data Offload transform data flow. The data flow through the internal hardware engines is operation dependant. For an encode operation, the engine order is compression and then encryption. For a decode operation, the engine order is decryption and then decompression. The hash engine may or may not be used, depending on the storage session operation.

Note: If the user sets the file hash operation to perform MD5, SHA1, or SHA256, this hash operation will be performed on the original uncompressed data.

If the user sets the file hash operation to perform HMAC, the hash operation will be performed on the encrypted data.

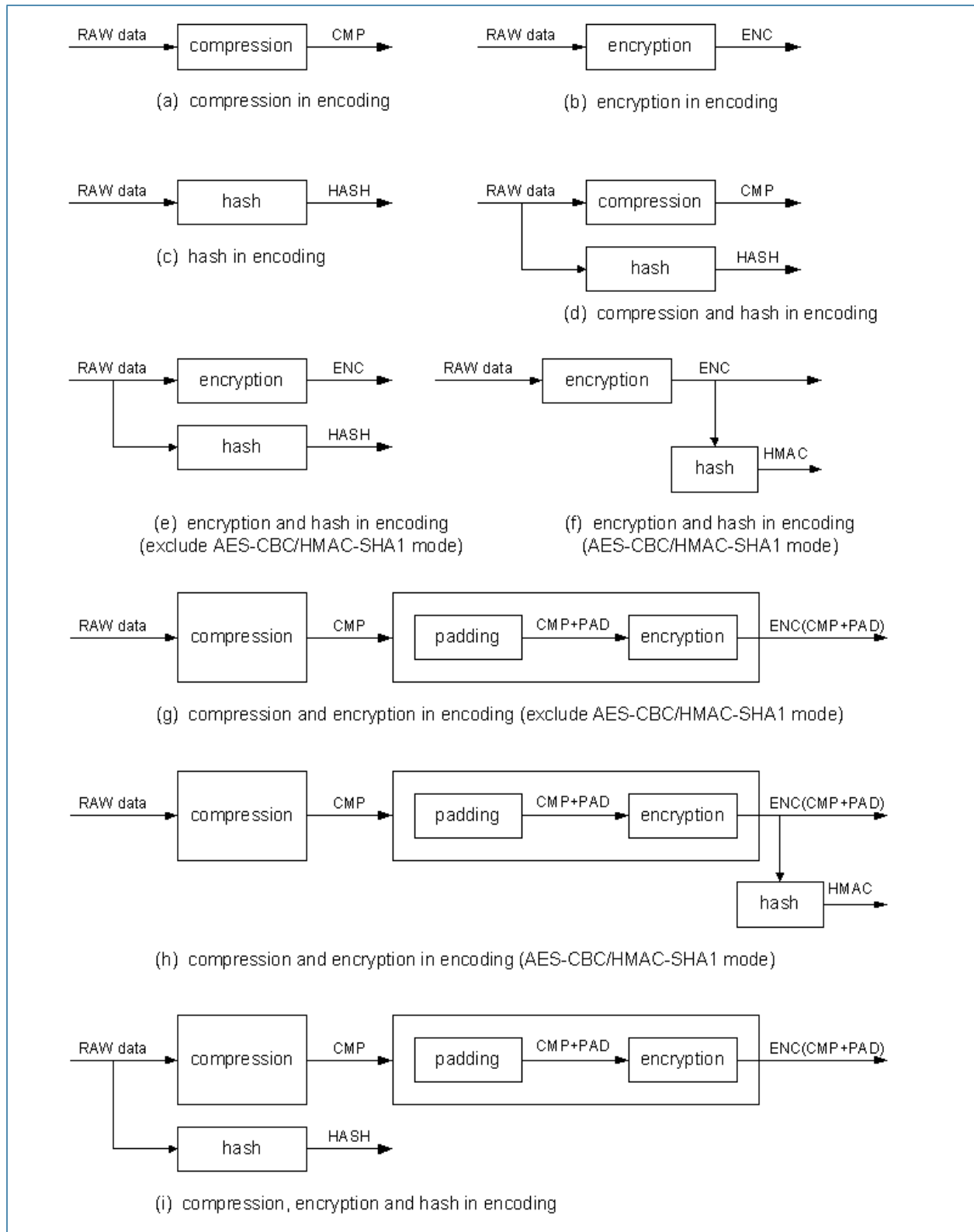


Figure 6-3. Data Offload Transform Data Flow Diagram

7 Application Programs

7.1 demo Application

The demo application performs a simple performance benchmark of the Express DX SDK Raw Acceleration and Data Offload API functions. Both kernel mode and user mode demo applications are provided for Linux.

The demo application can be run from the command line using the parameters defined in the demo configuration file. Configurable parameters include the number of threads to spawn, the operation to execute, and the test mode to be used.

The architecture of the demo application program is shown in [Figure 7-1](#) below.

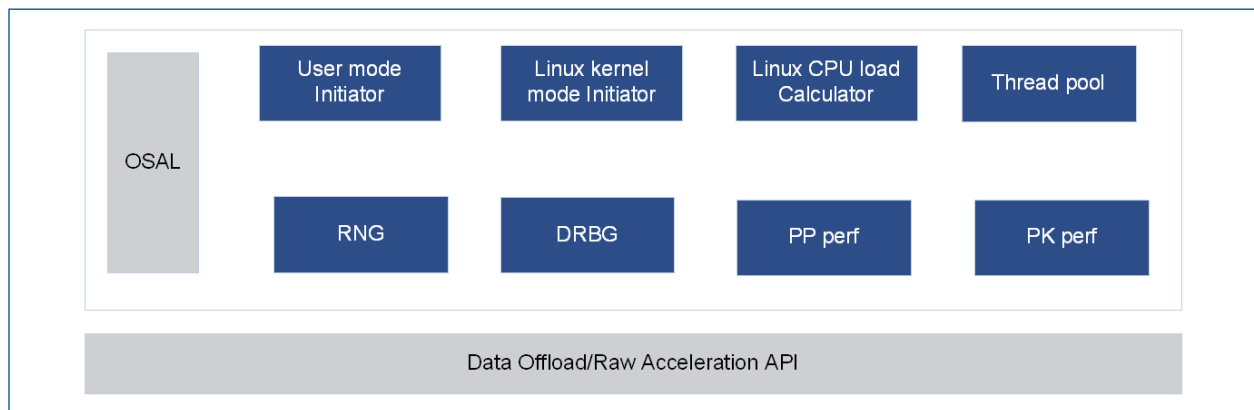


Figure 7-1. demo Application Flow Diagram

7.1.1 Initiators

The user mode and kernel mode Initiators are responsible for:

1. Bringing up the user mode or Linux kernel mode application.
2. Parsing the demo configuration file.
3. Setting up the environment.
4. Running the performance main test functions.

7.1.2 CPU Load Calculator

The CPU load calculator is responsible for getting the CPU load statistics and calculating the average CPU load.

7.1.3 Thread Pool

The thread pool is an internal module used by the Packet Processor performance test. This pool provides the next available thread to the caller to optimize the session creation and destruction overhead.

7.1.4 RNG

The RNG module is responsible for repeatedly reading raw random numbers and making sure no values are the same.

7.1.5 DRBG

The Deterministic Random Bit Generator (DRBG) module is responsible for initializing the DRBG instance(s), reading the DRBG data, and closing the DRBG instance(s).

7.1.6 PK Performance

The PK performance module calculates the performance of the following PK operations:

- DH: 512 bit, 1K bit, 1536 bit, 2K bit, 3K bit, 4K bit, 8K bit
- RSA: 512 bit, 1K bit, 2K bit, 4K bit, 8K bit
- DSA: 512 bit, 768 bit, 1024, 2048, 3072 bit
- ECDH: 192, 224, 256, 384, 521 bit curves
- ECDSA: 192, 224, 256, 384, 521 bit curves

The main flow of the PK perf module is:

1. Create all PK keys as needed.
2. Perform the flow shown in [Figure 7-2](#) for PK operations.
3. Delete all PK keys.
4. Log the performance results into the log file.

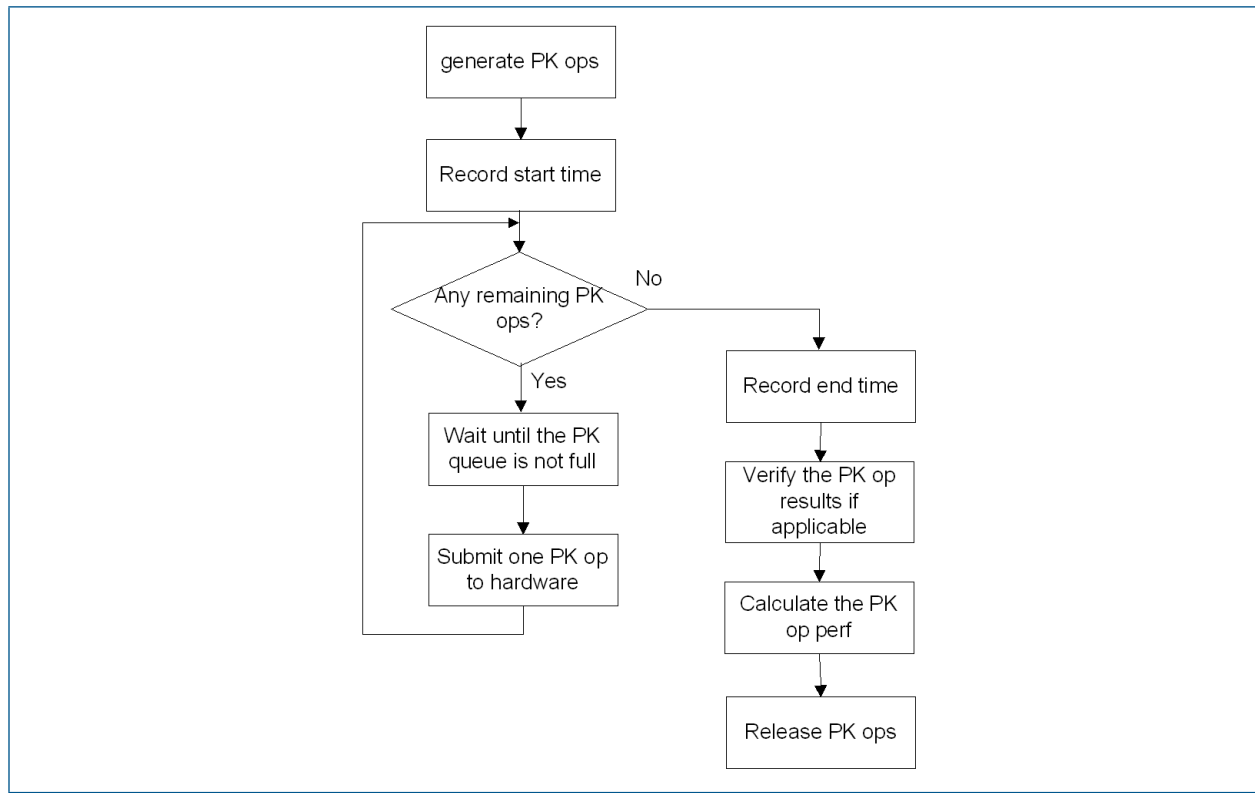


Figure 7-2. PK Performance Flow Diagram

7.1.7 Packet Processor Performance

The Packet Processor (PP) performance module calculates the performance and CPU load for symmetric key algorithm operations.

The main flow of PP performance module is:

1. Create the symmetric keys needed for the PP performance test.
2. Initialize the thread pool.
3. Allocate the temporary buffers.
4. Run the encode/decode performance test of Data Offload/Raw Acceleration session based on demo configuration file, as shown in [Figure 7-3](#).
5. Free the temporary buffers allocated in step 3.
6. Destroy the symmetric keys created in step 1.

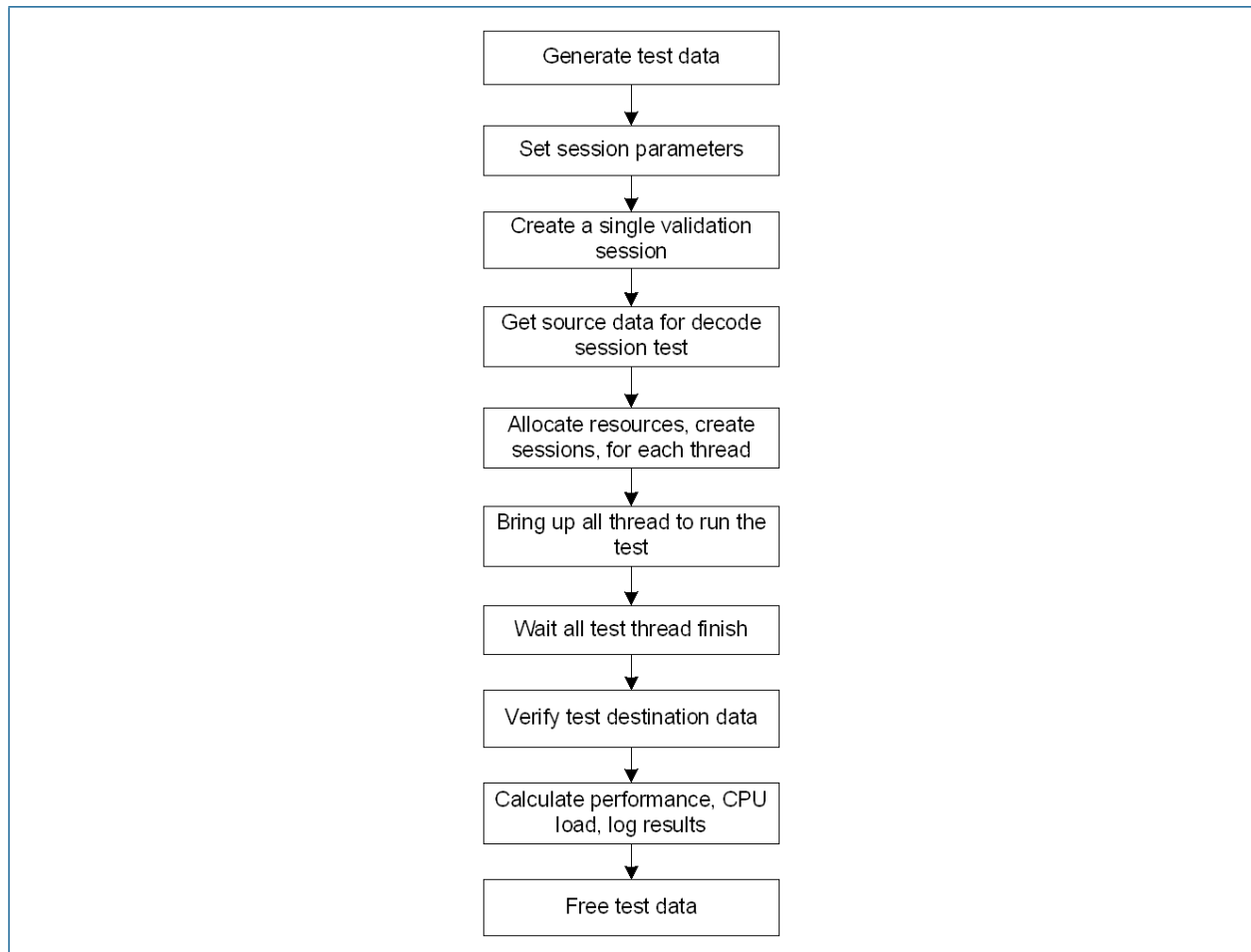


Figure 7-3. Packet Processor Performance Flow Diagram

For each thread in the thread pool, the operations performed in each task are illustrated in [Figure 7-4](#). Note: The operation flow is different for synchronous and asynchronous modes.

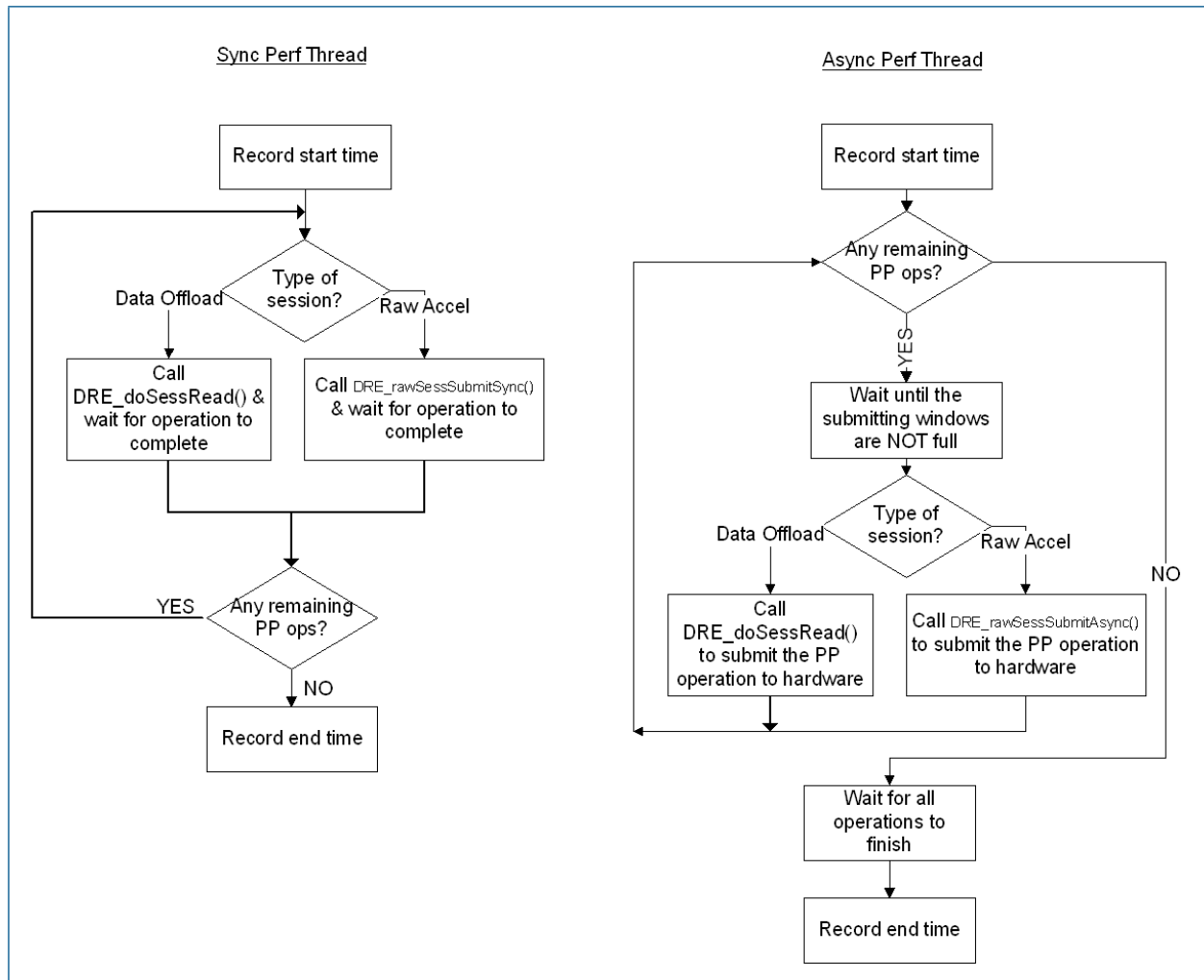


Figure 7-4. Operations in Synchronous and Asynchronous Packet Processor Thread

7.1.8 demo Configuration File

A configuration file, *demo.cfg*, is used to set the test parameters of the demo application. The demo configuration file parameters should be set to appropriate values before running the demo application.

7.1.8.1 Test Configuration Settings

This section lists the configuration parameters that control the demo test mode.

test_raw_pp

The parameter *test_raw_pp* sets the Raw Acceleration test mode. The two options are: test Raw Acceleration session performance and do not test Raw Acceleration session performance.

test_raw_pk

The parameter *test_raw_pk* sets the Raw Acceleration Public Key test mode. The two options are: test Raw Acceleration public key operation performance and do not test Raw Acceleration public key operation performance.

test_raw_rng

The parameter *test_raw_rng* sets the Raw Acceleration Random Number Generation test mode. The two options are: test Raw Acceleration Random Number Generation and do not test Raw Acceleration Random Number Generation.

test_raw_drbg

The parameter *test_raw_drbg* sets the Raw Acceleration API test mode for DRBG. The two options are: test Raw Acceleration API DRBG and do not test Raw Acceleration API DRBG.

test_do

The parameter *test_do* sets the Data Offload test mode. The two options are: test Data Offload session performance and do not test Data Offload session performance.

7.1.8.2 Global Configuration Settings

This section lists the configuration parameters that are relevant to all test modes.

cmd

The parameter *cmd* sets the number of commands to run for each thread of the performance test. The minimum value for *cmd* is 1, and the maximum value is 2^{23} .

The default value for *cmd* is 5000.

max_per_run

The parameter *max_per_run* sets the maximum number of commands for a single run of each thread.

This parameter should be used to limit the amount of commands that execute during a single pass of the demo application if there are too many commands for the amount of allocated memory resources. The demo application will allocate up to *max_per_run* commands for each thread and submit the allocated commands repeatedly, until *cmd* commands have been tested.

If (*cmd* < *max_per_run*) then each thread's allocated command num = *cmd*; # times *cmd* operations submitted = 1.

If (*cmd* ≥ *max_per_run*) then each thread's allocated command num = *max_per_run*; # times *cmd* operations submitted = *cmd*/*max_per_run*.

The default value for *max_per_run* is 100.

thread

The parameter *thread* sets the number of threads to spawn for the Data Offload or Raw Acceleration performance test. This parameter is only valid if *test_raw_pp*, *test_do*, or *test_raw_rng* are enabled.

The minimum value for *thread* is 1, and the maximum value is 8192. The default value for *thread* is 1.

7.1.8.3 Raw Acceleration and Data Offload Configuration Settings

This section lists the configuration parameters that are specific to the Raw Acceleration and Data Offload tests.

async

The parameter *async* specifies whether the Data Offload and Raw Acceleration operations should be run in synchronous or asynchronous mode. This parameter is only valid if *test_raw_pp* or *test_do* are enabled.

The default value for *async* is synchronous mode.

pkt_size

The parameter *pkt_size* specifies the block/packet size for each Data Offload and Raw Acceleration command.

For a Data Offload slice hash session, *pkt_size* must be a multiple of 512 if *slice_hash_algo* is not NONE.

For a Raw Acceleration slice hash session, *pkt_size* must be multiple of 64 if *slice_hash_algo* is not NONE.

In kernel mode, the maximum value for *pkt_size* is 4 Mbytes.

The default value for *pkt_size* is 2048.

src_data_file

The parameter *src_data_file* specifies the source data file name. If left empty, the demo application will generate a regular pattern for the source data (0x00, 0x01, ..., 0xff, 0x00, 0x01, ...).

The default value for *src_data_file* is blank.

en_alg_conf

The parameter *en_alg_conf* specifies whether the demo application automatically selects all transforms for the demo or the user configures a specific transform for the demo.

The default value for *en_alg_conf* is to allow the user to configure the transform.

direction

The parameter *direction* sets whether the Raw Acceleration or Data Offload test is in the encode only direction, decode only direction, or both. This parameter is only valid if *en_alg_conf* is = 1.

The default value for *direction* is both encode and decode direction.

comp_algo

The parameter *comp_algo* sets the Raw Acceleration or Data Offload compression algorithm. This parameter is only valid if *en_alg_conf* is = 1.

The valid values for *comp_algo* are:

- NONE
- LZS
- ELZS
- GZIP
- DEFLATE

The default value for *comp_algo* is ELZS.

enc_algo

The parameter *enc_algo* sets the Raw Acceleration or Data Offload encryption algorithm. This parameter is only valid if *en_alg_conf* is = 1.

The valid values for *enc_algo* are:

- NONE
- AES_CBC
- AES_CTR
- AES_ECB
- AES_GCM
- AES_XTS
- 3DES_CBC
- ARC4 (software only implementation in the Raw Acceleration API)

The default value for *enc_algo* is NONE.

seg_hash_algo

The parameter *seg_hash_algo* sets the Raw Acceleration or Data Offload segment hash algorithm. This parameter is only valid if *en_alg_conf* is = 1.

The valid values for *seg_hash_algo* when using the Data Offload API or the Raw Acceleration API are:

- NONE
- HASH_SHA1
- HASH_SHA256
- HASH_MD5
- HMAC_SHA1
- HMAC_SHA256
- HMAC_MD5

The following values for *seg_hash_algo* are only valid when using the Raw Acceleration API:

- SSLMAC_MD5
- SSLMAC_SHA1
- SSLMAC_SHA256
- AES_GCM_MAC (enc_algo must be set to AES_GCM))
- AES_GMAC
- AES_XCBC
- HMAC_SHA384 (software implementation only)
- HASH_SHA384 (software implementation only)

The default value for *seg_hash_algo* is NONE.

slice_hash_algo

The parameter *slice_hash_algo* sets the Raw Acceleration or Data Offload slice hash algorithm. This parameter is only valid if *en_alg_conf* is = 1.

If *seg_hash_algo* is NOT one of {NONE, HASH_SHA1, HASH_SHA256, HASH_MD5} then *slice_hash_algo* must be set to NONE.

The valid values for *slice_hash_algo* are:

- NONE
- HASH_SHA1
- HASH_SHA256
- HASH_MD5

The default value for *slice_hash_algo* is NONE.

cmd_bundle_num

The parameter *cmd_bundle_num* sets the maximum number of commands that the demo application may submit as a bundle using the Raw Acceleration function `DRE_rawSessCmdBundleSubmitAsync()`. This parameter is only valid in asynchronous mode.

If (*cmd_bundle_num* > *max_per_run*), the *max_per_run* number of commands will be bundled for each `DRE_rawSessCmdBundleSubmitAsync()` call.

The valid values for *cmd_bundle_num* are integers ≥ 1 .

The default value for *cmd_bundle_num* is 1.

7.1.8.4 Public Key Configuration Settings

This section lists the configuration parameters that are relevant to public key operations.

pk_algo

The parameter *pk_algo* specifies the public key algorithm. This parameter is only valid if *en_alg_conf* is = 1.

The valid values for *pk_algo* are:

- DH
- RSA
- DSA
- ECDH
- ECDSA
- ALL

The default value for *pk_algo* is DH.

pk_mbits

The parameter *pk_mbits* specifies the key size in bits for the public key algorithm. This parameter is only valid if *en_alg_conf* is = 1.

The valid values for *pk_mbits* are:

- DH: 512, 1024, 1536, 2048, 3072, 4096, 8192
- RSA: 512, 1024, 2048, 4096, 8192
- DSA: 512, 768, 1024, 2048, 3072
- ECDH: 192, 224, 256, 384, 521
- ECDSA: 192, 224, 256, 384, 521

The default value for *pk_mbits* is 256 for ECDH/ECDSA, and 512 for all others.

7.1.8.5 RNG Configuration Settings

This section lists the configuration parameter that is relevant to random number generation operations.

rng_pkt_size

The parameter *rng_pkt_size* specifies the random number generator packet size for a Raw Acceleration session. The maximum value for this parameter is 8k.

The default value for *rng_pkt_size* is 128.

7.1.8.6 DRBG Configuration Settings

This section lists the configuration parameters that are relevant to deterministic random bit generation operations.

drbg_algo

The parameter *drbg_algo* specifies the underlying cryptography algorithm used by the DRBG. The valid values are:

- AES_CTR
- DUAL_EC

The default value for *drbg_algo* is AES_CTR.

drbg_bits_per_request

The parameter *drbg_bits_per_request* specifies the number of pseudo-random bits to be returned from the DRBG generate function. The value for *drbg_bits_per_request* must be less than or equal to DRE_DRBG_MAX_NUM_BITS_PER_REQ, which is defined in *dre_config.h*.

The default value for *drbg_bits_per_request* is 1024 bits.

7.2 sdemo Application

The sdemo application is a simplified version of the demo application. The sdemo application parses the configuration files, reads the data from the specified source file, performs the specified operation, and then outputs the result to the specified destination file. The source data is processed in a single command.

7.2.1 sdemo Configuration Files

Two configuration files, *sdemo.encode.cfg* and *sdemo.decode.cfg*, are used to set the test parameters of the sdemo application. The configuration file parameters should be set to appropriate values before running the sdemo application.

7.2.1.1 File Settings

src_data_file

The parameter *src_data_file* specifies the source data file name. The source data file name cannot be left blank, otherwise an error will occur. The directory path to the source data file may be relative or absolute.

If the configuration file is set for a decode MAC operation, the last MAC_SIZE specified by the MAC algorithm will be treated as the MAC data.

The default value for *src_data_file* is *src_data*.

dst_data_file

The parameter *src_data_file* specifies the destination data file name. The sdemo application will create the destination data file if it does not exist, and will overwrite an existing file of the same name. The directory path to the destination data file may be relative or absolute.

The output result is written directly to destination file in binary mode; no header precedes the data.

The default value for *dst_data_file* is *dst_data*.

7.2.1.2 Transform Settings

direction

The parameter *direction* specifies whether the intended operation is an encode or decode operation.

The valid values for *direction* are:

- 1 = encode
- 0 = decode

The default value for *direction* is encode.

algo

The parameter *algo* specifies the general algorithm type for the sdemo test. The exact algorithm is specified by the parameters *comp_algo*, *enc_algo*, and *seg_hash_algo*, which are defined below.

The valid values for *algo* are:

- COMP
- ENC
- SEG_HASH
- PASS_THRU

The default value for *algo* is COMP.

comp_algo

The parameter *comp_algo* specifies the compression algorithm for the sdemo test. This parameter is only valid if *algo* is = COMP.

The valid values for *comp_algo* are:

- LZS
- ELZS
- GZIP
- DEFLATE

The default value for *comp_algo* is ELZS.

comp_mode

The parameter *comp_mode* specifies the Huffman coding for the GZIP and DEFLATE compression algorithms. This parameter is only meaningful when *algo* = COMP, and *comp_algo* = GZIP or DEFLATE.

The valid values for *comp_mode* are:

- STORE
- STATIC
- DYNAMIC

The default value for *comp_mode* is DYNAMIC.

enc_algo

The parameter *enc_algo* specifies the encryption algorithm for the sdemo test. This parameter is only valid if *algo* is = ENC.

The valid values for *enc_algo* are:

- AES_CBC_128, AES_CBC_192, AES_CBC_256
- AES_CTR_128, AES_CTR_192, AES_CTR_256
- AES_ECB_128, AES_ECB_192, AES_ECB_256
- AES_GCM_128, AES_GCM_192, AES_GCM_256
- AES_XTS_256, AES_XTS_512
- 3DES_CBC
- ARC4 (software only implementation)

The default value for *enc_algo* is AES_CBC_128.

seg_hash_algo

The parameter *seg_hash_algo* specifies the segment hash algorithm for the *sdemo* test. This parameter is only valid if *algo* is = SEG_HASH.

The valid values for *seg_hash_algo* are:

- HASH_SHA1, HASH_SHA256, HASH_SHA384 (software implementation only), HASH_MD5
- HMAC_SHA1, HMAC_SHA256, HMAC_SHA384 (software implementation only), HMAC_MD5
- SSLMAC_SHA1, SSLMAC_SHA256, SSLMAC_MD5
- AES_GMAC_128, AES_GMAC_192, AES_GMAC_256
- AES_XCBC

The default value for *seg_hash_algo* is HASH_SHA1.

crc_config

The parameter *crc_config* specifies whether CRC is enabled or disabled.

The valid values for *crc_config* are:

- CRC Disabled
- Host CRC Enabled
- Engine CRC Enabled

The default value for *crc_config* is CRC Disabled.

cmd_target

The parameter *cmd_target* specifies whether commands are sent to a specific 820x device, sent to the software library, or load balanced.

The valid values for *cmd_target* are:

- 0-31: Commands sent to a specific 820x device number (the max number of devices supported by the SDK is 32)
- 32: All commands sent to software library
- 33: All commands load balanced

The default value for *cmd_target* is Load Balanced.

7.2.2 sdemo Key File

A key file, *sdemo.key*, is used to define the keys for the encryption and MAC algorithms supported by the sdemo application.

- AES_CBC_128, AES_CBC_192, AES_CBC_256
- AES_CTR_128, AES_CTR_192, AES_CTR_256
- AES_ECB_128, AES_ECB_192, AES_ECB_256
- AES_GMAC_128, AES_GMAC_192, AES_GMAC_256
- AES_XTS_256, AES_XTS_512
- 3DES_CBC
- ARC4
- HMAC_MD5, HMAC_SHA1, HMAC_SHA256, HMAC_SHA384

The key file parameters should be set to appropriate values before running the sdemo application. In the sample key file, there is one key defined for each algorithm. To edit a single key value, replace the entry for the specified algorithm with a new key (in hex format); the remaining keys may be left unchanged.

7.2.3 sdemo IVAAD File

A IV and AAD file, *ivaad.key*, is used to define the IV and AAD data for those algorithms supported by the sdemo application that require this information.

- AES_CBC_128, AES_CBC_192, AES_CBC_256
- AES_CTR_128, AES_CTR_192, AES_CTR_256
- AES_ECB_128, AES_ECB_192, AES_ECB_256
- AES_GCM_128, AES_GCM_192, AES_GCM_256
- AES_XTS_256, AES_XTS_512
- 3DES_CBC

Note that the AAD input for an AES_GMAC_128, 192, or 256 bit key is in the source data file, not in the AAD segment of the *ivaad.cfg* file, while the AAD input for an AES_GCM_128, 192, or 256 bit key is in the AAD segment of the *ivaad.cfg* file.

7.3 example Application

The example application provides a sample reference to demonstrate using the Express DX SDK APIs for data operations such as passthrough, compression/decompression, encryption/decryption and hash.

7.3.1 Raw Session example Application

This section describes the basic flow for submitting a packet of data for processing in a Raw session in both synchronous and asynchronous mode.

7.3.1.1 Synchronous Mode

1. Call `DRE_symKeySet()` to create the symmetric keys if necessary.
2. Set the parameters for creating a new Raw session.
3. Call `DRE_rawSessOpen()` to open an encode Raw session.
4. Call `DRE_rawSessSubmitSync()` to process a packet of data using the encode Raw session.
5. Call `DRE_rawSessOpen()` to open a decode Raw session.
6. Call `DRE_rawSessSubmitSync()` to process a packet of data using the decode Raw session. The data to decode is taken from the destination buffer from the process in step 4.
7. Compare the decode data in the destination buffer from step 7 to the data in the source buffer from step 4.
8. Call `DRE_rawSessClose()` to close the encode Raw session.
9. Call `DRE_rawSessClose()` to close the decode Raw session.
10. Call `DRE_symKeyDestroy()` to destroy the symmetric keys if necessary.

7.3.1.2 Asynchronous mode

1. Call `DRE_symKeySet()` to create the symmetric keys if necessary.
2. Set the parameters for creating a new Raw session.
3. Call `DRE_rawSessOpen()` to open an encode Raw session.
4. Call `DRE_rawSessSubmitAsync()` to process a packet of data using the encode Raw session.
5. Wait for the corresponding callback notification that the data packet has completed.

6. Call `DRE_rawSessOpen()` to open a decode Raw session.
7. Call `DRE_rawSessSubmitAsync()` to process a packet of data using the decode Raw session. The data to decode is taken from the destination buffer for the process in step 4.
8. Wait for the corresponding callback notification that the data packet has completed.
9. Compare the decoded data in the destination buffer from step 8 to the data in the source buffer from step 4.
10. Call `DRE_rawSessClose()` to close the encode Raw session.
11. Call `DRE_rawSessClose()` to close the decode Raw session.
12. Call `DRE_symKeyDestroy()` to destroy the symmetric keys if necessary.

7.3.2 Data Offload example Application

This section describes the basic flow for submitting a packet of data for processing in a Data Offload session in both synchronous and asynchronous mode.

7.3.2.1 Synchronous mode

1. Call `DRE_symKeySet()` to create symmetric keys if necessary.
2. Call `DRE_doSessOpen()` to create a Data Offload session.
3. Call one of `DRE_doSessSetEncodeDecode()`, `DRE_doSessSetComp()`, `DRE_doSessSetEnc()`, or `DRE_doSessSetHash()` to set the encode processing session.
4. Call `DRE_doSessWrite()` to write the source data to the session.
5. Call `DRE_doSessRead()` to retrieve the processed data.
6. Call one of `DRE_doSessSetEncodeDecode()`, `DRE_doSessSetComp()`, `DRE_doSessSetEnc()`, or `DRE_doSessSetHash()` to set the decode processing session.
7. Call `DRE_doSessWrite()` to write the destination data from step 5 to the session.
8. Call `DRE_doSessRead()` to retrieve the processed data.
9. Compare the destination data read from step 8 to the source data written in step 4.
10. Call `DRE_doSessClose()` to destroy the Data Offload session.
11. Call `DRE_symKeyDestroy()` to destroy symmetric keys if necessary.

7.3.2.2 Asynchronous mode

1. Call DRE_symKeySet() to create symmetric keys if necessary.
2. Call DRE_doSessOpen() to create a Data Offload session.
3. Call DRE_doSessSetEncodeDecode(), DRE_doSessSetComp(), DRE_doSessSetEnc(), or DRE_doSessSetHash() to set the encode processing session.
4. Call DRE_doSessWrite() to write the source data to the session.
5. Wait for the corresponding callback notification that the data packet has completed.
6. Call DRE_doSessRead() to retrieve the processed data.
7. Wait for the corresponding callback notification that the data packet has completed.
8. Call DRE_doSessSetEncodeDecode(), DRE_doSessSetComp(), DRE_doSessSetEnc(), or DRE_doSessSetHash() to set the decode processing session.
9. Call DRE_doSessWrite() to write the destination data from step 6 to the session.
10. Wait for the corresponding callback notification that the data packet has completed.
11. Call DRE_doSessRead() to retrieve the processed data.
12. Wait for the corresponding callback notification that the data packet has completed.
13. Compare the destination data read from step 8 to the source data written in step 4.
14. Call DRE_doSessClose() to destroy the Data Offload session.
15. Call DRE_symKeyDestroy() to destroy symmetric keys if necessary.

7.3.3 PK example Application

This section describes the basic flow of the PK example application.

1. Call Dre_ExamplePkCreateKey() which calls DRE_pkKeyCreateKey() to create the appropriate PK keys according to the PK key type.
2. Call Dre_ExamplePkExecuteCmd which calls DRE_pkExecCmd to execute the PK command according to the PK command type.

3. Wait for the callback function `Dre_ExamplePkCallBack()` to notify the Example application that the PK commands have completed. Check whether the PK command result is ok or not in the callback function.
4. Call `DRE_pkKeyDestroy()` to destroy all the PK keys which were created in step 1.

7.4 Debugging Tools

The DX SDK includes three tools that are useful when debugging: `dx_monitor`, `dx_status`, and `dx_diag`.

7.4.1 Monitor Tool

The `dx_monitor` tool can be used to report the temperature of each installed 820x device. It is intended to be used in a production environment.

For help using the `dx_monitor` tool, enter:

```
dx_monitor -h
```

For the detailed man pages of the `dx_monitor` tool, enter:

```
man dx_monitor
```

7.4.2 Status Tool

The `dx_status` tool can be used to report the number of installed DX 18xx cards, the PCIe version that is supported by the local host, the PLX switch speed and width capabilities, and the PCIe speed and width for which the DX 18xx cards are operating. Note that running the status tool requires `lspci` version 2.2.7 or higher.

For help using the `dx_status` tool, enter:

```
dx_status -h
```

For the detailed man pages of the `dx_status` tool, enter:

```
man dx_status
```

7.4.3 Diagnostic Tool

The `dx_diag` tool is intended to be used in a non-production test environment. If used incorrectly in a production environment, `dx_diag` may cause serious problems with the system.

The tool has many options that are useful for debugging a DX 18xx card, such as the ability to run POST on a specific 820x device or all 820x devices, ability to turn on the Flash LED to help map the physical to logical slot numbers, enable or disable the PCIe link to each 820x device, inject a false error, and more.

For help using the `dx_diag` tool, enter:

```
dx_diag -h
```

For the detailed man pages of the dx_diag tool, enter:

```
man dx_diag
```

8 Error Handling

This chapter provides supplemental material to help Express DX SDK users understand the types of errors they may encounter and the recommended actions that should be taken.

The error handling in this section pertains to Exar's Data Offload and Raw Acceleration APIs.

8.1 Definition of Error Status Codes

The DX SDK provides APIs to facilitate the customer application design. Each of the APIs return a 32-bit status code to the application. SDK provides macros to easily extract useful information from the status code.

DRE_IS_RESULT_ERR(status)

This macro returns true if any error is found in the status code, and returns false if no errors are found.

DRE_GET_ERR_CATEGORY(status)

This macro returns the category field of the error code.

DRE_GET_ERR_SEVERITY(status)

This macro returns the severity field of the error code.

DRE_GET_ERR_FLAG(status)

This macro returns the flags field of the error code.

Please refer to *dre_err_define.h* for more information.

The first macro that the user application should call is **DRE_IS_RESULT_ERR()** to identify whether the operation was successful. If an error occurred, then the user can extract the error category from the status code using the macro **DRE_GET_ERR_CATEGORY()**. Typically, the user application will only require the error category to determine the action required to handle the error. The status code information is also sent to the log file.

The status code is a 32-bit value which is defined in [Table 8-1](#). Each field is described in more detail in the sub-sections that follow.

Table 8-1. Error Status Fields (Sheet 1 of 2)

Bits	Name	Description
31:28	Status	Status of the test. 0000: DRE_OK 0100: DRE_WARN 0010: DRE_INFO 1110: DRE_ERR All other values are reserved.

Table 8-1. Error Status Fields (Sheet 2 of 2)

Bits	Name	Description
27:24	Flag	Error flag bits returned by the API. xxx0: The command was not processed by the software library xxx1: The command was processed by the software library xx0x: No expansion occurred during the compression operation. xx1x: A compression operation expanded beyond the data length plus threshold that was set when the session was opened. All other values are reserved.
23:20	Reserved	
19:16	Category	The error category returned by the API. 0x00: DRE_NO_ERR 0x01: DRE_SYSTEM_BUSY 0x02: DRE_USER_USAGE 0x03: DRE_HARDWARE_UNCONFIRMED 0x04: DRE_HARDWARE_CONFIRMED All other values are reserved.
15:8	Module code	Sub-module where error occurred. This values for this field are product specific. 0 = DRE_GENERAL 1 = DRE_OSAL 2 = DRE_DRIVE_FRAMEWORK 3 = DRE_REGISTER 4 = DRE_PP 5 = DRE_PDM 6 = DRE_PK 7 = DRE_SESSION_MANAGER 8 = DRE_NOTIFY_MANAGER 9 = DRE_DEV_MANAGER 10 = DRE_KEY_MANAGER 11 = DRE_RNG_MANAGER 12 = DRE_LB_MANAGER 13 = DRE_SW_LIB 14 = DRE_FILE_PARSE 15 = DRE_LOG 16 = DRE_USER_SPACE_CONTEXT_MANAGER 17 = DRE_USER_SPACE_API 18 = DRE_POST 19 = DRE_HW 20 = DRE_FLASH
7:0	Error Code	The returned detailed error code. This values for this field are product specific. Please refer to Table 8-2 for details.

Table 8-2 lists the specific error codes (bits [7:0] of the status code) that apply to the 820x device and DX cards.

Table 8-2. Error Code Field Description

Value	Name	Description
2	DRE_C_INVALID_VALUE	A specified parameter has an illegal value
3	DRE_C_INVALID_CARD_NUM	The card number is an illegal value
4	DRE_C_NOT_ENOUGH_RESOURCE	Failed to allocate the memory resource
5	DRE_C_REG_VERIFY	The value read from the register is not equal to the written value
6	DRE_C_FILE_OPEN	Failed to open file
7	DRE_C_FILE_END	The end of the file was reached
8	DRE_C_BUSY	Resources are busy; call again later
9	DRE_C_IRQL_TOO_HIGH	IRQL of calling context is too high
10	DRE_C_TIMER_STOPPED	Timer has stopped
11	DRE_C_WORK_ITEM_SCHEDULED	The work item has already been scheduled
12	DRE_C_WAITING_INTERRUPTED	Semaphore has been interrupted
13	DRE_C_CMD_NODE_IN_USE	Command node is in use
14	DRE_C_CMD_FREE_LIST_EMPTY	Free list is empty
15	DRE_C_CMD_PENDING_QUEUE_EMPTY	Pending queue is empty
16	DRE_C_NOT_FIND	Search for resource failed
17	DRE_C_RESOURCE_NOT_INIT	Resource not initialized
18	DRE_C_INIT_CARDS	Failed to initialize cards
19	DRE_C_GET_KEY_INDEX	Get key index failed
20	DRE_C_WRITE_KEY	Write key failed
21	DRE_C_CRC_VERIFICATION	CRC verification failed
22	DRE_C_DIF_VERIFICATION	DIF verification failed
23	DRE_C_BUFLNGTH_NOT_ENOUGH	Buffer too small to store data
24	DRE_C_RESWRITE_READ_NOT_EQUAL	Value read from register is not what was written
25	DRE_C_RECOVERY_UNDER_PROCESSIN	Error recovery process is still running
26	DRE_C_POWER_STATE_TRANSIT	Power state transition failed
27	DRE_C_FLASH_DATA_CORRUPTED	A flash region is corrupted
28	DRE_C_PLL_STATUS	PLL status error
29	DRE_C_LANE_STATUS	PCIe lane status error
30	DRE_C_CMD_NO_RESULT	Command result is missing
31	DRE_C_TIMEOUT	Command time out
32	DRE_C_DAEMON_EXIT	Daemon exit
33	DRE_C_HELP	Internal use
34	DRE_C_EXCEED_LIMIT	Exceeded number of allowed devices
35	DRE_C_DUPLICATED	Resource duplicated

Table 8-2. Error Code Field Description

Value	Name	Description
36	DRE_C_HMAC_CHECK	HMAC verification failed
37	DRE_C_GCM_TAG_CHECK	GCM tag verification failed
38	DRE_C_COMP	Compression failed
39	DRE_C_DECOMP	Decompression failed
40	DRE_C_INVALID_CALL	Improper function was called
41	DRE_C_SYSTEM	Failed to call system APIs
42	DRE_C_FILEFORMAT	File format error
43	DRE_C_DST_OVERFLOW	Destination buffer overflow
44	DRE_C_SRC_DATA_CORRUPTED	Source data corrupted
45	DRE_C_DATA_VERIFICATION	Data pair wise test failed
46	DRE_C_HW_EXE_ERR	Hardware execute command failed
47	DRE_C_OVERHEAT	Hardware temp is over the threshold
48	DRE_C_CARD_UNDER_RECOVERY	Card is undergoing error recovery
49	DRE_C_HOST_COMPLETE_ABORT	Invalid pAddr error
50	DRE_C_DSA_VERIFY_FAILED	DSA verification failed
51	DRE_C_DSA_SIGN_FAILED	DSA sign failed
52	DRE_C_LOAD_CODE_FAILED	Code load failed
53	DRE_C_UNKNOWN_ERR	Unknown error
54	DRE_C_RNG_EXCEED_MAX_SAFE_NUMBER	RNG exceeded the max safe number
55	DRE_C_POST_FAILED	POST failed
56	DRE_C_RESOURCE_LEAK	User did not release a resource
57	DRE_C_NOT_SUPPORT	Operation is not supported
58	DRE_C_PACKET_PENDING	Internal use
59	DRE_C_PACKET_KEEPING	Internal use, packet not releasing the session
60	DRE_C_PACKET_JUST_RECLAIM	Internal use, packet no longer used, is reclaimed to the packet pool, but did not call its callback function or release its sync semaphore
61	DRE_C_ECPOINT_VERIFY_FAILED	EC point verify failed
62	DRE_C_NEED_RESEED	Internal use, DRBG instance needs reseeding
63	DRE_C_NOT_ENABLED	The specified operation is not enabled. (e.g. submit PK command but pk_enable=0 in <i>driver.cfg</i>)

8.1.1 Error Category

The Error Category field should be read by calling `DRE_ERR_CATEGORY()` or by reading the error category field, bits [19:16] of the error code if the returned status from an API function call returns an error status. This field of the error code defines the error category as one of the following:

`DRE_ERR_SYSTEM_BUSY`

A System Busy error rarely occurs, has no impact on other API calls, and the call itself is likely to be correct. The user should resubmit the API call until the error is no longer reported. For example, if a user calls `DRE_rawSessOpen()` and receives the error code with category `DRE_ERR_SYSTEM_BUSY`, the processing of this call may be caused by memory resource allocation failure. In this case, the user should call the routine again.

`DRE_ERR_USER_USAGE`

This error is purely a user failure, has no impact on other API calls, and the call or data should be corrected before being resubmitted. Normally such an error is caused by passing the wrong parameter to the API or by corrupted data. The user should correct the parameter or data and resubmit the command. For example, if the user calls the API `DRE_rawSessSubmitAsync()` and supplies an invalid algorithm parameter, the error code `DRE_ERR_USER_USAGE` will be returned. Another example would be if the eLZS software verification failed due to a RCRC error, then error code `DRE_ERR_USER_USAGE` will be returned with `DRE_C_SRC_DATA_CORRUPT` in status code bit [7:0] set to indicate that the data is corrupted.

`DRE_ERR_HARDWARE_UNCONFIRMED_ERR`

This category of error represents a data verification error. Users may retrieve statistics on the frequency of this error category using the function `DRE_cardInfoGet()`. If this error occurs infrequently, the user should resubmit the failed command. If this error occurs frequently, the user should reset the hardware. If the problem persists, the hardware has failed and the user should contact technical support.

`DRE_ERR_HARDWARE_CONFIRMED_ERR`

This error is a confirmed hardware error, and may occur if the hardware has entered the error state. The user must perform a full reset of the hardware, reload the driver, and re-initialize the SDK. The user should then try resubmitting the command. If the problem persists, the hardware has failed and the user should contact technical support.

8.1.2 Error Flags

The Flags field of the error code provides the application additional information about the command result. The SDK provides a macro `DRE_GET_ERR_FLAG()` to easily retrieve the error Flags.

The Flags identify whether the command was processed by the hardware or by the software library, and also whether or not there was expansion of a compression command.

Express DX SDK versions 1.3.0L and later have the option of processing all commands for a particular session in hardware or in software. Please refer to the [“Failover”](#) section for more information.

The application must evaluate the return values of both `DRE_IS_RESULT_ERR()` and `DRE_IS_CMD_PROC_BY_SW()` macros in order to determine the error cause and corrective action. [Table 8-3](#) defines the relationship between these two macros and the error conditions.

If a hardware error occurs frequently (more than once a week or once per million commands), the host software should alert the operator to read the SDK log file to determine the cause of the problem. If needed, issue “make report” in the root path of the SDK directory and send the generated tar file to technical support for evaluation.

Table 8-3. Combined Error Status and Flag Conditions (Sheet 1 of 5)

Failover	CMD Target	Status Field DRE_IS_RESULT_ERR() ()	Software Library Flag Field DRE_IS_CMD_PROC_BY_SW() ()	Description	Action
X	HW	FALSE	FALSE	Command processed by hardware without errors.	None.
Disabled	HW	TRUE	FALSE	Command returned with error and not sent to software library. This condition may be caused by limited hardware bandwidth, malformed commands, or a hardware failure.	Call DRE_ERR_CATEGORY() for more information about the error. Follow the suggestions in "Error Category".
Disabled	HW	FALSE	TRUE	Case 1: Command processed but returned with a hardware error. Command then sent to the software library and is returned without errors.	Call DRE_cardInfoGet() to determine the hardware status. The device status will be OK because the hardware failure was not catastrophic. The error may be ignored. If the error reoccurs, reset the device. If the error still occurs frequently, replace the card.
				Case 2: Command cannot be processed by the hardware (e.g. device has failed or is recovering). Command then sent to the software library and is returned without errors.	This condition implies a hardware error and as a result will cause all pending commands to fail within a short period of time. Call DRE_cardInfoGet() to confirm the hardware error status. If the device was taken offline by the SDK due to a hardware failure, either continue processing on the remaining operational devices or replace the card. If the device is recovering, this error can be ignored but the status should be tested again at a later time.

Table 8-3. Combined Error Status and Flag Conditions (Sheet 2 of 5)

Failover	CMD Target	Status Field DRE_IS_RESULT_ERR() ()	Software Library Flag Field DRE_IS_CMD_PROC_BY_SW() SW()	Description	Action
Disabled	HW	TRUE	TRUE	Case 1: Command processed but returned with a hardware error. Command then sent to the software library and is returned with errors.	This condition implies a data corruption error. Verify the source data. If the data is correctable, resubmit the command.
				Case 2: Command cannot be processed by the hardware (e.g. device has failed or is recovering). Command then sent to the software library and is returned with errors.	
Enabled	HW	TRUE	FALSE	Command returned with error and is not sent to software library. This condition may be caused by limited system resources or an invalid command parameter.	Call DRE_ERR_CATEGORY() for more information about the error. If the error category is System Busy, resubmit the command. If the error category is User Usage, correct the invalid parameter(s) and resubmit the command.

Table 8-3. Combined Error Status and Flag Conditions (Sheet 3 of 5)

Failover	CMD Target	Status Field DRE_IS_RESULT_ERR() ()	Software Library Flag Field DRE_IS_CMD_PROC_BY_SW() SW()	Description	Action
Enabled	HW	FALSE	TRUE	Case 1: Command processed but returned with a hardware error. Command then sent to the software library and is returned without errors.	Call DRE_cardInfoGet() to determine the hardware status. If the device status will be OK because the hardware failure was not catastrophic. The error may be ignored. If the error reoccurs, reset the device. If the error still occurs frequently, replace the card.
				Case 2: Command cannot be processed by the hardware (e.g. device has failed or is recovering). Command then sent to the software library and is returned without errors.	This condition implies a hardware error and as a result will cause many commands to fail within a short period of time. Call DRE_cardInfoGet() to confirm the hardware error status. If the device was taken offline by the SDK due to a hardware failure, either continue processing on the remaining operational devices or replace the card. If the device is recovering, this error can be ignored but the status should be tested later.
				Case 3: All devices were in an errored state when the command was submitted. Command then sent to the software library and is returned without errors.	In this case, all submitted commands will return with an error. Call DRE_cardInfoGet() to determine the hardware status. Reset all devices in an unhealthy state.

Table 8-3. Combined Error Status and Flag Conditions (Sheet 4 of 5)

Failover	CMD Target	Status Field DRE_IS_RESULT_ERR() ()	Software Library Flag Field DRE_IS_CMD_PROC_BY_SW() ()	Description	Action
Enabled	HW	TRUE	TRUE	Case 1: Command processed but returned with a hardware error. Command then sent to the software library and is returned with errors.	This condition implies a data corruption error. Call DRE_ERR_CATEGORY() for more information about the error. If the error category is User Usage, call DRE_ERR_DETAIL_CODE() for the detailed error code. If the error code is DRE_C_SRC_DATA_CORRUPTED, verify the source data. and if the data is correctable, resubmit the command.
				Case 2: Command cannot be processed by the hardware (e.g. device has failed or is recovering). Command then sent to the software library and is returned with errors.	Refer to the Action for Case 1 above.
				Case 3: (Continued on next page.)	

Table 8-3. Combined Error Status and Flag Conditions (Sheet 5 of 5)

Failover	CMD Target	Status Field DRE_IS_RESULT_ERR() ()	Software Library Flag Field DRE_IS_CMD_PROC_BY_SW() ()	Description	Action
Enabled	HW	TRUE	TRUE	Case 3: All devices were in an errored state when the command was submitted. Command then sent to the software library and is returned with errors.	This condition implies a data corruption error, input parameter error, or that the system was busy. Call DRE_ERR_CATEGORY() for more information about the error. If the error category is System Busy, resubmit the command. If error category is User Usage, call DRE_ERR_DETAIL_CODE() for the detailed code. If the error code is DRE_C_SRC_DATA_CORRUPTED, verify the source data, and if the data is correctable, resubmit the command. Otherwise correct the invalid input parameter(s) and resubmit the command.
X	SW	FALSE	TRUE	Command processed by software without errors	None.
X	SW	TRUE	TRUE	Command returned by software library with error. This condition may be caused by corrupted source data or an invalid command parameter.	Call DRE_ERR_CATEGORY() for more information about the error. If error category is System Busy, resubmit the command. If error category is User Usage, correct the invalid parameter(s) and resubmit the command.

8.2 Failover

The decision whether to failover to software will only occur when all 820x devices are offline and will not occur during a recovery process. Packet processing commands that have been submitted to the hardware, but complete with an error will be processed by the software library, *swlib*, regardless of whether failover is enabled or disabled in the driver configuration file. Public key commands that have been submitted to the hardware, but complete with an error will return with an error and not be processed by the software library. [Section 8.3](#) describes the command error processing in more detail.

Failover applies only to future commands submitted to a card whose 820x devices are down. If failover is enabled, new commands will automatically be transferred to a healthy device or to the software library for processing if all devices are unhealthy.

8.3 Single Command Error Handling

8.3.1 Packet Processing Command Error Handling

If a packet processing command is retrieved from the hardware DMA ring with a hardware error flagged, the DX SDK will handle this command with the process described in [Figure 8-1](#). All packet processing commands that return with a hardware error will be sent to the software library for processing even if failover is not enabled.

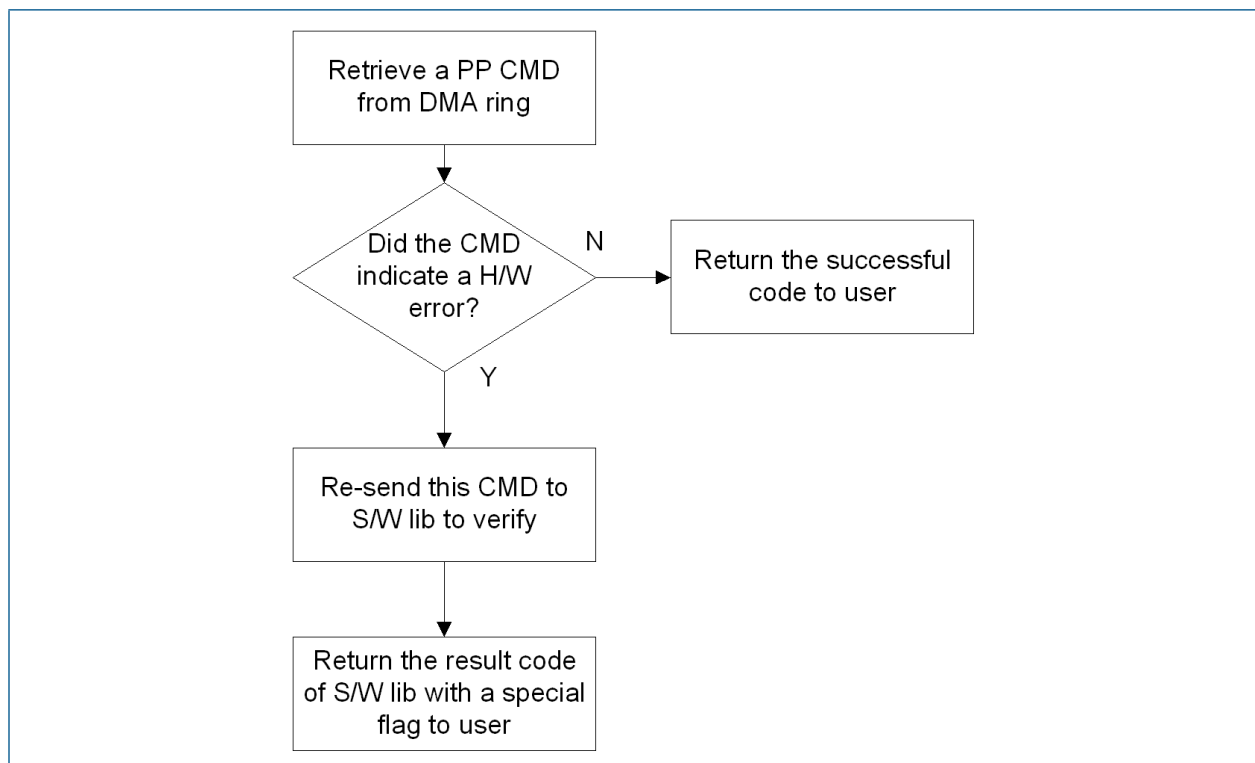


Figure 8-1. Single Packet Processing Command Failure Error Processing

8.3.2 Public Key Command Error Handling

For PK commands, the SDK will report a failure to the application, but the SDK will not route the PK command to the software library.

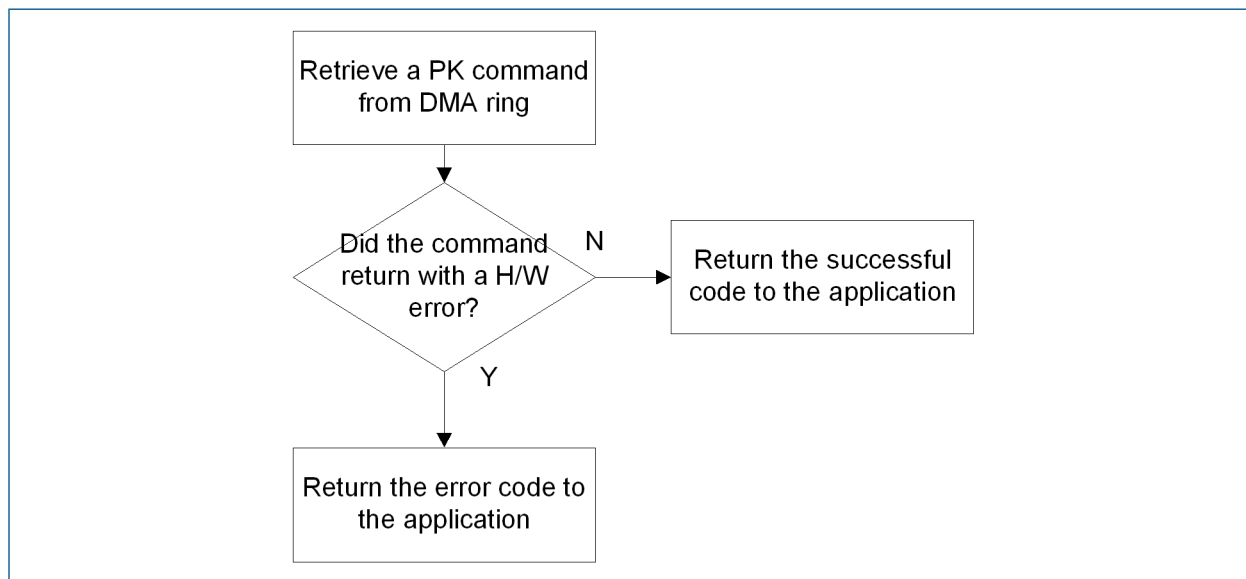


Figure 8-2. Single Public Key Command Failure Error Processing

8.4 Hardware Timeout Error Handling

The DX SDK maintains a 10 second interval a timer that is used to monitor a hardware timeout condition. If the hardware does not respond within the timeout interval, the SDK will soft reset the hardware and re-submit the commands from the DMA ring. If the hardware recovers successfully, it will continue to accept commands.

The following flow chart describes this process.

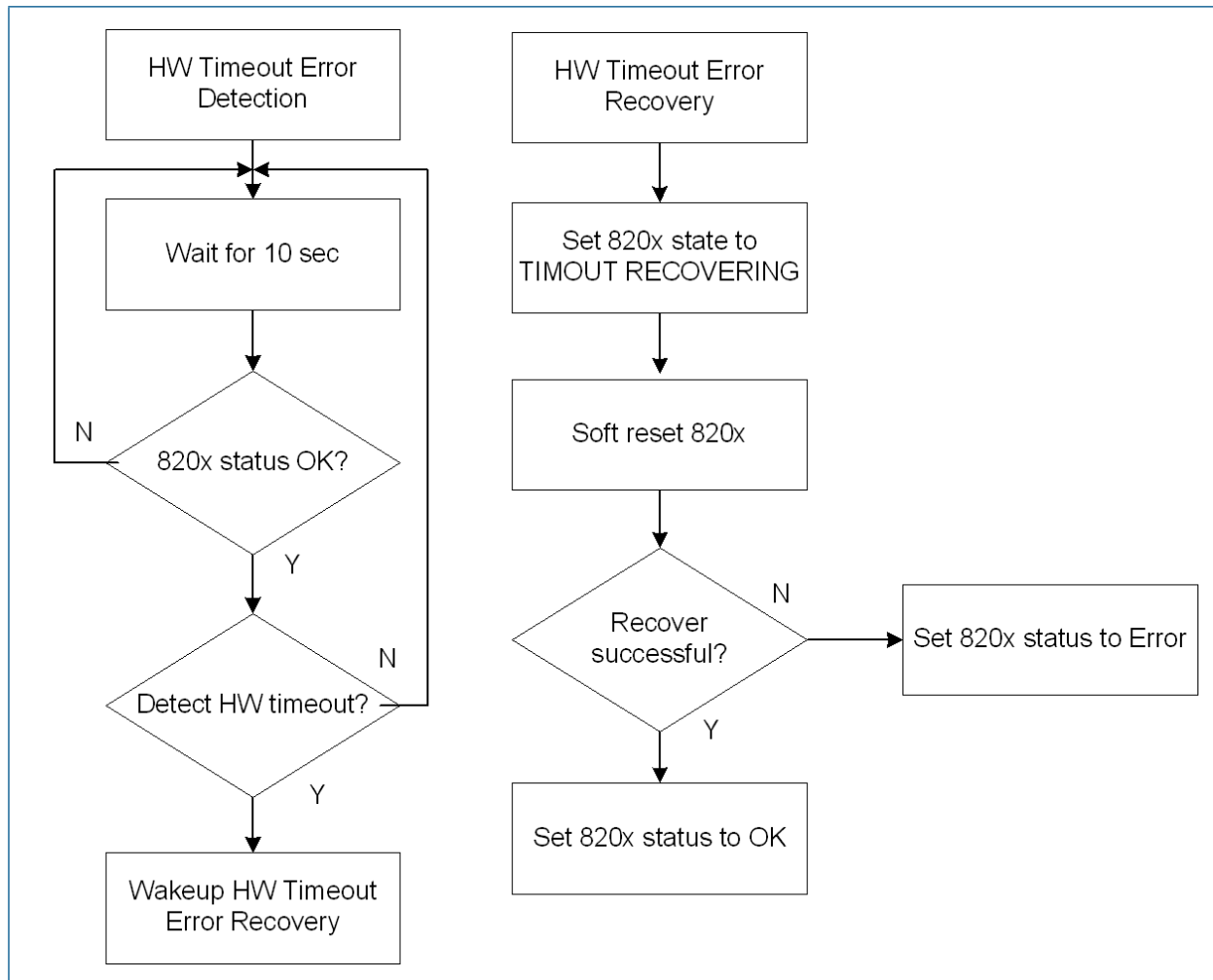


Figure 8-3. Hardware Timeout Error Processing

8.5 Overheated Condition Error Handling

Figure 8-4 illustrates the error handling by the DX SDK for detecting and recovering an 820x device that is overheated. The left hand side of the flow diagram represents the detection process while the right hand side represents the recovery process.

The DX SDK internally maintains more 820x status states than those exposed by the status field. If the DX SDK determines that a 820x device is overheated, the SDK will internally set that device's status to "overheated" and flash the Error LED. When a device is in the overheated state, that device will be removed from the load balancing table so that future commands will not be sent to that device until its temperature is recovered. Once the overheated condition is eliminated, the SDK will turn off the error LED.

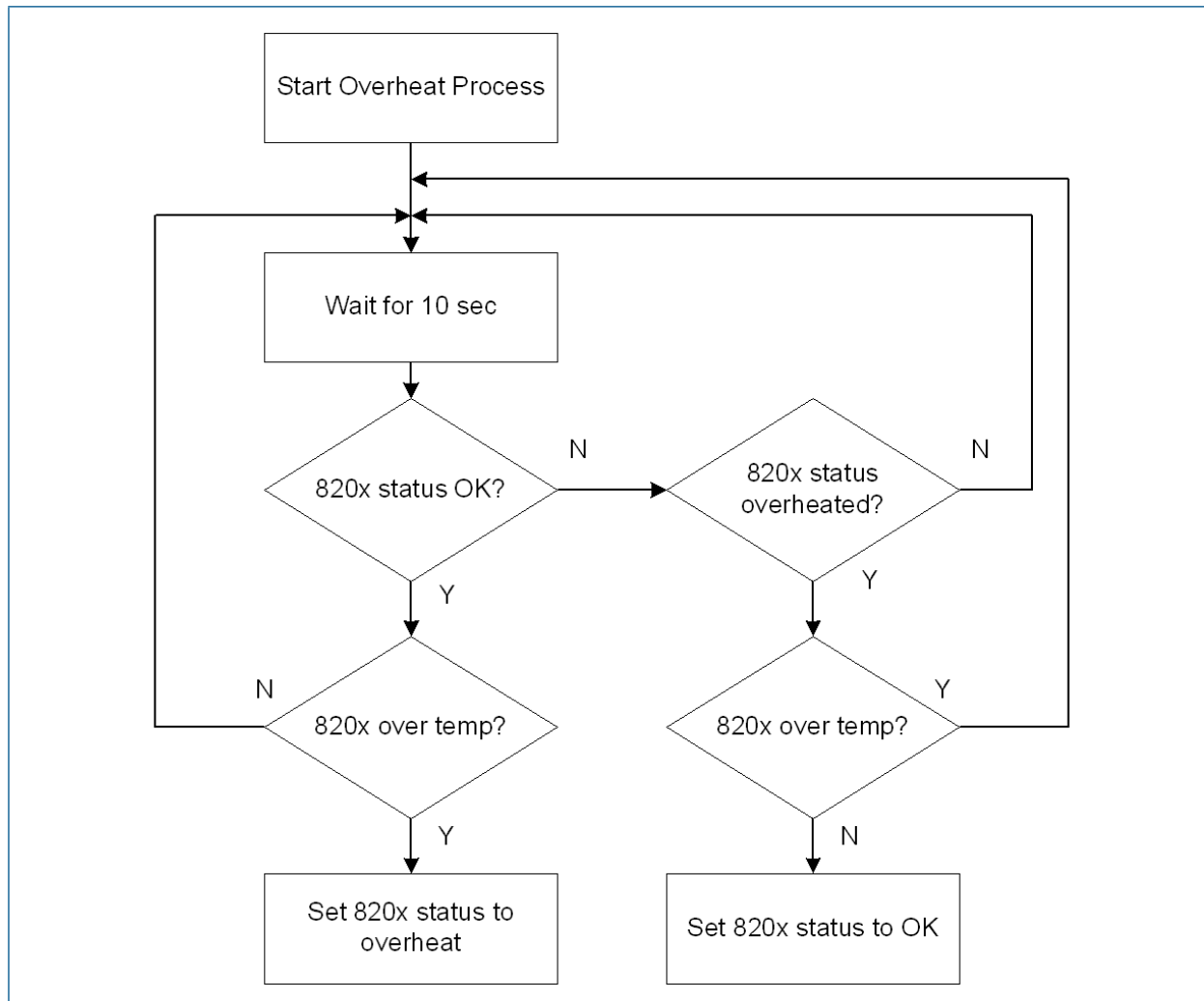


Figure 8-4. Hardware Overheated Error Processing

8.6 PCIe Error Handling

DX SDK versions 1.2.1L and later are able to detect and handle PCIe errors. The DX SDK will initiate PCIe error handling if any of the conditions below occur:

1. A register access returns 0xFFFFFFFF. The DX SDK will log the error message: "Register access failed (returns 0xFFFFFFFF) on Chip %d. The SDK is attempting to recover the device."
2. A 820x interrupt status register contained a PCIe error. The DX SDK will log the error message: "Chip %u asserted a PCIe error interrupt. The SDK is attempting to recover the device."

3. The DX SDK detects a PCIe error in the Root Complex port or the PCIe switch or the 820x device on the Express DX card after retrieving commands from the result ring. The DX SDK will log the error message: "One or more PCIe errors were detected. The SDK is attempting to recover from the error."

Figure 8-5 illustrates the PCIe error detection and Figure 8-6. Additional details are given in the following sections.

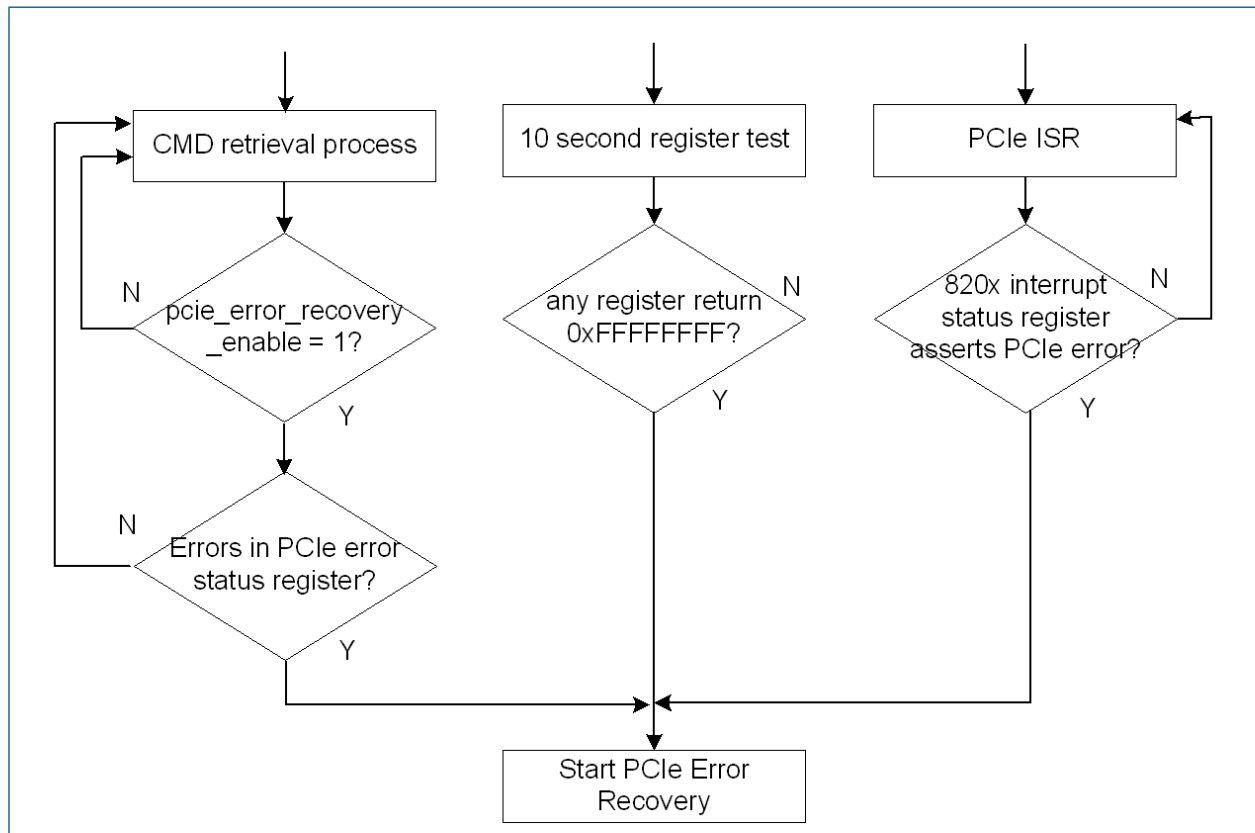


Figure 8-5. PCIe Error Detection Flow

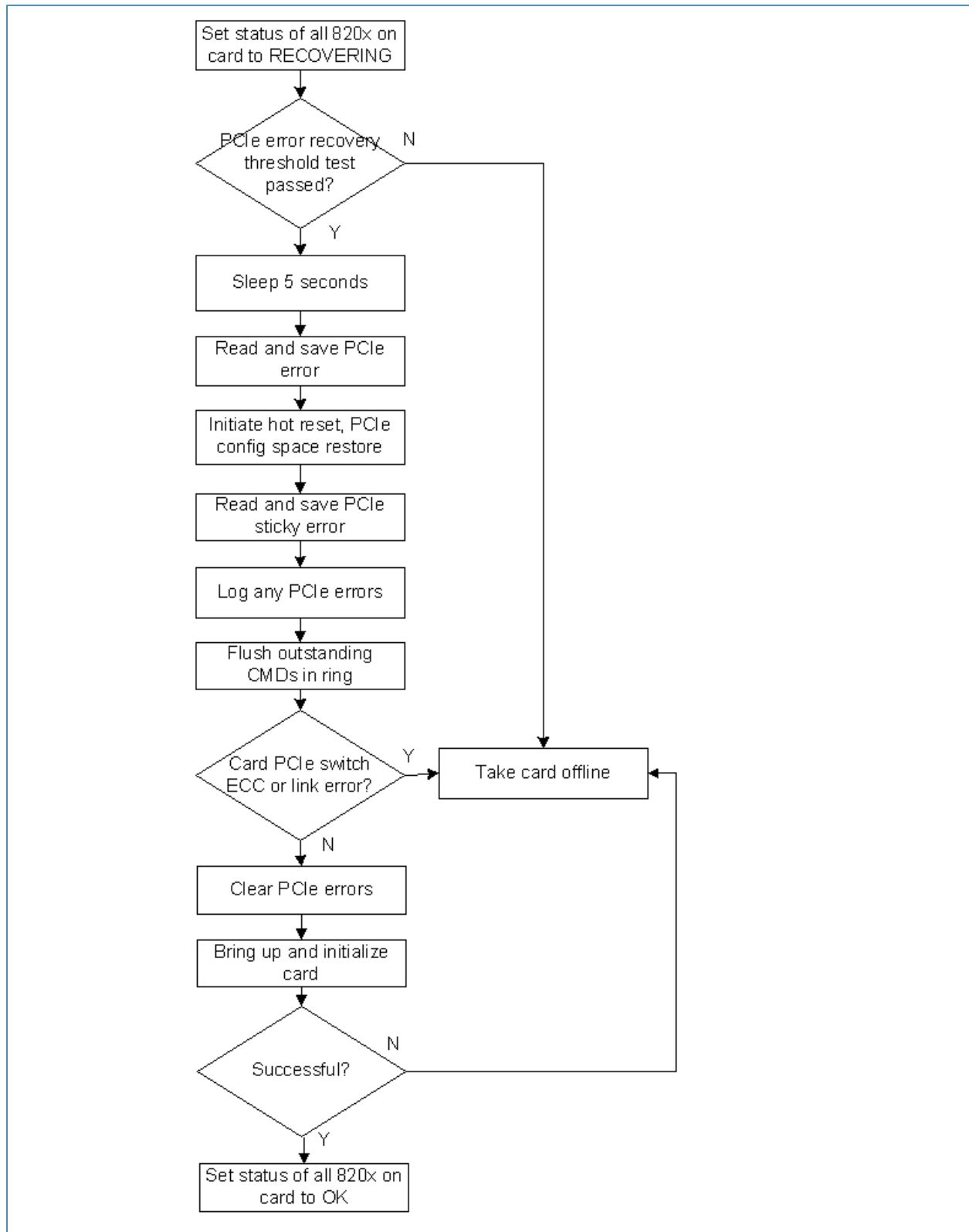


Figure 8-6. PCIe Error Recovery Flow

8.6.1 Register Access Error Detection

To improve the robustness of the DX SDK error detection, the DX SDK implements a timeout mechanism for every 820x register access that requires a polling mechanism to confirm whether the result is ready, e.g. to retrieve random bytes via API `DRE_rngRequestRaw()` or to read temperature via API `DRE_readTemp()`. If more than 2 seconds have elapsed before the 820x responds in those cases, the DX SDK will abort the polling, and return a status with error code `DRE_C_TIMEOUT`.

The SDK will also test for a potential PCIe link down error whenever an 820x register is accessed and trigger the PCIe error handling flow if `0xFFFFFFFF` is returned.

8.6.2 Handling Uncorrectable PCIe Errors

The DX SDK has the ability to handle PCIe root complex errors. This feature may be enabled or disabled via the driver configuration parameter `"pcie_error_recovery_enable"`.

If enabled, the DX SDK monitors the frequency of the PCIe errors for every installed DX card. If an uncorrectable PCIe error occurs, the DX SDK will reset that DX card, recover the card and all existing commands and continue the command processing. If a second PCIe error occurs within the interval defined by the driver configuration parameter `"pcie_error_interval_threshold"`, the DX SDK will take that DX card offline. The default value for `"pcie_error_interval_threshold"` is eight hours. To force the DX SDK to attempt to recover every PCIe error, set `"pcie_error_interval_threshold"` to zero. Note that the SDK will implement the PCIe error handling and reset the entire card even if the error is reported by a single 820x device.

8.7 Data Corruption Error Handling

Express DX SDK versions 1.2.1L and later added new features to detect and prevent PCIe data corruption on the Express DX 18xx cards.

To detect potential PCIe data corruption cases, for every group of completed commands, the DX SDK reads all PCIe error related registers and tests for errors. If an error is detected, the SDK will attempt to recover the error by applying a PCIe hot reset to the card and then reprocessing the affected commands in software using `swlib`.

To help prevent data corruption on a DX 18xx card, the DX SDK enables ECC fencing mode on the PCIe switch device and enables ECRC protection between the PCIe switch and the 8204 devices. Fencing mode essentially blocks PCIe traffic when a fatal PCIe or ECC error occurs. However, ECC fencing mode on the DX 18xx card PCIe switch device can only be triggered once. For this reason, the severity configuration for all PCIe errors reported by the PCIe switch except "Surprise Link Down" for the host side port connected to the Root complex have been modified in the SDK to be non-fatal so that only a fatal ECC error or a "Surprise Link Down" to the Root Complex will trigger fencing mode. The DX 18xx PCIe switch device has an erratum that will not allow a Surprise Link Down PCIe error to be configured as a non-fatal error on Port 0, the port connected to the Root Complex.

If a fatal ECC error occurs, the SDK will reset the card and take it offline. Commands that were already submitted to that card will be processed in software using swlib; all future commands will be submitted to other DX cards for processing. In this case, the DX 18xx card requires a hard reset when convenient to recover the card. The card does not need to be replaced or returned.

Appendix A: Usage and Standards

Compliance of the 820x Random Number Generator

A.1 Overview

Exar's 820x security processor and its Express DX Software Development Kit (SDK) provide a hardware and software Random Number Generator (RNG) solution that is approved by the Federal Information Processing Standards (FIPS) and National Institute of Standards and Technology (NIST) regulatory agencies.

The 820x processor contains a random number generator that produces a nondeterministic random number.

Customers may generate a cryptographic quality pseudo-random number, known as a Deterministic Random Number Bit Generator (DRBG), by implementing their choice of a FIPS 140-2 approved algorithm and using the RNG value from the 820x device as the initial seed.

The Express DX SDK provides an API function to retrieve RNG values from the 820x device (see ["Retrieve a Nondeterministic RNG Value from a 820x Device"](#) for more details), and a process entry that may be used to monitor the RNG statistics (see ["Monitoring the RNG Statistics"](#) for more details).

A.2 Hardware Implementation

Exar's 820x acceleration processor contains an internal RNG module that is used to generate a random number which may be used to seed a software generated Deterministic Random Number Bit Generator (DRBG).

The hardware RNG module generates FIPS compliant random number sequences. A high level block diagram of the RNG is shown in [Figure A-1](#).

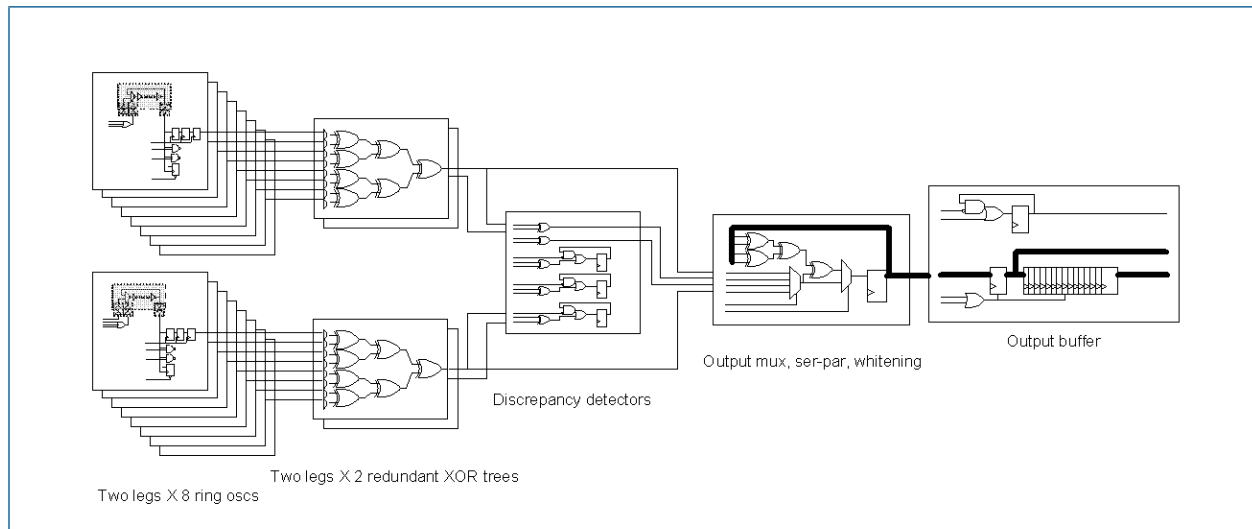


Figure A-1. RNG High Level Block Diagram

The RNG is implemented as 16 shielded, free running ring oscillators, with each ring containing a different prime number of gate inversions such that all ring oscillators generate an independent frequency higher than the 125 MHz core clock frequency that is used to sample the oscillator state. Exact frequencies are dependent on individual component process factors, operating die temperature, and operating voltage (PVT), but the ratios should be similar over these process variations.

Synchronizers are used to sample the state of each ring oscillator at the 125 MHz clock.

The synchronized ring oscillator samples are XOR'd together in redundant trees, with the two paths compared so as to raise an interrupt in the case of a discrepancy (hardware fault or synchronizer failure).

The selected tree output is fed through logic that performs the serial to parallel conversion and serves as a whitening LFSR when enabled. This is done at a small integer divide of the 125 MHz core clock.

At a much larger divide of the 125 MHz core clock, the parallel output then enters an output buffer. The software reads the random value from the output buffer.

Random number hardware registers provide the interface for software to start the random number hardware engine, retrieve sixteen 32-bit random number values, and configure the random number hardware engine. Please refer to the *820x Data Sheet* for a detailed description of the RNG registers. The Raw Acceleration API provides functions to retrieve a random number from the 820x RNG module, precluding the need to access the internal 820x registers.

A.3 Software Implementation

The Express DX SDK configures and controls the 820x RNG module. The Express DX SDK uses the default reset values for the 820x RNG registers, however DX SDK versions 1.2.2L and later allow for configuration of some RNG settings through parameters in the driver configuration file (see [Section 5.2.1](#)).

The DX SDK, and specifically the Raw Acceleration API, may be used to:

- Retrieve a nondeterministic RNG value from the RNG module
- Test the 820x RNG module

DX SDK versions 1.3.0L and later support NIST SP 800-90 (http://csrc.nist.gov/publications/nistpubs/800-90/SP800-90revised_March2007.pdf), and implement the CTR_DRBG that supports AES 128, 192, 256 bit keys, and implement the Dual_EC_DRBG that supports 256, 384, 521 bit keys. Refer to the *Raw Acceleration API User Guide*, UG-0189, for details.

In addition, DX SDK versions 1.2.2L and later include a “/proc” entry for “[Monitoring the RNG Statistics](#)”.

A.3.1 Retrieve a Nondeterministic RNG Value from a 820x Device

The Express DX SDK Raw Acceleration API provides a function to retrieve a RNG value from a 820x device. Within the DX SDK API, the nondeterministic RNG value retrieved from an 820x device is referred to as the Raw RNG. Please refer to the *Raw Acceleration Application Programming Interface Reference Guide*, UG-0189, for more detailed information.

```
DRE_status DRE_rngRequestRaw (
    DRE_u32b      cardNum,
    DRE_u08b      *pDataBuf,
    DRE_u32b      numReq,
    DRE_u32b      *pNumRet);
```

When called, this function will read the requested number of random numbers directly from the 820x RNG module.

[Figure A-2](#) depicts the flow for how the Express DX SDK retrieves a RNG value from the 820x. The SDK maintains a pool of random numbers. If the number of requested bytes is larger than the amount of random numbers stored in the pool, then the SDK will retrieve another 2048 bytes to fill its internal pool buffer. This process will repeat until the requested amount of random numbers is fulfilled.

The Express DX SDK implements one RNG object for each 820x device in the system. Each RNG object may be uniquely seeded by its 820x RNG.

If any of the RNG errors listed in the RNG register section of the 820x Data Sheet occur, this function will return a general RNG module error (module code = 11).

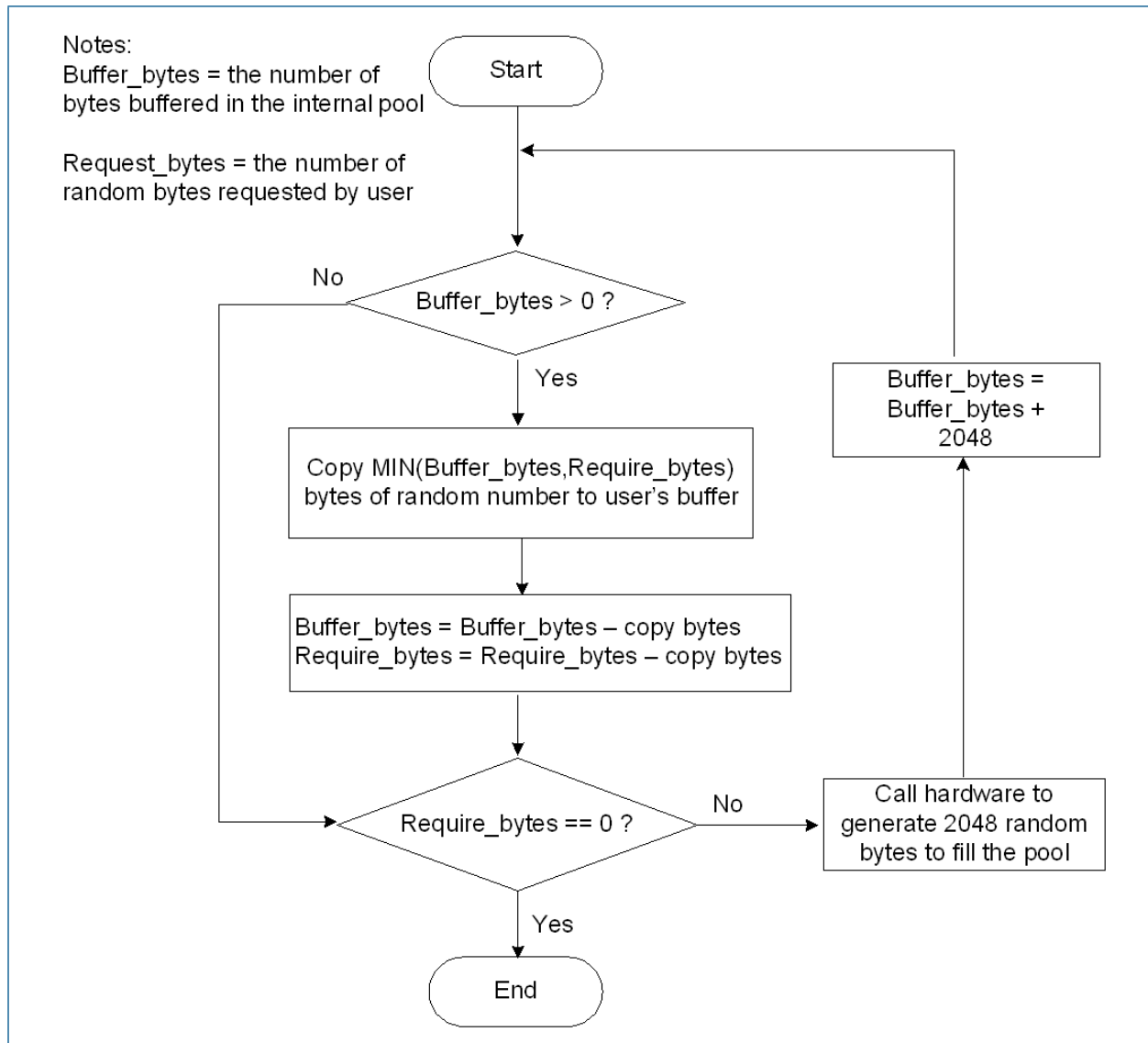


Figure A-2. Flow to Retrieve RNG Number from a 820x Device

A.3.2 RNG Test

The Express DX SDK versions 1.0.x and later meet the FIPS 140-2 Section 4.9.2 requirement for a continuous random number generator test. The Express DX SDK automatically runs the RNG test for every 32-bit words of data retrieved from the hardware.

The RNG test consists of the following steps.

1. Retrieve 16 double words (4 bytes) from the 820x RNG and save them as R0, R1,...,R15.
2. Verify that $R0 \neq R1$, $R1 \neq R2$, ..., $R14 \neq R15$.
3. If any of the equations in step 2 are not satisfied, the RNG test fails. Otherwise, the RNG test passes.

If the RNG test fails, any contiguous duplicated 32-bit words will be discarded, but the 820x device will continue to process commands. The Express DX SDK maintains a RNG test failure counter that is printed to the *dresys.log* file. The driver log level must be set to Warning or higher in order to trigger the RNG failure notice. For example,

```
Sep 01 09:57:38 Warning: RNG engine cardNum = 0 continuous test failed
Previous random value is 0x00000000
Current random value is 0x00000000
Failed RNG generation times are 4
The total number of generated rng data is 1036460256
```

A.3.3 Monitoring the RNG Statistics

Express DX SDK versions 1.2.2L and later include a Linux `/proc` entry that allows users to easily monitor the RNG status of all 820x devices in the system. As described in [Section A.3.1](#), a per-device pool buffers the random bytes retrieved from the 820x RNG engine.

To view the RNG status, enter:

```
cat /proc/panther_rng_status
```

The example output below was generated on a system with a single DX 1845 card after the driver was loaded.

```
----- chip 0 ----- PoolSize: 2048  PoolAvail: 1920 TotalGen:  2112      TestFailCnt: 0
----- chip 1 ----- PoolSize: 2048  PoolAvail: 1920 TotalGen:  2112      TestFailCnt: 0
----- chip 2 ----- PoolSize: 2048  PoolAvail: 1920 TotalGen:  2112      TestFailCnt: 0
----- chip 3 ----- PoolSize: 2048  PoolAvail: 1920 TotalGen:  2112      TestFailCnt: 0
```

Several comments need to be made about the hardware RNG and DX SDK behavior to explain the output results.

1. After the 820x RNG module is initialized, the first 64 random bytes are discarded and not entered into the pool. These 64 bytes will be included in the `TotalGen` statistic, but not in the `PoolAvail` statistic.
2. Every user request for random bytes from the RNG via the API function `DRE_rngRequestRaw()` will be serviced by an internal pool buffer whose size is 2048 bytes (`PoolSize`). If the number of available bytes in the pool (`PoolAvail`) is less than the number of random bytes requested, the DX SDK will copy the existing random bytes in the pool to the user buffer and then request another 2048 random bytes from the 820x device. This flow loops until the requested number of random bytes have been copied to the user buffer. This process is illustrated in [Figure A-2](#).

3. For every device probed and initialized when the driver is loaded, POST will retrieve 128 random bytes for its use.
4. The RNG test defined in [Section A.3.2](#) runs continuously after the driver had been loaded. The DX SDK maintains a counter for the number of times the test fails.

In the example output above, the total number of generated random bytes is given by `TotalGen = 64+2048 = 2112` bytes. The number of random bytes currently in the pool is given by the `PoolAvail = 2048-128 = 1920` bytes. The number of RNG test failures is given by `TestFailCnt`. In general, `TestFailCnt` will be incremented until a very large number of random bytes is generated.

A.4 Standards Requirements and Compliance

A.4.1 FIPS Requirements

The publication *Federal Information Processing Standards (FIPS) 140-2*, section 4.9.2 *Conditional Tests*, lists the tests that must be performed by a cryptographic module that contains a continuous random number generator. The continuous random number generator test states:

If each call to a RNG produces blocks of n bits (where $n > 15$), the first n -bit block generated after power-up, initialization, or reset shall not be used, but shall be saved for comparison with the next n -bit block to be generated. Each subsequent generation of an n -bit block shall be compared with the previously generated block. The test shall fail if any two compared n -bit blocks are equal.

A.4.2 NIST Requirements

The National Institute of Standards and Technology (NIST) publication *Annex C: Approved Random Number Generators for FIPS PUB 140-2, Security Requirements for Cryptographic Modules*, lists the algorithms that may be used to generate a deterministic random number.

National Institute of Standards and Technology, NIST-Recommended Random Number Generator Based on ANSI X9.31 Appendix A.2.4 Using the 3-Key Triple DES and AES Algorithms January 31, 2005.

For FIPS 140-2 approval, NIST requires that the random number be generated using a NIST-approved Deterministic Random Bit Generator (DRBG). The DRBG may be seeded from a hardware source, such as the 820x RNG. The benefit of this approach is that the DRBG algorithm can be chosen to be as strong and fast as possible, with the hardware RNG required to only produce a fairly small (e.g., 128-256 bits), one-time initial random seed for the DRBG.

Customers will typically use a DRBG to generate cryptographic keys and initialization vectors.

Cryptographic keys

Section 4.7.1 of FIPS 140-2 states:

An Approved RNG shall be used for the generation of cryptographic keys used by an Approved security function.

Initialization Vectors (IVs)

Appendix C: *Generation of Initialization Vectors* in the NIST Special Publication 800-38A - *Recommendations for Block Cipher Modes of Operation* states:

There are two recommended methods for generating unpredictable IVs. The first method is to apply the forward cipher function, under the same key that is used for the encryption of the plaintext, to a nonce. The nonce must be a data block that is unique to each execution of the encryption operation. For example, the nonce may be a counter, as described in Appendix B, or a message number. The second method is to generate a random data block using a FIPS approved random number generator.

A.4.3 NIST Compliance

The 820x device was validated by NIST as conforming to the Deterministic Random Bit Generator (DRBG) algorithm as specified in Special Publication 800-90, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*.

The certification may be viewed online using the link below. Exar Corporation's validation number is #53.

<http://csrc.nist.gov/groups/STM/cavp/documents/drbg/drbgval.html>

In order to achieve NIST certification, a special application was run on top of the Express DX SDK's Raw Acceleration API to access the RNG and SHA-256 functions of the 820x device to realize the Hash_DRBG method described in NIST SP 800-90.

It is anticipated that a future release of the Express DX SDK will contain a DRBG implementation and API function that when called will return a NIST SP 800-90 approved deterministic random number.

A.5 References

1. *NIST-Recommendation Random Number Generator Based on ANSI X9.31 Appendix A2.4 Using the 3-Key Triple DES and AES Algorithms*,
<http://csrc.nist.gov/groups/STM/cavp/documents/rng/931rngext.pdf>
1. Federal Information Processing Standards (FIPS) 140-2,
<http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf>
2. Annex C: Approved Random Number Generators for FIPS PUB 140-2, Security Requirements for Cryptographic Modules,
<http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexc.pdf>
3. NIST Special Publication 800-90 Recommendation for Random Number Generator Using Deterministic Random Bit Generators (Revised),
http://csrc.nist.gov/publications/nistpubs/800-90/SP800-90revised_March2007.pdf
4. NIST Special Publication 800-90 Recommendation for Block Cipher Modes of Operation, Methods and Techniques,
<http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf>

Appendix B: Exported Software Algorithms

This section documents the exported eLZS and hash software functions.

B.1 eLZS

The eLZS221-C Data Compression Software Library provides a processor independent software implementation of Exar's enhanced LZS (eLZS) algorithm. This document refers to version 1.20.0 of eLZS221-C.

The eLZS functions can be called under any context in kernel or user space.

Note that files compressed with LZS may be decompressed with eLZS, but files compressed with eLZS cannot be decompressed by LZS. Files compressed or decompressed with hardware or software may be compressed or decompressed interchangeably with eLZS hardware or software.

B.1.1 eLZS_Compress()

Include Files:

elzs.h

Syntax:

```
eLZS_len_type eLZS_Compress(
    const void          *srcPtr,
    eLZS_len_type       srcCnt,
    void                *dstPtr,
    eLZS_len_type       dstBufSize,
    unsigned int         flags);
```

Description:

The function `eLZS_Compress()` can be used to perform synchronous software-based eLZS compression. The detailed declaration of this function is located in the file *elzs.h*.

The parameter `srcPtr` is a pointer to source data buffer whose length is specified by the parameter `srcCnt`. The argument `dstPtr` specifies the location of the destination buffer to hold the compressed data. The parameter `dstBufSize` defines the maximum length of the destination buffer. Due to an internal requirement, the source or destination buffer length cannot exceed 3M.

When the function successfully returns, the destination buffer is filled with the compressed data, and the return value contains the length of the compressed data stored in the destination buffer. If the operation fails, this function returns an appropriate error status. The user should call the boolean macro `eLZS_IsError()` on the return value to determine if the operation succeeded or failed (see "Returns" below).

The parameter `flags` is used to modify the behavior of this function and fine-tune the compression ratio and speed, as defined in [Table B-1](#) below. The flags may be logically ORed to set more than one flag option.

Table B-1. eLZS Compression Flag Definitions

Flag Value	Description
0x000 = ELZS_FLAG_SRCH_LVL_DEFAULT	Balanced speed versus compression ratio (default)
0x001 = ELZS_FLAG_CRC32	A 32-bit CRC will be computed, compressed, and appended to the destination buffer
0x008 = ELZS_FLAG_EOCD_NO_PAD	Do not pad to 32-bit boundary in compression after EOCD
0x100 = ELZS_FLAG_SRCH_LVL_MAX_SPEED	Maximize speed with no regard for compression ratio
0x200 = ELZS_FLAG_SRCH_LVL_MAX_COMP	Maximize compression ratio with reduced speed
0x300 = ELZS_FLAG_SRCH_LVL_OPTIMAL	Optimize compression ratio with no regard for speed
All other values are undefined.	

Parameters:

srcPtr	Input	The pointer to the data to be compressed.
srcCnt	Input	Number of source data bytes to be compressed.
dstPtr	Input/Output	Pointer to where compressed data will be stored.
dstBufSize	Input	Size in bytes of the destination buffer.
flags	Input	Flags used to modify the behavior of the function. See Table B-1 .

The tables below demonstrate the performance variation of the **flags** parameter settings 0x000 (balanced), 0x100 (max speed), 0x200 (improved compression at reduced speed) and 0x300 (optimal compression). Note that the default setting will be appropriate for most applications.

[Table B-2](#) illustrates the performance on three standard Calgary Corpus data files and over an average of 18 Calgary Corpus data files.

Table B-2. Calgary Corpus Performance Test Results

Calgary Filename	Balanced	Max speed	Improved Comp	Optimal Comp
BIB	Comp ratio = 1.68:1 Speed = 41.3 clocks/byte	Comp ratio = 1.50:1 Speed = 24.8 clocks/byte	Comp ratio = 1.95:1 Speed = 92.0 clocks/byte	Comp ratio = 2.05:1 Speed = 559.9 clocks/byte
GEO	Comp ratio = 1.25:1 Speed = 42.9 clocks/byte	Comp ratio = 1.18:1 Speed = 26.9 clocks/byte	Comp ratio = 1.31:1 Speed = 95.7 clocks/byte	Comp ratio = 1.37:1 Speed = 815.9 clocks/byte
PIC	Comp ratio = 6.02:1 Speed = 11.0 clocks/byte	Comp ratio = 5.71:1 Speed = 8.5 clocks/byte	Comp ratio = 6.76:1 Speed = 580.9 clocks/byte	Comp ratio = 7.04:1 Speed = 5533.3 clocks/byte
Average of 18 files	Comp ratio = 1.92:1 Speed = 37.76 clocks/byte	Comp ratio = 1.72:1 Speed = 23.23 clocks/byte	Comp ratio = 2.22:1 Speed = 279.03 clocks/byte	Comp ratio = 2.33:1 Speed = 1053.21 clocks/byte
Test setup: Intel Core 2 Duo running 64-bit Windows 7 Enterprise. Compiled with 64-bit MSVC 2008 compiler with optimization setting /O2. Block size = 16K				

Table B-3 shows the performance using standard Canterbury Corpus data.

Table B-3. Canterbury Corpus Performance Test Results

Calgary Filename	Balanced	Max speed	Improved Comp	Optimal Comp
ALICE29	Comp ratio = 1.70:1 Speed = 44.6 clocks/byte	Comp ratio = 1.50:1 Speed = 26.2 clocks/byte	Comp ratio = 1.99:1 Speed = 114.0 clocks/byte	Comp ratio = 2.10:1 Speed = 708.3 clocks/byte
PTT5	Comp ratio = 6.02:1 Speed = 11.0 clocks/byte	Comp ratio = 5.71:1 Speed = 8.4 clocks/byte	Comp ratio = 6.76:1 Speed = 585.7 clocks/byte	Comp ratio = 7.04:1 Speed = 5534.2 clocks/byte
PLRABN1 2.TXT	Comp ratio = 1.53:1 Speed = 49.2 clocks/byte	Comp ratio = 1.40:1 Speed = 27.1 clocks/byte	Comp ratio = 1.78:1 Speed = 124.8 clocks/byte	Comp ratio = 1.89:1 Speed = 647.0 clocks/byte
Average of 11 files	Comp ratio = 2.41:1 Speed = 34.33 clocks/byte	Comp ratio = 2.19:1 Speed = 21.16 clocks/byte	Comp ratio = 2.74:1 Speed = 185.58 clocks/byte	Comp ratio = 2.86:1 Speed = 1463.26 clocks/byte
Test configuration: Intel Core 2 Duo running 64-bit Windows 7 Enterprise. Compiled with 64-bit MSVC 2008 compiler with optimization setting /O2. Block size = 16K				

Returns:

The boolean macro `eLZS_IsError()` should be called on the return value of `eLZS_Compress()` to determine the status of the compression operation. If `eLZS_IsError()` returns false, the function completed without errors and the return value represents the number of bytes of compressed data in the destination buffer. If `eLZS_IsError()` returns true, the destination buffer will not be filled with compressed data and the return value will contain one of the error codes defined below.

Error	Value	Description
ELZS_ERROR_SRC_UNDERRUN	-1	Source data corrupted.
ELZS_ERROR_DST_OVERFLOW	-2	Not enough space in destination buffer to store the result
ELZS_ERROR_BAD_TOKEN	-3	Source data corrupted.
ELZS_ERROR_BAD_CRC	-4	CRC verification failed.
ELZS_ERROR_BAD_RAW_LEN	-5	Source data corrupted.
ELZS_ERROR_BAD_RAW_TOK	-6	Source data corrupted.

B.1.2 eLZS_Compress_Description()

Include Files:

elzs.h

Syntax:

```
const char *eLZS_Compress_Description(unsigned int flags);
```

Description:

This function is used to return a textual description of the value that was set for the compression performance settings using the **flags** parameter.

Parameters:

flags	Input	Returns the compression performance.
		0x000 = balanced speed versus compression ratio (default)
		0x100 = maximize speed at expense of compression ratio
		0x200 = improved compression ratio with reduced speed
		0x300 = maximize compression ratio at expense of speed
		All other values are undefined.

Returns:

This function returns the compression performance setting.

B.1.3 eLZS_Decompress()

Include Files:

elzs.h

Syntax:

```
eLZS_len_type eLZS_Decompress (
    const void          *srcPtr,
    eLZS_len_type       srcCnt,
    void                *dstPtr,
    eLZS_len_type       dstBufSize,
    unsigned int         flags);
```

Description:

The function `eLZS_Decompress()` can be used to perform synchronous software-based eLZS decompression. The detailed declaration of this function is located in the file *elzs.h*.

The parameter `srcPtr` is a pointer to source data buffer whose length is specified by the parameter `srcCnt`. The argument `dstPtr` specifies the location of the destination buffer to hold the decompressed data. The parameter `dstBufSize` defines the maximum length of the destination buffer. Due to an internal requirement, the source or destination buffer length cannot exceed 3M.

When the function successfully returns, the destination buffer is filled with the decompressed data, and the return value contains the length of the decompressed data stored in the destination buffer. If the operation fails, this function returns an appropriate error status. The user should call the boolean macro `eLZS_IsError()` on the return value to determine if the operation succeeded or failed (see "Returns" below).

The parameter `flags` is used to modify the behavior of this function, as explained in [Table B-4](#) below. The flags may be logically ORed to set more than one flag option.

Table B-4. eLZS Decompression Flag Definitions

Flag Value	Description
0x001 = ELZS_FLAG_CRC32	Decompress and verify the CRC
0x002 = ELZS_FLAG_APPEND_CNT	Silently append consumed byte count to decompressed data
0x004 = ELZS_FLAG_EOCD_CONTINUE	Continue decompressing after EOCD
0x016 = ELZS_FLAG_ERR_UNUSED_SRC	If enabled, the <code>ELZS_ERROR_SRC_UNDERRUN</code> error will be returned if there are unused decompressed source bytes after the EOCD token. This flag is ignored if <code>ELZS_FLAG_EOCD_CONTINUE</code> flag is set.
All other values are undefined.	

Parameters:

srcPtr	Input	Pointer to the data to be decompressed.
srcCnt	Input	Number of source data bytes to be decompressed.
dstPtr	Input/Output	Pointer to where decompressed data will be stored.
dstBufSize	Input	Size in bytes of the destination buffer.
flags	Input	Flags used to modify the behavior of the function. See Table B-4 .

Returns:

The boolean macro `eLZS_IsError()` should be called on the return value of `eLZS-Decompress()` to determine the status of the decompress operation. If `eLZS_IsError()` returns false, the function completed without errors and the return value represents the number of bytes of decompressed data in the destination buffer. If `eLZS_IsError()` returns true, the destination buffer will not be filled with decompressed data and the return value will contain one of the error codes defined below.

Error	Value	Description
ELZS_ERROR_SRC_UNDERRUN	-1	Source data corrupted.
ELZS_ERROR_DST_OVERFLOW	-2	Not enough space in destination buffer to store the result
ELZS_ERROR_BAD_TOKEN	-3	Source data corrupted.
ELZS_ERROR_BAD_CRC	-4	CRC verification failed.
ELZS_ERROR_BAD_RAW_LEN	-5	Source data corrupted.
ELZS_ERROR_BAD_RAW_TOK	-6	Source data corrupted.

B.2 DRE_swHashSha256()

Include Files:

dre_api.h, dre_swlib_priv.h

Syntax:

```
DRE_status DRE_swHashSha256(
    const DRE_u08b    *src,
    DRE_u32b          srcLen,
    const DRE_u08b    *inIhv,
    DRE_u32b          ihvLen,
    DRE_u08b          *hashBuf,
    DRE_u64b          totalBytes,
    DRE_u32b          flag);
```

Description:

The function `DRE_swHashSha256()` can be used to perform synchronous software-based stateful and stateless SHA256 hash.

Stateful SHA256 hash operations require multiple calls to this function. The hash result for the intermediate calls are partial results that are then passed back as an input for the subsequent calls to this function. *Stateless* SHA256 hash operations are processed in a single pass.

The operation is controlled by the parameter `flag`, as shown in the table below.

Table B-5. swHashSha256 Flag Definitions

Flag Value	Description
DRE_PKT_FIRST	The first data block of a stateful SHA256 hash.
DRE_PKT_MIDDLE	The middle data blocks of a stateful SHA256 hash.
DRE_PKT_LAST	The last data block of a stateful SHA256 hash.
DRE_PKT_FIRST & DRE_PKT_LAST	The entire data block of stateless SHA256 hash.
All other values are undefined.	

The parameter `src` is a pointer to the source data buffer whose length is specified by the parameter `srcLen`. For stateful SHA256 operations, the source data buffer size must be a multiple of 64 bytes (the underlying hash iteration block size for SHA256) for the first and middle call to `DRE_swHashSha256()`, otherwise the error `DRE_ERR_SW_INVALID_ARG` will be returned.

The parameter `hashBuf` specifies the location of the destination buffer that holds the hash result. Depending on the setting of the parameter `flag`, the result is either a partial hash (if `flag` is `DRE_PKT_FIRST` or `DRE_PKT_MIDDLE`) or a final hash (if `flag` is `DRE_PKT_LAST`).

The parameter `inIhv` is the partial hash for a stateful SHA256 operation that is generated by the previous call to `DRE_swHashSha256()`. The partial hash result must be passed as `inIhv` in the following call to `DRE_swHashSha256()`. The partial hash length is denoted by `ihvLen`, and must be set to 32 bytes for SHA256.

The parameter `totalBytes` is required if `flag` contains `DRE_PKT_LAST`. The parameter `totalBytes` is used to pad the last data block of the hash.

Parameters:

src	Input	Pointer to the source data buffer to be hashed. This parameter cannot be null.
srcLen	Input	Number of bytes in the source data buffer. This parameter cannot be zero.
inIhv	Input	Pointer to the partial hash returned by the previous call to this function. This parameter is required for stateful operations if <code>flag</code> is set to <code>DRE_PKT_MIDDLE</code> or <code>DRE_PKT_LAST</code> , otherwise the error <code>DRE_ERR_SW_INVALID_ARG</code> will be returned.
ihvLen	Input	Number of bytes in the partial hash buffer. This parameter is required for stateful operations if <code>flag</code> is set to <code>DRE_PKT_MIDDLE</code> or <code>DRE_PKT_LAST</code> , otherwise the error <code>DRE_ERR_SW_INVALID_ARG</code> will be returned. The value of this parameter will always be 32 bytes for SHA256 hash.
hashBuf	Output	Pointer to the buffer that will hold the hash result. This parameter cannot be null.
totalBytes	Input	Total number of bytes to be hashed.
flag	Input	Flag that controls the hash operation. See Table B-5 .

Returns:

Call `DRE_IS_RESULT_ERR()` to determine the return status. If the macro returns `DRE_FALSE`, the operation finished successfully, otherwise, an error occurred.

Error	Description
DRE_OK	The command completed successfully without any errors.
DRE_ERR_SW_INVALID_ARG	The command failed due to an invalid parameter.

I Document Revision History

This section lists the additions, deletions, and modifications made to this document for each release of this document.

Document Revision A

Initial release.

Document Revision B

- Update 1.** Section 1.2.2.2 API Layer: corrected specific version of Intel Quick Assist API.
- Update 2.** Section 5.3 Hifn API Buffer Requirements: added alignment restrictions for the hash buffer.

Document Revision 00

- Update 1.** Chapter 2 Session Structure: reorganized this chapter and added more description of the RA and DO sessions.
- Update 2.** Chapter 4 System Considerations: added this new section.
- Update 3.** Section 5.2 Driver Configuration File: expanded the section to include recommendations for the drv config settings.
- Update 4.** Section 5.1.3 Retrieve the HW Information: updated the retrieved fields.
- Update 5.** Section 5.2.2 Retrieve the HW Information: updated the retrieved fields.
- Update 6.** Section 7.2.4 QuickAssist Example Implementations: added this new section.
- Update 7.** Section 7.3 Qaperf Application: added this new section.
- Update 8.** Chapter 8 Error Handling: added this new section.

Document Revision 01

- Update 1.** Added support for the DX 1845 card.
- Update 2.** Section 4.3.3 Driver Memory Allocation: corrected the equation for MemCmd.

Document Revision 02

- Update 1.** Section 2.4.1 Intel Quick Assist Crypto Algorithms: added note that prime number test is sw implementation only.
- Update 2.** Section 2.5 Support for 32-bit Applications: added this new section.
- Update 3.** Section 5.2.1 Host Initialization Settings: added new options for notification_mode. Added new parameters pp_malloc_mem_threshold and msi_interrupt_enable.
- Update 4.** Section 7.1.7.2 Global Config Settings: removed sentence that Raw PK always uses a single thread.
- Update 5.** Section 8.1 Definition of Error Status Codes: added Module Code 20, DRE_FLASH. Added Error code 61, DRE_C_ECPOINT_VERIFY_FAILED.
- Update 6.** Appendix A: added this new section.

Document Revision 03

- Update 1.** Added support for the Express DX 1825, 1835 cards throughout.

- Update 2.** Section 4.1.3 Scatter/Gather Memory Scheme: added paragraph describing the virtual buffers. In Figure 4-1, replaced data “buffer” with “fragments”.
- Update 3.** Section 4.4 API Buffer Requirements: added description about eLZS expansion.
- Update 4.** Section 5.2.3 Log File Settings: added description for null log file value.

Document Revision 04

- Update 1.** Corrected Performance Application Notes documents numbers to reflect the new API specific documents.
- Update 2.** Section 1.1.2 Operating System: added new supported operating systems.
- Update 3.** Section 2.2 Public Key (PK) Algorithms: updated ECDH and ECDSA support to all FIPS curves. Added EC point verify and multiply.
- Update 4.** Section 2.4 Intel Quick Assist Crypto Algorithms: updated ECDH and ECDSA support to FIPS186-3.
- Update 5.** Section 4.2.5 Single/Dual Command Ring: corrected error that driver has parameter used to select between dual and single command/result rings.
- Update 6.** Section 5.2.1 Host Initialization Settings:
- Update 7.** Section 6.2.3.5 Submit Data for PK Operation: replaced text that load balancing is no longer supported.
- Update 8.** Section 7.1 demo Application: removed functional portion from demo application.
- Update 9.** Section 7.1.5 PK Performance: added ECDH and ECDSA.
- Update 10.** Section 7.1.7.4 Public Key Configuration Settings: added 192, 224 pk_mbits values to ECDH and ECDSA.
- Update 11.** Appendix B: added this new section.

Document Revision 05

- Update 1.** Section 1.1.2 Operating System: added new supported operating systems.
- Update 2.** Section 1.2.3 Apps: added dx and sdemo.
- Update 3.** Section 2.2 Public Key (PK) Algorithms: added DSA key length (3072,384).
- Update 4.** Section 2.3 Random Number Generator (RNG): removed text about PRNG.
- Update 5.** Section 4.3 API Buffer Requirements: removed the addr alignment requirements for the buffers.
- Update 6.** Section 5.2.3.11 RNG: removed text about PRNG.
- Update 7.** Section 5.2.3 Log file Settings: added log_file_size.
- Update 8.** Section 7.1.4 RNG: removed text about PRNG.
- Update 9.** Section 7.1.5 PK Performance: added DSA bits 2048, 3072.
- Update 10.** Section 7.1.7.4 Public Key Configuration Settings: added DSA bits 2048, 3072. Updated default value for pk_mbits.
- Update 11.** Section 7.2 sdemo App: added this new section.
- Update 12.** Section 7.5 Diagnostic Tools: added this new section.
- Update 13.** Appendix A: removed all text and section relating to the PRNG.

Document Revision 06

- Update 1.** Section 5.2.1 Host Initialization Settings: added the parameters `pcie_error_recovery_enable` and `pcie_error_interval_threshold`. Changed default setting of failover and RTV to enabled.
- Update 2.** Section 7.2.1.2 Transform settings: added PASSTH to algo. Added new parameter `comp_mode`, `crc_config`, `cmd_target`.
- Update 3.** Section 7.5.2 `dx_status` Diagnostic Tool: multiple instances may be run.
- Update 4.** Section 8.2 Failover: clarified when sw failover will occur.
- Update 5.** Section 8.4 Single Command Error Handling: updated flow diagram to remove failover.
- Update 6.** Section 8.6 PCIe Error Handling: added this new section.
- Update 7.** Section 8.7 Data Corruption Error Handling: added this new section.

Document Revision 07

- Update 1.** Section 1.2.1 Operating System: removed SUSE9 and added CentOS 6.2 and 6.3.
- Update 2.** Section 5.2.1 Host Initialization Settings: modified the upper bound for `max_key_num`. Added two new parameters: `rng_bit_rate`, `rng_sample_rate`.
- Update 3.** Section 7.2.1.2 Transform settings: added option 32 to `cmd_target`.
- Update 4.** Section A.2.1 Hardware RNG: added reference to proc entry.
- Update 5.** Section A.6 Monitoring the RNG Statistics: added this new section.

Document Revision 08

- Update 1.** Removed references to QuickAssist throughout.
- Update 2.** Section 1.2.1 Operating System: added SUSE9 and Cent OS 5.5.
- Update 3.** Section 1.2.3 Applications: change `dx` to `dx_monitor` and `dx_diag`.
- Update 4.** Section 2.2 Public Key (PK) Algorithms: added Miller-Rabin Test.
- Update 5.** Section 4.2.1 Scatter/Gather Memory Scheme: removed the restriction that the virtual buffers must point to physically contiguous memory for kernel mode apps. Replaced formulas for calculating the total buffer size.
- Update 6.** Section 4.2.3 Driver Memory Allocation: added rough estimate for calculating the memory required.
- Update 7.** Section 4.4 Performance Considerations: added note about bundled commands.
- Update 8.** Section 5.1 Initialization Sequence: added note about `pk_enable` to step 9.
- Update 9.** Section 5.2.1 Host Initialization Settings: added `pk_enable`. For `notification_mode`, added that threads are spawned for each device. Added note about perf degradation for `pcie_error_recovery_enable`.
- Update 10.** Section 5.2.2 Command Structure Settings: modified description for `data_desc_per_cmd`.
- Update 11.** Section 5.2.3 Log file Settings: added explanation for when a value is not entered for `log_file_name`.
- Update 12.** Section 5.2.4 Temperature Sensor Settings: added the valid range for `normal_temp` and `over_heat_temp`.
- Update 13.** Section 7.1.5 DRBG: added this new section.

- Update 14.** Section 7.1.8.1 Test Configuration Settings: added the parameter `test_raw_drbg`.
- Update 15.** Section 7.1.8.3 Raw Acceleration and Data Offload Configuration Settings: removed the user mode restriction on parameter `pkt_size`. Added `HASH_SHA384` and `HMAC_SHA384` to the parameter `seg_hash_algo`. Added new parameter `cmd_bundle_num`.
- Update 16.** Section 7.1.8.6 DRBG Configuration Settings: added this new section.
- Update 17.** Section 7.2.1.2 Transform settings: added `PASSTH` to `algo`. Added new parameter `comp_mode`, `crc_config`, `cmd_target`. Added `HASH_SHA384` and `HMAC_SHA384` to the list of valid values for `seg_hash_algo`.
- Update 18.** Section 7.5 Diagnostic Tools: split this section into 3 sub-sections to reflect the new diag tools that split the `dx` tool into `dx_monitor` and `dx_diag`.
- Update 19.** Section 8.1 Definition of Error Status Codes: added new error codes `DRE_C_NEED_RESEED` and `DRE_C_NOT_ENABLED`.
- Update 20.** Section 8.1.2 Error Flags: added this new section.
- Update 21.** Section 8.2 Failover: modified description of failover.
- Update 22.** Section 8.3 Single Command Error Handling: split this section into PP and PK error handling. Created new flowchart for PK error handling.
- Update 23.** Section 8.5 Overheated Condition Error Handling: added text about flashing the error LED.
- Update 24.** Section A.3 Software Implementation: added support for `CTR_DRBG` and `Dual_EC_DRBG`.
- Update 25.** Section A.3.1 Retrieve a Nondeterministic RNG Value from a 820x Device: removed text saying the 820x will be put in an unhealthy state after a RNG error.
- Update 26.** Section A.3.3 Monitoring the RNG Statistics: removed instructions to start the process.



A New Direction in Mixed-Signal

48720 Kato Road
Fremont, CA 94538
p: 510.668.7000
www.exar.com

Exar Confidential