DX

Software Development Kit

User Guide

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

List of Tables

Preface

About This Document

Welcome to the DX Software Development Kit (SDK) User Guide. This document describes the operation and features of the DX SDK version 2.1.0L release.

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

- XR9240 coprocessor
- DX2040 card

Please note that in the context of this document, references to the 9240 device and the DX card are interchangeable. Note that the SDK software refers to the XR9240 device as “92xx” to accommodate future variations of the device.

Audience

This document is intended for:

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

Prerequisites

Before proceeding, you should generally understand:

- Compression algorithms LZS, Deflate, gzip, zlib
- Advanced Encryption Standard (AES), Triple Data Encryption Standard (3DES), ARC4, and their modes of operation
- Cryptographic hash functions SHA, MAC, HMAC, GMAC, CMAC, XCBC, SSLv3
- The functionality of the XR9240 coprocessor and the DX2040 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 DX SDK.

Chapter 2, “Features” gives an overview of the operations supported by the DX SDK.

Chapter 3, “Session Structure” introduces the concept of a session as it applies to the 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 DX card using the 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 Random Number Generator” provides a more detailed description of the 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.

DX SDK Release Notes, RLN-0009

DX SDK Getting Started Guide, USR-0038

Raw Acceleration API 2.1 Reference Guide, USR-0040

DX SDK 2.1.0L Raw Acceleration API Performance Application Note, APN-0006

XR9240 Compression and Security Coprocessor Data Sheet, DAT-0001

DX2040 Compression and Security Acceleration Card Data Sheet, DAT-0009

DX2040 Compression and Security Acceleration Card & XR9240 Compression and Security Coprocessor Errata, ERR-0005

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
AER	Advanced Error Reporting
AES	Advanced Encryption Standard
API	Application Programming Interface
CBC	Cipher Block Chaining encryption mode
CMAC	Cipher-based Message Authentication Code
CTR	Counter encryption mode
DES	Data Encryption Standard
DIF	Data Integrity Field
DSA	Digital Signature Algorithm
DSD	Device Specific Driver
ECB	Electronic Codebook encryption mode
ECDH	Elliptic curve Diffie Hellman
ECDSA	Elliptic Curve DSA
ESF	Exar Service Framework
eLZS	Enhanced Lempel-Ziv-Stac Compression
GCM	Galois Counter Mode
GMAC	Galois Message Authentication Code
HIV	Hash Initialization Vector
HMAC	Hash Message Authentication Code
IOMMU	Input/Output Memory Management Unit
IV	Initial Vector
LZS	Lempel-Ziv-Stac Compression
MD5	Message Digest 5
OSAL	Operating System Abstraction Layer
PK	Public Key
PP	Packet Processor
RNG	Random Number Generator
SAI	Service Assistant Infrastructure
SDK	Software Development Kit
SHA	Secure Hash Algorithm

Term	Definition
SSLv3	Secure Sockets Layer Version 3
XCBC	eXtended Cipher Block Chaining
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 DX SDK interfaces to the user application and the Exar device(s).

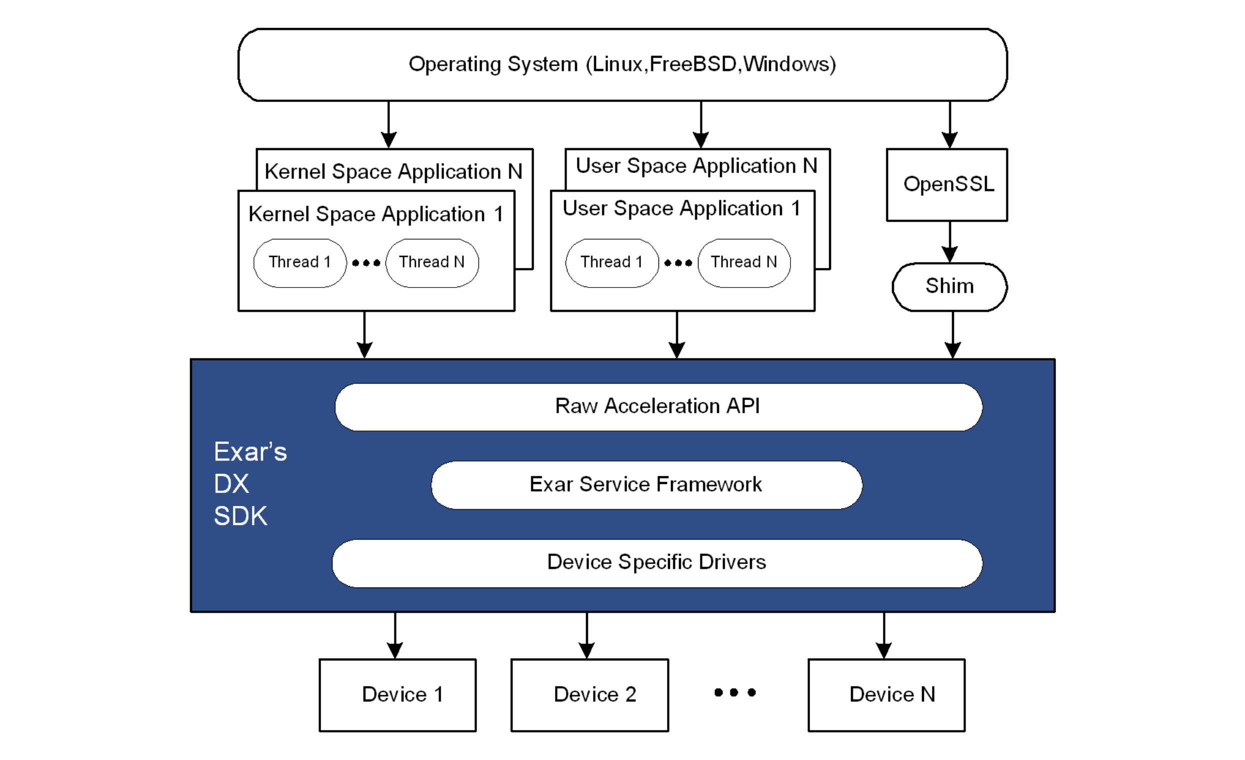


Figure 1-1. High Level DX SDK System Model

The DX SDK currently supports the following Linux distributions:

- Red Hat Enterprise Linux 6.0 (Kernel 2.6.32-71.el6 for x86 64-bit)
- Red Hat Enterprise Linux 6.2 (Kernel 2.6.32-220.el6 for x86 64-bit)
- Red Hat Enterprise Linux 6.3 (Kernel 2.6.32-279.el6 for x86 64bit)
- Red Hat Enterprise Linux 6.4 (Kernel 2.6.32-358.el6 for x86 64-bit)
- CentOs release 6.1 (Kernel 2.6.32-131.0.15.el6 for x86 64bit)
- CentOs release 6.2 (Kernel 2.6.32-220.el6 for x86 64bit)
- CentOs release 6.3 (Kernel 2.6.32-279.el6 for x86 64bit)
- CentOS release 6.4 (Kernel 2.6.32-358.el6 for x86_64)
- SUSE Linux Enterprise Server 10 SP3 (Kernel 2.6.16.60-0.54.5-smp for x86 64-bit)
- SUSE Linux Enterprise Server 11 SP1 (Kernel 2.6.32.36-0.5-default for x86 64-bit)
- SUSE Linux Enterprise Server 11 SP2 (Kernel version 3.0.10 for x86 64-bit)

- Fedora 19 (Kernel version 3.9.4 for x86 64-bit)

The supported hardware platforms are:

- x86_64

Future releases of the DX SDK are expected to support Windows and FreeBSD operating systems.

1.1 Software Architecture

Figure 1-2 illustrates the detailed architecture of the DX SDK.

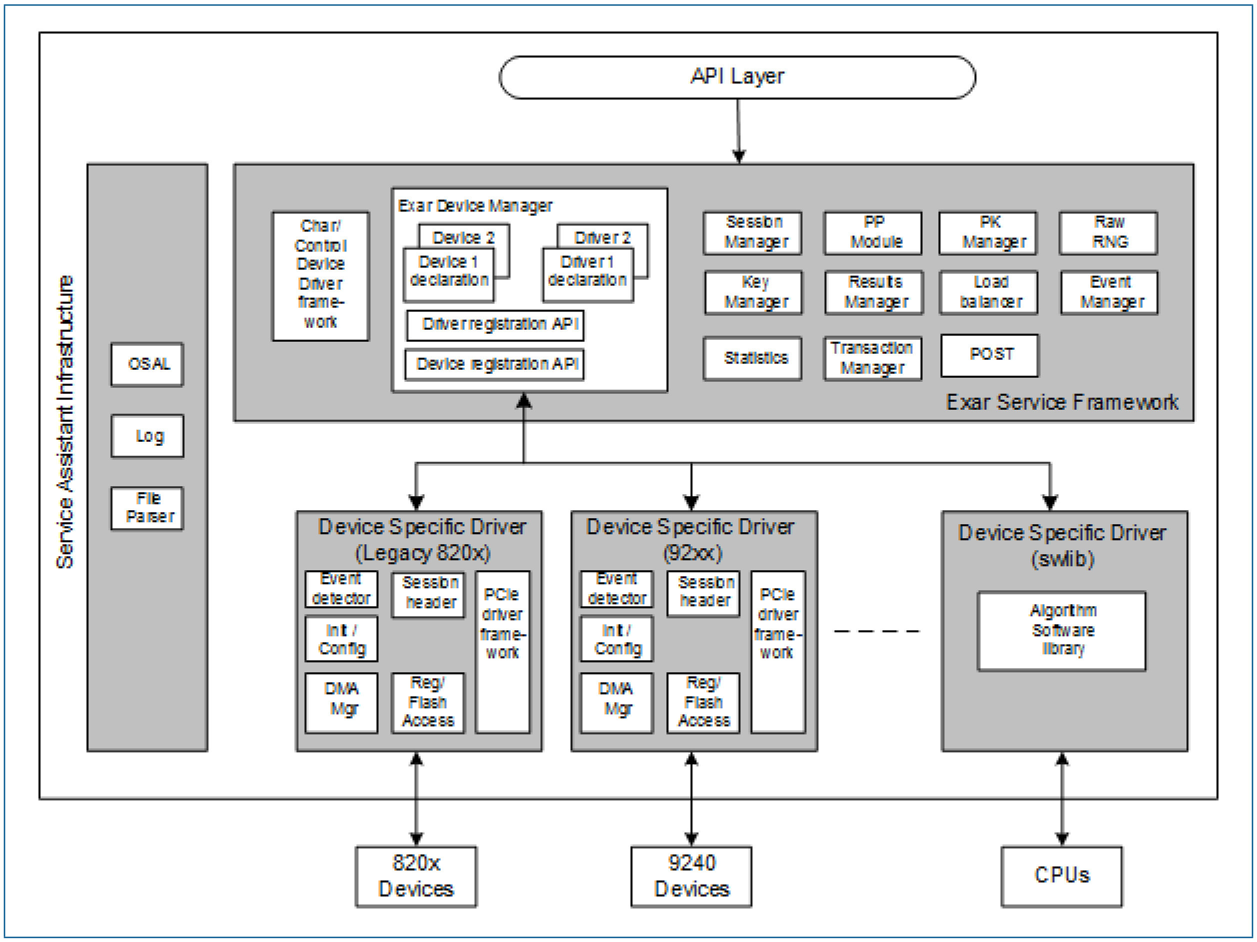


Figure 1-2. Detailed Software Architecture

1.2 Software Modules

This section briefly describes the software modules that comprise the DX SDK in more detail.

1.2.1 Service Assistant Infrastructure (SAI)

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

1.2.2 API Layer

Exar's DX SDK provides the Raw Acceleration (Raw) API to interface to the user application. The Raw Acceleration API leverages all functionality of Exar XR9240 device, including compression, encryption, authentication, RNG and PK operations. Please refer to the *Raw Acceleration API Reference Guide*, USR-0040, for detailed syntax and usage.

1.2.3 Exar Service Framework (ESF)

Exar Service Framework (ESF) provides the algorithm acceleration for the API Layer. All chipset-independent code is located in the Exar Service Framework, while all chipset-dependent code is located in the Device Specific Driver. The Exar Service Framework (ESF) module manages all sessions, keys and devices that were registered with the Device Specific Driver, thus enabling the hardware acceleration and software library operations.

The ESF retrieves operations from the API layer, translates those operations into hardware commands, submits the commands to the hardware, retrieves the completed commands, and returns the completed operations to the API layer. The ESF also manages the load balancing, session context and key pool.

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

1.2.4 Device Specific Driver (DSD)

The Device Specific Driver (DSD) is a chipset-dependent module. The Device Specific Driver provides a unified hardware interface to the Exar Service Framework (ESF). The DSD converts each device's specific structure format to a uniform structure for the ESF.

Currently, the DX SDK supports a DSD for the XR9240 coprocessors, and a DSD for the software library, and a DSD for the legacy 820x processors. A single instance of a DSD will manage all devices of that class installed in the system. In other words, one instantiation of the XR9240 DSD module will manage all XR9240 coprocessors installed on all DX cards in the system.

1.2.5 Software Library

The Software library executes the operations, such as compression, authentication, encryption and public key operations in software. The Software library is implemented as a Device Specific Driver in order to simulate the hardware if an XR9240 device errored or is recovering from an error, or if there are no operable Exar devices in the system.

1.3 Applications

The DX SDK includes the following applications to demonstrate the capabilities of the XR9240 coprocessor.

demo

> The demo application can be used to demonstrate the functionality of the Exar XR9240 device and its API-based system performance.

example

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

sdemo

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

dx_monitor, dx_status, and dx_diag

> The dx_monitor, dx_diag and dx_status applications are debugging tools that can be used to verify the DX card’s or XR9240 coprocessor’s status and performance.

2 Features

This section describes the features supported by the XR9240 DX SDK.

2.1 Symmetric Key Algorithms

2.1.1 Compression/Decompression Algorithms

The following compression and decompression operations are supported:

- Stateless LZS
- Stateless eLZS with support for anti-expansion
- Stateless and stateful gzip (RFC1952)
- Stateless and stateful Deflate (RFC1951)
- Stateless and stateful zlib (RFC1950)

2.1.2 Encryption/Decryption Algorithms

The XR9240 DX 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-GCM-256
    - AES-XTS-256, AES-XTS-512
- 3DES-CBC
- DES (supported via 3DES API with K1=K2=K3)
- ARC4

Stateful encryption for CBC, CTR, GCM mode transforms are supported by storing the next IV in the Host-based session state. The DX SDK also supports an IV replacement feature. The IV replacement feature frees the application from maintaining the IV by enabling the XR9240 device to generate a random, unpredictable IV. Refer to the *Raw Acceleration Reference Guide*, USR-0040, for detailed information

2.1.3 Authentication Algorithms

The authentication algorithms supported are:

- SHA1, SHA256, SHA224, SHA384, SHA512

- MD5
- HMAC-SHA1, HMAC-SHA256, HMAC-SHA224, HMAC-SHA384, HMAC-SHA512, HMAC-MD5
- SSLMAC-SHA1, SSLMAC-SHA256
- AES-GMAC-128, AES-GMAC-192, AES-GMAC-256
- AES-XCBC-128, AES-XCBC-192, AES-XCBC-256

2.2 Public Key (PK) Algorithms

The Public Key algorithms supported are:

- Diffie-Hellman
  - DH shared secret generation
  - Supports key sizes 1024 through 4096 bits
- Rivest-Shamir-Adleman
  - RSA encryption and decryption
  - RSA sign and verify
  - Supports key sizes 1024 through 4096 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 (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 XR9240 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.

3 Session Structure

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

A session is an information exchange between the application and the 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.

3.1 Terminology

Command — A command consists of a source buffer which contains the input data, a destination buffer which stores the output data, and an optional authentication buffer which stores the hash result, as well as the required command specific parameters such as the IV or AAD. Commands are grouped together to form a session.

Session — A session defines the operation or transform that will be applied to the commands in the session. Each session is represented by a set of configurable parameters that will apply to all commands in that session.

All Packet Processing commands must be associated with a session. A session must be opened by the user before PP commands may be submitted to the hardware. Public Key commands are not required to be associated with a session.

Packet — A packet is an internal SDK construct that combines the user supplied session and command parameters into a usable structure for the SDK.

3.2 Raw Acceleration Sessions

As shown in Figure 3-1, Raw Acceleration sessions are self-contained. The algorithm parameters are configured for each command of data submitted to that session. A Raw Acceleration session may be configured to perform compression, padding, encryption, and/or authentication, however all commands in a particular session must be the same algorithm. The session information such as the algorithm type, algorithm mode, IV, will be shared by all commands in that session.

A Raw session does NOT buffer the source data or destination result data. After the packet has been processed, the packet information is no longer available to the SDK.

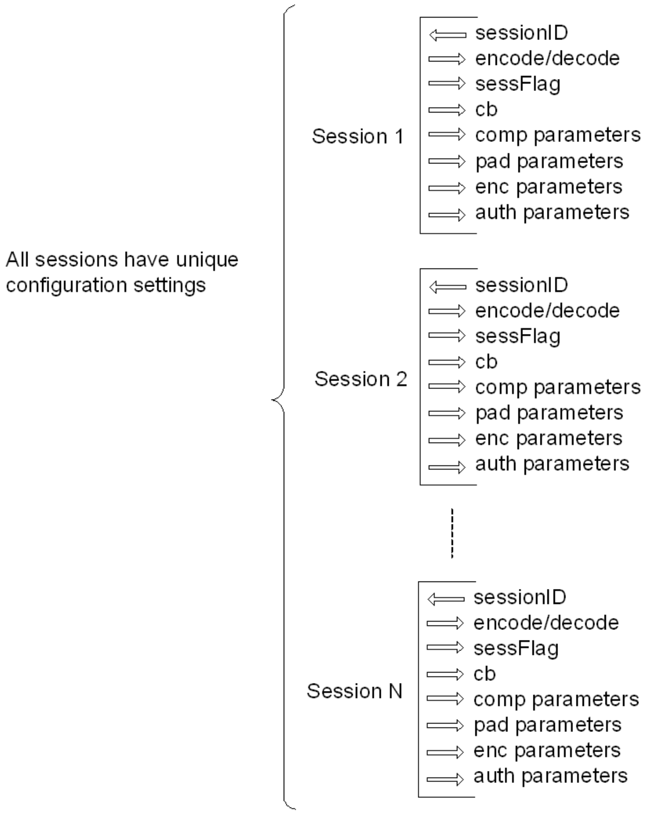


Figure 3-1. Visual Representation of a Raw Acceleration Session

3.2.1 Raw Acceleration Session Model

A Raw Acceleration session is modeled after a command processing operation as illustrated in Figure 3-2. The user calls a command submittal function to submit a command 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. Commands may be submitted synchronously or asynchronously to a Raw Acceleration session.

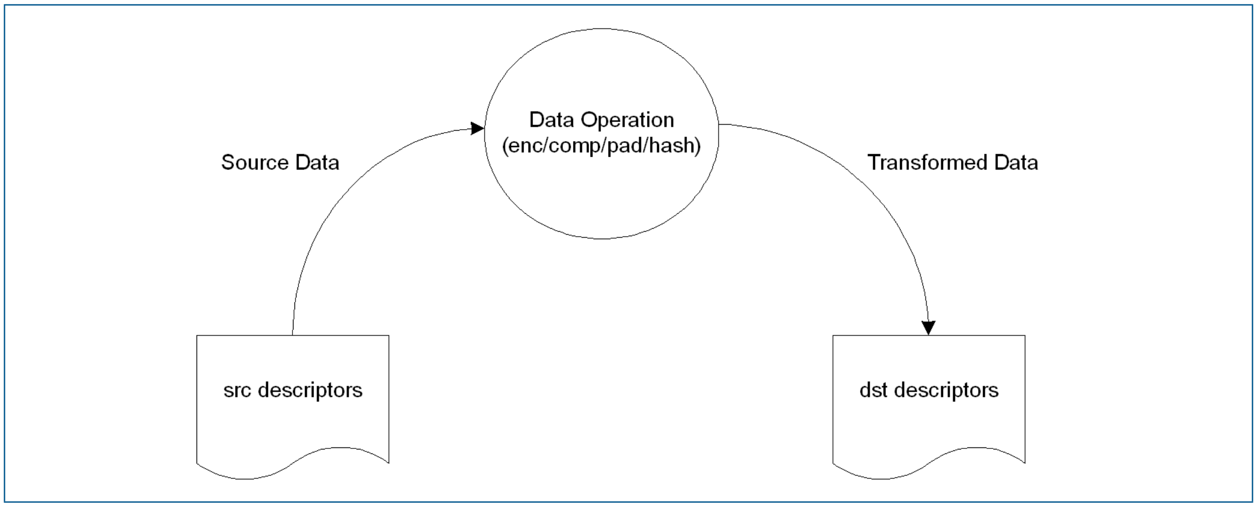


Figure 3-2. Raw Acceleration Session Model

3.2.2 Stateful Sessions

The Raw Acceleration API allows for stateful operations. A stateful operation is achieved by setting the Boolean parameter `stateful` to TRUE in the compression, encryption, or authentication session management descriptors when the session is opened. Stateful sessions do not support a combination of transforms such as compression + encryption in the same session. However, multiple engines of the same transform type may be enabled for a single stateful session.

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 submitting the last command of a stateful session.

Stateful commands execute one after another because the result of current command will be used by next command. Typically, stateful commands are submitted using the synchronous API function call. If a stateful command is submitted using the asynchronous API function call, the SDK will maintain an internal command list, and only submit a new command to the hardware after the current command completes.

When in stateful mode, the SDK will automatically save the relevant session information across commands with the session. For stateful deflate/zlib/gzip compression, the history is saved across the commands. For stateful encryption operations, the initialization vector (IV) is saved across the commands in a stateful session. For stateful hash operations, an intermediate hash value (IHV) is saved across the commands. 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.

For DX SDK versions 2.1.0L and later, if a single command in a synchronous mode stateful session fails, the application may re-submit only the failed command if the failure is considered recoverable, such as an overflow error or ring-full error.

4 System Considerations

This chapter discusses system issues that should be considered before installing and using the 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 XR9240 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*, USR-0040, for a detailed description of the Raw Acceleration API synchronous and asynchronous modes.

4.1.2 Kernel/User Mode

The system environment will dictate whether the DX SDK will operate in user mode or kernel mode. The DX SDK has been designed to link with applications running in either user mode or kernel mode.

4.1.3 Single/Multi-Threaded Applications

The Raw Acceleration API supports single and multi-threaded applications. The decision to use single or multi-threaded applications will be based primarily on the system architecture. Multi-threaded applications will typically yield improved performance.

4.1.4 Interrupt Modes

The XR9240 device supports MSI-X, MSI and legacy interrupt modes. Versions 2.x.x of the DX SDK and later automatically select the interrupt mode in the sequence:

MSI-X -> MSI -> Legacy

based on the interrupts modes supported by the hardware.

4.2 Memory Considerations

4.2.1 Scatter/Gather Memory Scheme

The XR9240 and 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.

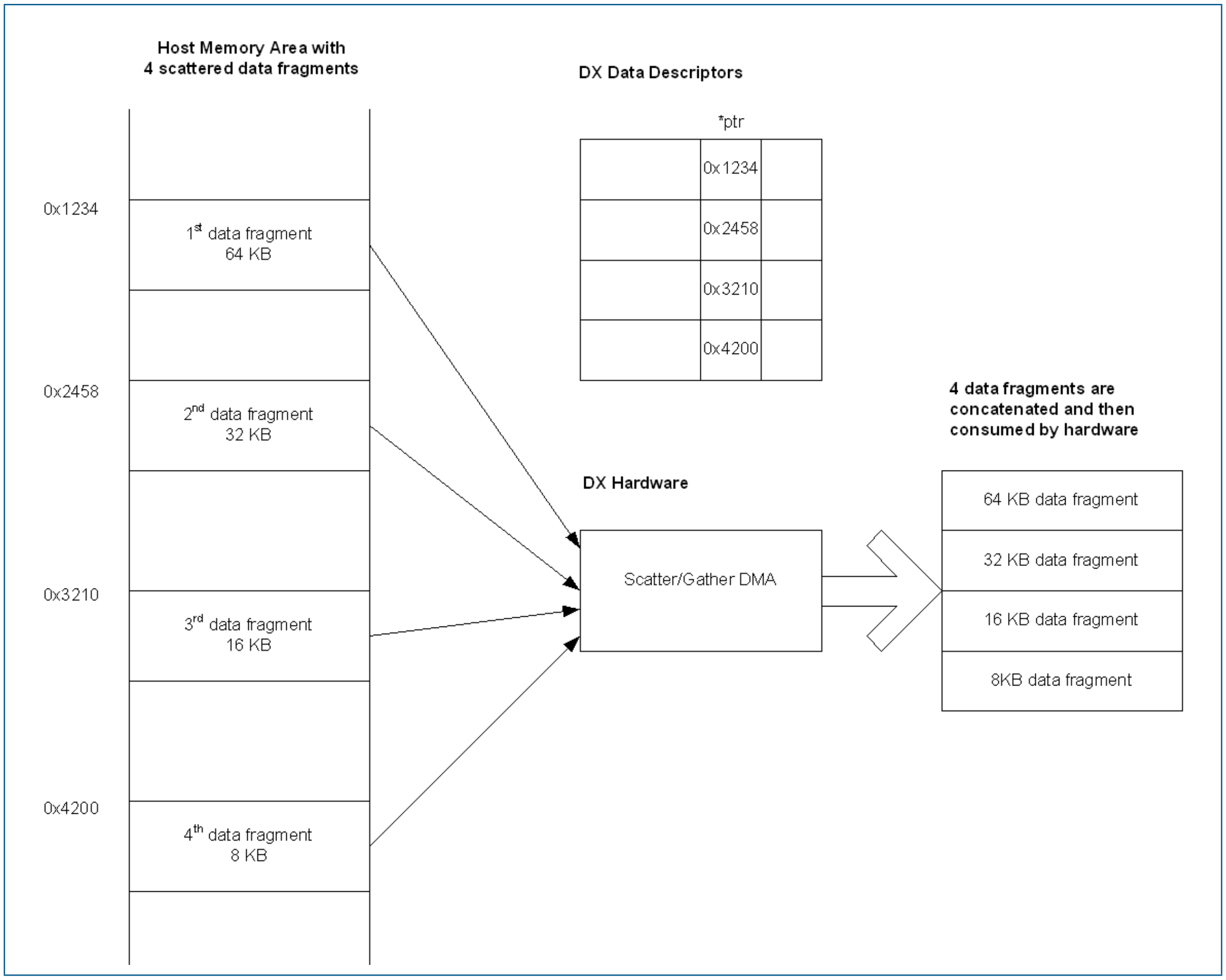


Figure 4-1. Scatter/Gather Illustration

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 XR9240 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 400 MB. Submitted commands whose buffer sizes exceed 400 MB 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 hardware command structure.

Refer to the *XR9240 Data Sheet*, DAT-0001, for more details on the command structure and data descriptors.

4.2.2 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 XR9240 devices. The driver load may fail if there is not enough DMA coherent memory in the system.

For DX SDK versions 2.1.0L and later, the required amount of memory may be calculated using the formula:

> REQUIRED_MEM_SIZE = Nd * Nr * Nn * Np * C
>
> Where:
>
> > Nd = Number of XR9240 devices
> >
> > Nr = Number of rings configured per device (default is 12 PP rings + 4 PK rings)
> >
> > Nn = Number of Non-Uniform Memory Access (NUMA) nodes in the system
> >
> > Np = Size of the per-ring and per-NUMA-node command pool (1024 by default)
> >
> > C = A constant, set to 6 KB

4.3 API Buffer Requirements

There are three types of data buffers used in the Exar XR9240 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(), and DRE_rawSessSubmitAsync().

The array of destination buffers is specified with the parameter **dst** in the Raw Acceleration functions DRE_rawSessSubmitSync(), and DRE_rawSessSubmitAsync().

The hash buffer is specified with the parameter **hash** in the Raw Acceleration functions DRE_rawSessSubmitSync(), and DRE_rawSessSubmitAsync(). The hash buffer size must be set large enough to store hash result. For example, the hash buffer must be not smaller than 64 bytes for the hash algorithm SHA512.

4.3.1 Alignment Requirements

The XR9240 devices have no alignment requirements.

4.3.2 Expansion 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:

Maximum additional bytes = 1 / 8 * (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:

Maximum additional bytes = 4 * (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 Exar 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 Exar 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 Exar device's pad engine to achieve the proper padding.

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 packet performance is better in kernel mode than it is in user mode.

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 includes the Service Assistant Infrastructure (SAI), Exar Service Framework (ESF), and Device Specific Drivers (DSDs). The Driver module initializes the Exar XR9240 devices and provides services to the applications.

After reading this document, refer to the *DX SDK Getting Started Guide*, USR-0038, 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 device.

1. Initialize the SAI layer.
2. Parse the XML driver configuration file.
3. Initialize the Exar Service Framework Module.
4. Probe for Exar 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 failover ("failover=1") in the driver, all data operations will be sent to the software library for processing. If no Exar 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 Exar device's register memory space. Each probed Exar device and DX card will be registered with the ESF device manager module.
6. Reset all Exar devices.
7. Ignore the first sixteen 32-bit words of the RNG, and then start the RNG continuous test.
8. 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.
9. Configure the hardware DMA rings.
10. Load the public key firmware into the hardware if public key operations are enabled (pk_enable = 1 in the driver configuration file).
11. Configure the hardware for PK (if enabled) and RNG operations.
12. Register the interrupt service routine to the OS if necessary.
13. Run POST.

After the driver is successfully installed, the Exar 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 reviewed and modified to reflect the user 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 Exar device should use to notify the host driver of completed packet processing (PP) and Public Key (PK) commands and how the host driver processes the completed commands. The *notification_mode* options are:

0 = Interrupt mode using a kernel thread

1 = (obsolete)

2 = (obsolete)

3 = Interrupt mode using a tasklet (DPC)

For options 0 and 3, the XR9240 device will interrupt the host driver when a command completes. For option 0, a kernel thread is spawned for each XR9240 ring to handle the interrupt and then notifies 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.

The default value for *notification_mode* is interrupt mode using a tasklet (option 3).

failover

The parameter *failover* determines the SDK behavior in the unlikely event of all Exar devices or DX cards failing. The two options are: enable failover and disable failover.

If failover is enabled and all hardware devices have failed, further submitted commands will be processed by the software library, 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 enabled.

real_time_verification

The parameter *real_time_verification* enables or disables real time verification of the compression, authentication and encryption engines. The two options are: enable real time verification and disable real time verification.

The default value for real_time_verification is real time verification is enabled.

load_balance_algorithm

The parameter *load_balance_algorithm* sets the command ring load balancing algorithm that the SDK will uses for all cards/devices. The options are: round-robin, queue depth, and CPU ring-binding.

In the round-robin scheme, the SDK sequentially assigns the data operation commands to all rings in the system. When the last ring in the sequence is reached, the SDK assigns the next command to the first ring 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 depth scheme, the SDK considers each ring's current command queue depth before assigning the next command. For example, if a system has two rings and ring A's queue has 10 commands to process, and ring B's queue has 20 commands to process, then the SDK will assign the next command to ring A.

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

In the CPU ring-binding scheme, the DX SDK automatically binds the CPUs to the rings. For example, if a system has 16 CPU cores and a single XR9240 device that is configured with 16 rings, the DX SDK will bind 16 CPU cores to the 16 rings, one to one. Threads running on CPU #0 will submit commands to ring #0, threads running on CPU #1 will submit commands to ring #1, and so on. The CPU ring-binding scheme reaches maximum performance when the packet size is smaller than 1024 bytes because it reduces competition and lock between the CPU cores.

Ultimately, the operating environment and application specifics will determine the best load-balancing algorithm to use, and customers are encouraged to experiment with all 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.

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 SDK maintains a separate key table with "max_key_num" entries for the symmetric and PK keys.

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 due to memory restrictions.

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 sessions. The minimum value for this parameter is 2 and the maximum value is 32M.

The SDK dynamically allocates the sessions as entries in a table. 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. Typically, the size of a RAW session structure is less than 1K bytes. Sessions that specify stateful compression or decompression may require additional memory for the history buffer.

The default value for *max_session_num* is 4096.

pp_statistics_enable

The parameter *pp_statistics_enable* indicates whether to enable or disable gathering packet processing statistics. 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*, USR-0040.

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

pp_malloc_mem_threshold

This legacy parameter is ignored for DX SDK versions 2.0.0L and later.

pk_enable

The parameter *pk_enable* controls the clock and power 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.

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. This parameter is only valid when *pcie_error_recovery_enable* = 1.

The SDK will attempt to recover a single PCIe error within the time interval defined by this parameter. 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 XR9240 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.

cpu_dma_zero_latency

The parameter *cpu_dma_zero_latency* controls the amount of CPU DMA latency. Enabling this parameter will increase performance for systems where the CPU enters an idle C-state.

Enabling this parameter increases power consumption significantly. Applications that do not experience reduced performance should disable this parameter to conserve power.

The default value for *cpu_dma_zero_latency* is disabled.

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 supports multiple command rings, so this parameter sets the maximum number of commands for all enabled command rings (the value will be applied to all rings).

For optimum command processing throughput, the command ring should never be empty. Selecting too small of a value for *cmds_per_ring* may limit the performance because the software command submittal thread cannot sustain the hardware. 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 errors occur too frequently, it is an indication that the value for *cmds_per_ring* is set too small.

The valid values for *cmds_per_ring* are:

32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536

The default value for *cmds_per_ring* is 4096.

data_desc_per_cmd

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 discontiguous 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 not saved to the log file.

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.

5.2.4 Temperature Sensor Settings

temp_over_enable

The parameter *temp_over_enable* enables or disables over-temperature protection in the XR9240 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 maximum temperature at which the XR9240 can operate normally. If the XR9240 temperature exceeds the normal temperature and then returns to a temperature less than or equal to the value for *normal_temp*, the XR9240 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 XR9240 device to not operate correctly. If the XR9240 exceeds this temperature, the XR9240 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 Components

This section describes the Driver components in more detail.

5.3.1 Service Assistant Infrastructure

5.3.1.1 OS Abstraction Layer (OSAL)

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. The OSAL APIs can be called by other sub-modules within the SDK.

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

5.3.1.2 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.

5.3.1.3 XML File Parser

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

5.3.2 Exar Service Framework

The Exar Service Framework (ESF) contains functional modules that assist the API Layer and internal modules.

5.3.2.1 Session Manager Module

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

5.3.2.2 Packet Processing Module

The Packet Processing Module performs the pre-processing tasks before a packet is submitted to the hardware and the post-processing tasks after a packet is returned by the hardware.

5.3.2.3 Load Balancing Module

The Load Balancing module executes the load balancing algorithm defined in the driver configuration file.

5.3.2.4 Device Manager Module

The Device Manager module registers and stores all Exar devices probed by the DSD.

5.3.2.5 Results Retrieval Module

The Results Retrieval module reads the results from commands completed in hardware from the DSD.

5.3.2.6 Event Manager Module

The Event Manager module handles all events reported by the DSD, such as PCIe related errors, temperature over-heating, etc.

5.3.2.7 Key Manager Module

The Key Manager module provides centralized key storage and index management for both symmetric keys and public keys. The maximum number of keys that may be stored in the Key Manager is configured in the driver configuration file.

5.3.2.8 User Space Transaction Manager Module

This module provides user space service to the user space API layer.

5.3.2.9 Public Key Manager Module

The PK Manager works with the DSD module to provide PK algorithm acceleration to the API layer.

5.3.2.10 Raw RNG Module

The Raw RNG module works with the DSD module to provide RNG acceleration to the API layer.

5.3.3 Device Specific Driver Module

The Device Specific Driver (DSD) probes and initializes all Exar devices and registers them with the Exar Service Framework module. The DSD provides a uniform interface to the ESF among the various Exar devices.

5.3.3.1 Linux PCIe Driver Module

The Linux PCIe Driver module probes all generic PCIe devices and provides a driver framework for each PCIe device.

5.3.3.2 Initialization and Configuration Module

This module initializes all probed Exar devices and configures them according to the driver configuration file.

5.3.3.3 Register Access Module

The Register Access module performs register accesses for the other modules within the DSD.

5.3.3.4 Flash Access Module

The Flash Access module performs access to the Flash for the DSD modules.

5.3.3.5 DMA Manager Module

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

6 Operation

The DX SDK package extrapolates the low-level management and use of the XR9240 device from the user application program. The DX SDK automatically and transparently identifies the Exar 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 DX SDK through the Raw Acceleration API. For detailed syntax and format of the Raw Acceleration API, please refer to the Exar *Raw Acceleration API Reference Guide*, USR-0040.

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

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 Exar 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 = DRE_MAX_CARD_NUM;

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_rawSymKeySet(). 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);
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 commands 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().

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.

The pseudo code may look like:

```
DRE_rawSyncOpData synOpData;
DRE_dataDesc src, dst;
```

```
//set up the opData
……
//set the src & dst data descritpors
… …

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

PK commands are not required to be associated with a session and may be submitted directly to the hardware. 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_rawSymKeySet() should be destroyed using the function DRE_rawSymKeyDestroy() when the user no longer requires that key.

The pseudo code may look like:

```
status= DRE_rawSymKeyDestroy(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).

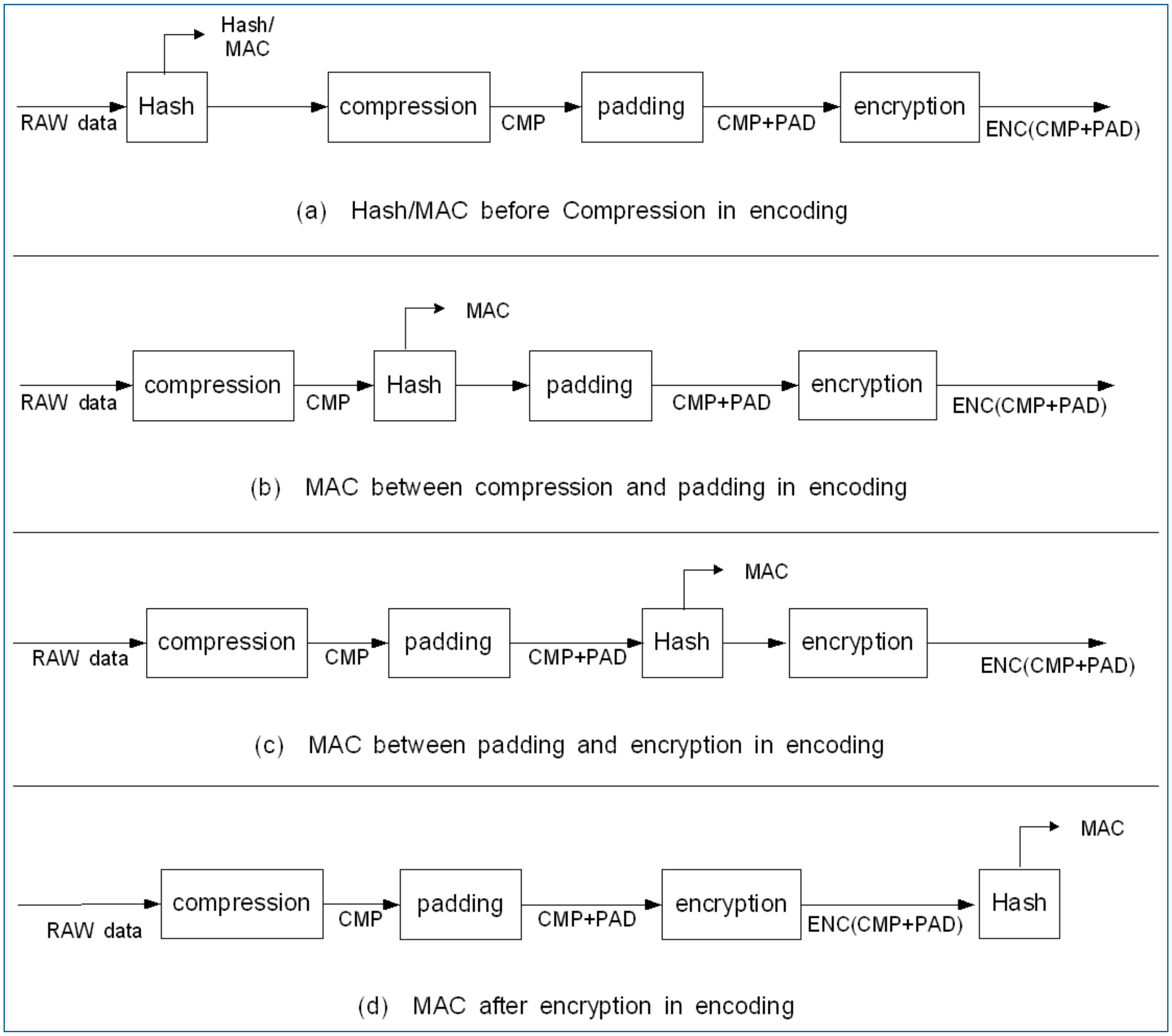
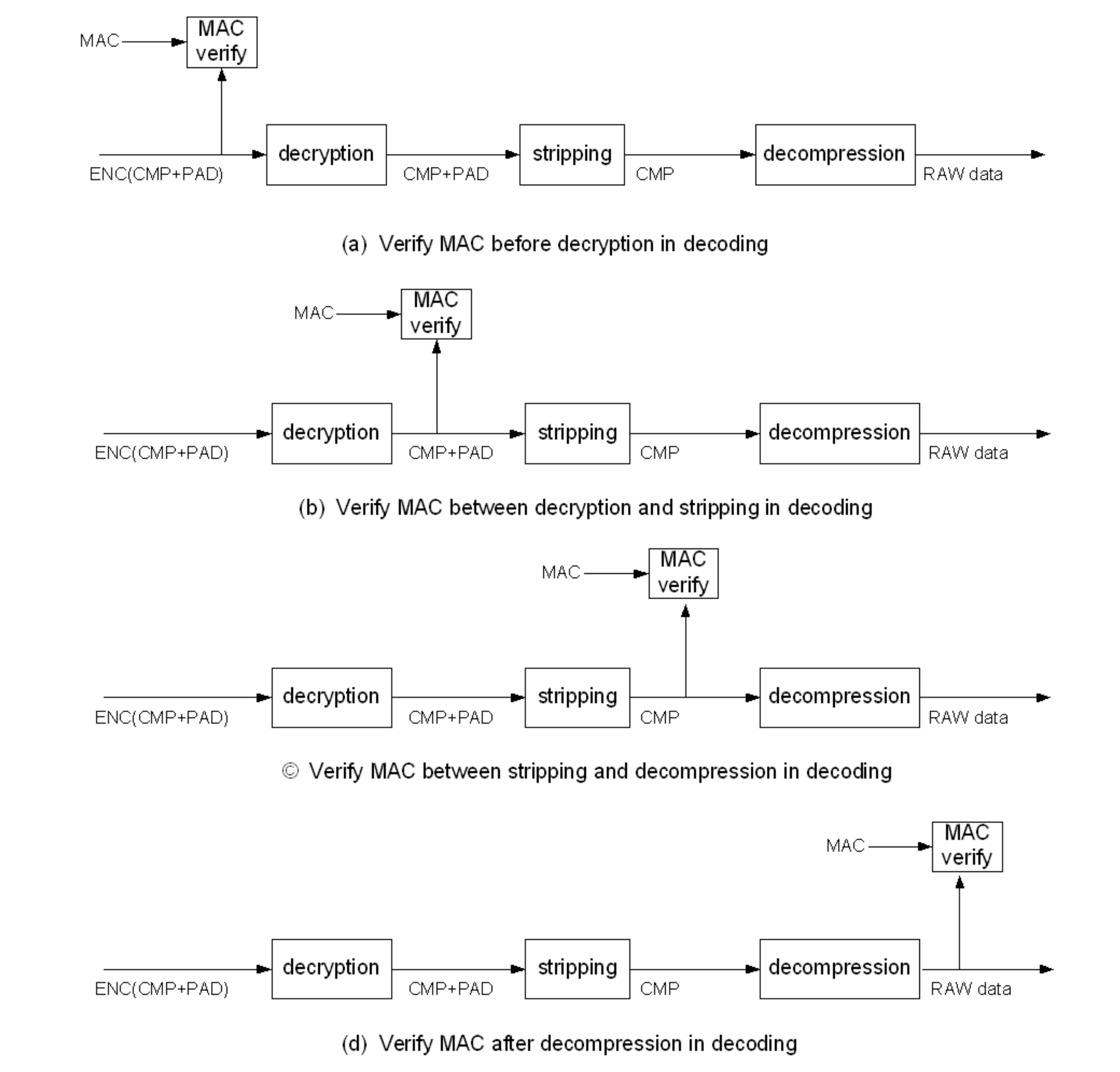


Figure 6-1. Raw Encode Session Data Operation Flow

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.



(a) Verify MAC before decryption in decoding

(b) Verify MAC between decryption and stripping in decoding

© Verify MAC between stripping and decompression in decoding

(d) Verify MAC after decompression in decoding

Figure 6-2. Raw Decode Session Data Operation Flow

7 Application Programs

7.1 demo Application

The demo application performs a simple performance benchmark of the DX SDK Raw Acceleration 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.

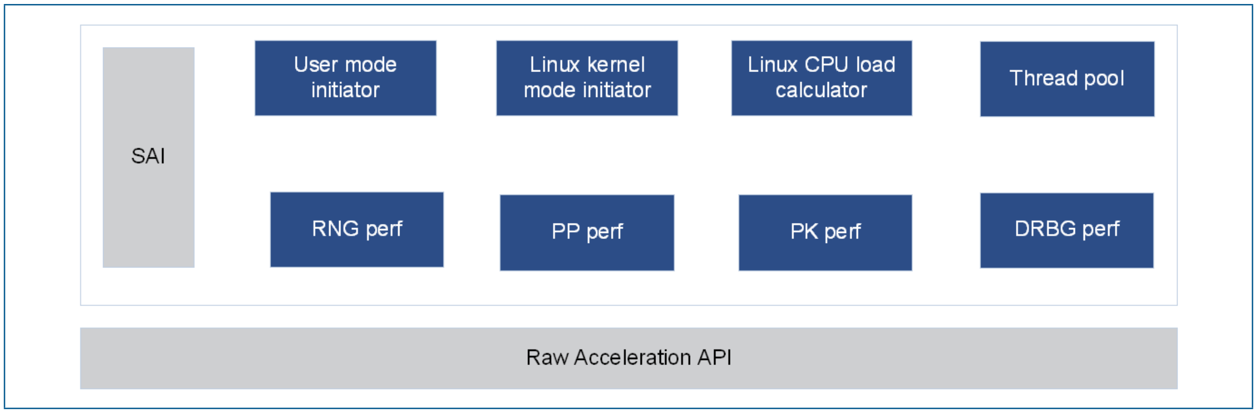


Figure 7-1. demo Application Flow Diagram

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: 1K bit, 2K bit, 4K bit
- RSA: 1K bit, 2K bit, 4K bit
- DSA: 1K bit, 2K bit, 3K 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.

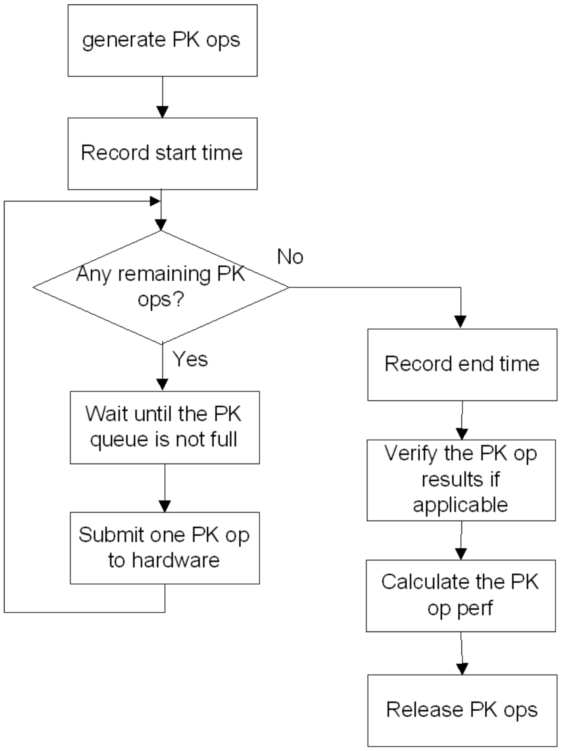


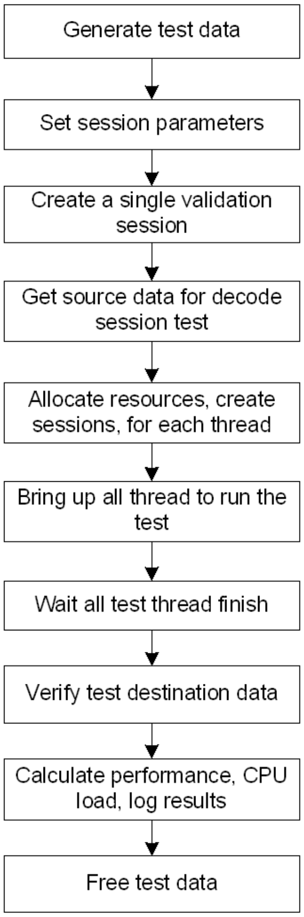
Figure 7-2. PK Performance Flow Diagram

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 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.



Generate test data
Set session parameters
Create a single validation session
Get source data for decode session test
Allocate resources, create sessions, for each thread
Bring up all thread to run the test
Wait all test thread finish
Verify test destination data
Calculate performance, CPU load, log results
Free test data

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.

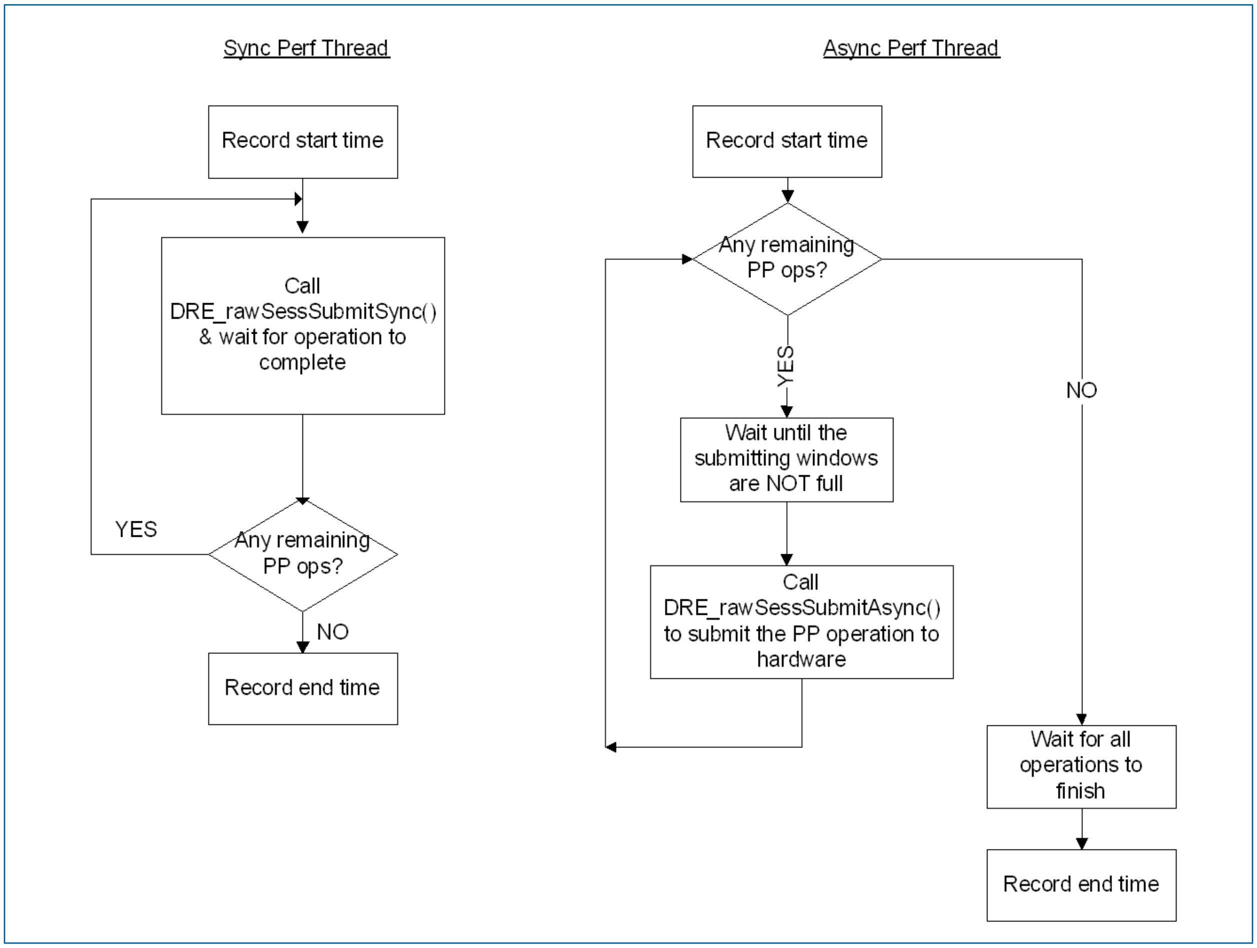

Sync Perf Thread
Record start time
Call DRE_rawSessSubmitSync() & wait for operation to complete
YES
Any remaining PP ops?
NO
Record end time
Async Perf Thread
Record start time
Any remaining PP ops?
YES
NO
Wait until the submitting windows are NOT full
Call DRE_rawSessSubmitAsync() to submit the PP operation to hardware
Wait for all operations to finish
Record end time

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

7.1.8 demo Configuration File

A configuration file, *demo.cfg.xml*, 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.

DX SDK versions 2.1.0L and later support the option of running two demo application programs simultaneously. This feature is useful for testing parallel packet processing and public key operations. To do so, set the configuration parameters for the first demo program to enable *test_raw_pp* and disable *test_raw_pk*, and then set the second demo program configuration parameters to disable *test_raw_pp* and enable *test_raw_pk*. The run time, controlled by the parameter *stop_sec*, should be set to the same value for both demo programs, e.g. 60 seconds. To run both PP and PK demo tests for the same comparable time, the parameter *stop_sec* must take into consideration all parameters specified. Take for example the default configuration file, *demo.cfg.xml*. Both DEFLATE compression and decompression will run for 10 seconds each, for a total runtime of 20 seconds. If the PK RSA test is also enabled, it will run RSA encryption/decryption/sign/verify for 40 seconds. In order for the two tests to complete at the same time, *stop_sec* in the PP demo test should be set to 40 seconds.

test_raw_pp

The parameter *test_raw_pp* sets the Raw Acceleration test mode. The two options are: test Raw Acceleration packet processing session performance and do not test Raw Acceleration packet processing session performance. The default value is enabled.

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. The default value is disabled.

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. The default value is disabled.

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. The default value is disabled.

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. This parameter is only valid if *stop_sec* is = 0. The minimum value for *cmd* is 1, and the maximum value is 2^{23}.

The default value for *cmd* is 20,000.

stop_sec

The parameter *stop_sec* sets the demo application run time in seconds. The minimum value for *stop_sec* is 0. There is no maximum value.

If the value for *stop_sec* is greater than zero, the demo program will run for the specified time. If the value for *stop_sec* is equal to zero, the demo program will run the number of commands specified by the configuration parameter *cmd*.

The default value for *stop_sec* is 10 seconds.

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 *stop_sec* has expired *or 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 200.

thread

The parameter *thread* sets the number of threads to spawn for all tests. The value selected for *thread* should be a multiple of the number of CPUs available in the system.

If *thread* is set to zero, the demo application will allocate the number of threads based on the available number of CPU cores. In synchronous mode, *thread* = 3 * CPU cores; in asynchronous mode, *thread* = 1.5 * CPU cores.

The minimum value for *thread* is 1, and the maximum value is 8192. The default value for *thread* is 0.

sess_per_thread

The parameter *sess_per_thread* sets the number of sessions to run for all threads. This parameter is used primarily for testing a large number of sessions, for example, thousands of sessions.

The minimum value for *sess_per_thread* is 1; the maximum value must satisfy the condition *thread* * *sess_per_thread* < *max_session_num*. The default value for *sess_per_thread* is 1.

7.1.8.3 Raw Acceleration Configuration Settings

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

async

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

The default value for *async* is asynchronous mode.

pkt_size

The parameter *pkt_size* specifies the packet size for each Raw Acceleration command.

The maximum value for *pkt_size* is 512 Kbytes.

The default value for *pkt_size* is 32768.

src_data_file

The parameter *src_data_file* specifies the source data file name using either a relative or absolute path.

Several comments must be made about the size of the source data file. For the sake of discussion, let “file_size” represent the size in bytes of the configured source data file.

1. If file_size ≥ (*pkt_size* * *max_per_run*), then the demo will read (*pkt_size* * *max_per_run*) bytes from the source data file to construct *max_per_run* number of commands, each *pkt_size* bytes in length.

   For example, if file_size = 800 KB, *pkt_size* = 4 KB, and *max_per_run* = 100, the demo application will read the first 400K bytes from the specified source data file to form 100 commands, each 4 KB in length. The remaining 400K bytes are not used.

2. If file_size < (*pkt_size* * *max_per_run*), the demo application will read the entire source data file to construct (file_size / *pkt_size*) number of commands, each *pkt_size* bytes in length. If there are still some unread bytes in the source data file but not enough to form a command that is *pkt_size* in length, the demo application will combine the remaining bytes in the source data file with enough of the starting bytes of the source data file to form a command of *pkt_size* length.

   For example, if file_size is 800 KB, *pkt_size* = 64 KB, and *max_per_run* = 100, the demo application will read 768K bytes from the source data file to form 12 commands that are 64 KB in length, and then append the first 32 KB of the source data file to the last remaining 32K bytes to form another 64 KB command, for a total of 13 64KB commands. These 13 64KB commands are looped to form the 100 (*max_per_run*) 64KB commands.

The source data file may also be left blank. If the source file is left blank, the demo application will create a 256 byte file that contains the values 0x00 to 0xFF (0x00, 0x01, ..., 0xFF, 0x00, 0x01, ...).

The default value for *src_data_file* is demo, the executable file generated when the SDK was built.

en_alg_conf

The parameter *en_alg_conf* specifies whether the demo application will continuously run all selected transforms or will run the specified transform once.

The default value for *en_alg_conf* is to run the configured transform once.

direction

The parameter *direction* sets whether the Raw Acceleration 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 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
- ZLIB

The default value for *comp_algo* is DEFLATE.

stateful_comp

The parameter *stateful_comp* is used to select stateful compression. This parameter is only valid if *comp_algo* is = DEFLATE, GZIP or ZLIB*.*

The valid values for *stateful_comp* are:

- 1 = stateful
- 0 = stateless

The default value for *stateful_comp* is stateless.

enc_algo

The parameter *enc_algo* sets the Raw Acceleration 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

The default value for *enc_algo* is NONE.

seg_hash_algo

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

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

- NONE
- HASH_SHA1
- HASH_SHA256
- HASH_SHA384
- HASH_SHA512
- HASH_MD5
- HMAC_SHA1
- HMAC_SHA256
- HMAC_SHA384
- HMAC_SHA512
- HMAC_MD5
- SSLMAC_MD5
- SSLMAC_SHA1
- SSLMAC_SHA256
- AES_GCM_MAC (enc_algo must be set to AES_GCM)
- AES_GMAC
- AES_XCBC

The default value for *seg_hash_algo* is NONE.

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 RSA.

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

The default value for *pk_mbits* is 256 for ECDH/ECDSA, and 1024 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 2048 bytes.

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
- ALL

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 configured value will be aligned to the closest byte boundary as necessary, e.g. requesting 1020 bits will return 1024 bits.

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 file, reads the data from the specified source file, performs the specified operation, and then outputs the result to the specified destination file. For encryption and hash, the source data is processed in a single command. However, for stateful compression commands with non-zero block sizes, the source data may be read block-by-block and then each data block may be submitted as a unique SDK command. For stateful compression commands, the source file or block size must be less than or equal to 400MB.

7.2.1 sdemo Configuration Files

The configuration file parameters should be set to appropriate values before running the sdemo application.

Two configuration files, *sdemo.encode.cfg.xml* and *sdemo.decode.cfg.xml*, are used to set the test parameters of the sdemo application. The output file of the sdemo encode test is used as the input source file for the sdemo decode test. The output file of the sdemo decode test should be identical to the input source file for the sdemo encode test.

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 ”README.public” in the encode configuration file, and “README.public.encode” in the decode configuration file.

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 README.public.encode in the encode configuration file and README.public.check in the decode configuration file.

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 in the file s*demo.encode.cfg.xml,* and decode in the file s*demo.decode.cfg.xml* .

op_type

The parameter *op_type* 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 *op_type* are:

- COMP
- ENC
- SEG_HASH
- PASS_THRU

For backwards compatibility to DX SDK version 1.x.x, the parameter name *algo* and the setting PASSTH are also recognized.

The default value for *op_type* is COMP.

comp_algo

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

The valid values for *comp_algo* are:

- LZS
- ELZS
- GZIP
- DEFLATE
- ZLIB

The default value for *comp_algo* is DEFLATE.

stateful_comp

The parameter *stateful_comp* is used to select stateful compression. This parameter is only valid if *op_type* is = COMP and *comp_algo* is = DEFLATE, GZIP or ZLIB*.*

The valid values for *stateful_comp* are:

- 1 = stateful
- 0 = stateless

For backwards compatibility, the parameter name *stateful* is also recognized.

The default value for *stateful_comp* is stateless.

block_size

The parameter *block_size* specifies the block size or stateful command size in bytes for stateful compression operations. This parameter is only valid if *stateful_comp* is enabled.

Each block of data in the source file will be submitted as a unique command. Setting *block_size* to zero will force the data in the source file to be submitted as a single command. The maximum value for *block_size* is 400MB.

The default value for *block_size* is zero.

comp_mode

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

The valid values for *comp_mode* are:

- STORE
- STATIC
- DYNAMIC
- OPTIMAL

The default value for *comp_mode* is OPTIMAL.

enc_algo

The parameter *enc_algo* specifies the encryption algorithm for the sdemo test. This parameter is only valid if *op_type* 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 (XR9240 only)

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 *op_type* is = SEG_HASH.

The valid values of *seg_hash_algo* for both encode and decode are:

- HMAC_SHA1, HMAC_SHA256, HMAC_SHA384 ,HMAC_SHA512 (XR9240 only), HMAC_MD5
- SSLMAC_SHA1, SSLMAC_SHA256, SSLMAC_MD5
- AES_GMAC_128, AES_GMAC_192, AES_GMAC_256
- AES_XCBC

The valid values of *seg_hash_algo* for encode only are:

- HASH_SHA1, HASH_SHA256, HASH_SHA384, HASH_SHA512 (XR9240 only), HASH_MD5

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
- Hardware 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 Exar device, sent to the software library, or load balanced.

The valid values for *cmd_target* are:

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

- 33: Device selected using load balancing algorithm

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_192
- ARC4_2048
- HMAC_MD5, HMAC_SHA1, HMAC_SHA256, HMAC_SHA384, HMAC_SHA512

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_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.xml* file, while the AAD input for an AES_GCM_128, 192, or 256 bit key is in the AAD segment of the *ivaad.cfg.xml* file.

7.2.4 sdemo DRBG File

An example DRBG file, *sdemo.drbg.xml*, defines the DRBG data for testing the DRBG function using the sdemo application. In normal operating mode, the DRBG generates an unpredictable pattern of bytes. This file is used to seed the DRBG with a known input so that the output may be verified against the known output.

The SDK must have been built with the option DRE_DEBUG_DRBG enabled in the driver configuration file to test the DRBG. The sdemo DRBG test supports the following algorithms.

- AES_128, AES_192, AES_256
- P_256, P_384, P_521
- SHA-1, SHA-224, SHA-256, SHA-384, SHA-512

7.3 example Application

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

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 standard synchronous and asynchronous modes, as well as synchronous FPGA 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.

7.3.1.3 Synchronous FPGA 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_rawSessOpenEx() to open an extended encode Raw session.

4. Call DRE_rawSessSubmitSyncEx() to request the FPGA to process a packet of data using the encode Raw session.

5. Call DRE_rawSessOpenEx() to open an extended decode Raw session.

6. Call DRE_rawSessSubmitSyncEx() to request the FPGA 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_rawSessCloseEx() to close the extended encode Raw session.

9. Call DRE_rawSessCloseEx() to close the extended decode Raw session.

10. Call DRE_symKeyDestroy() to destroy the symmetric keys if necessary.

7.3.1.4 Asynchronous FPGA 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_rawSessOpenEx() to open an extended encode Raw session.
4. Call DRE_rawSessSubmitAsyncEx() 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_rawSessOpenEx() to open an extended decode Raw session.
7. Call DRE_rawSessSubmitAsyncEx() to request the FPGA 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_rawSessCloseEx() to close the extended encode Raw session.
11. Call DRE_rawSessCloseEx() to close the extended decode Raw session.
12. Call DRE_symKeyDestroy() to destroy the symmetric keys if necessary.

7.3.2 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. The command "make install" must be issued during the SDK build process in order to access the debugging tools man pages.

7.4.1 Monitor Tool

The dx_monitor tool can be used to report the temperature of each installed Exar 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 cards, the PCIe version that is supported by the local host, the PLX switch speed and width capabilities (if present), and the PCIe speed and width for which the DX cards are operating. Note that running the status tool requires lspci version 2.2.7 or higher.

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

```
man dx_status
```

7.4.3 Diagnostic Tool

The tool has many options that are useful for debugging a DX card, such as the ability to run POST on a specific Exar device or all Exar 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 Exar device, inject a false error, and more.

Caution

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.

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 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 Raw Acceleration API.

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_IS_CMD_PROC_BY_SW(status)
This macro returns true if the command was processed by the software library and returns false otherwise.

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.

After each command completes, the user application should call the macro DRE_IS_RESULT_ERR() to identify whether the operation was successful. If the operation was successful, the user application should also call DRE_IS_CMD_PROC_BY_SW() to determine whether the command was completed by the software library. For more information, refer to Table 8-3.

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.
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.

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

Bits	Name	Description
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 21 = DRE_SESSION_HEADER 22 = DRE_MAILBOX 23 = DRE_SDDINIT 24 = DRE_GLOBALCONFIG 25 = DRE_CHARDEV 26 = DRE_DRBG_MANAGER 27 = DRE_STATISTICS 28 = DRE_ERRHANLDING
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 Exar devices and DX cards. The shaded error codes are obsoleted.

Table 8-2. Error Code Field Description

Value	Name	Description
0	DRE_C_NO_ERROR	No error
1	DRE_C_GENERAL_ERROR	Unclassified error, has to check log to get the detail
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_BUFLENGTH_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 *driver.cfg*)
64	DRE_C_DMA_MAP_FAILED	DMA mapping failed
65	DRE_C_CHIP_FAILED	Exar device initialization failed
66	DRE_C_READ_FAILED	Read file failed
67	DRE_C_XML_READ_FAILED	Parse XML file failed
68	DRE_C_STATEFUL_SESS_FAILED	Command in a stateful session failed (will result in all commands in that session to fail)
255	DRE_C_APP_ERR	For use by customer applications; not used by the DX SDK

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.

The DX SDK has 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 6)

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.

Table 8-3. Combined Error Status and Flag Conditions (Sheet 2 of 6)

Failover	CMD Target	Status Field DRE_IS_RESULT_ERR ()	Software Library Flag Field DRE_IS_CMD_PROC_BY_SW()	Description	Action
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 3 of 6)

Failover	CMD Target	Status Field DRE_IS_ RESULT_ERR ()	Software Library Flag Field DRE_IS_CMD _PROC_BY_ 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.	
X	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 4 of 6)

Failover	CMD Target	Status Field DRE_IS_RESULT_ERR()	Software Library Flag Field DRE_IS_CMD_PROC_BY_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 is OK, the hardware failure was not catastrophic and 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 5 of 6)

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 6 of 6)

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

Failover controls the SDK's behavior in the unlikely event off all Exar hardware in the system failing. If failover is enabled in the driver configuration file, the SDK will process all submitted commands using the software library. If failover is not enabled in the driver configuration file, the submitted commands will not be processed and will return with an error. The decision whether to failover to software will not occur during a recovery process.

8.3 Single Command Error Handling

If a 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. 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.

For DX SDK versions 2.0.0L and later, the command processing behavior was modified when an overflow error occurs. If the only error returned is an overflow error, the SDK will return an error code from the API call but will not log the error message and will not send the command to the software library. If multiple hardware errors are returned, one of which is overflow, the SDK will print the error message and resubmit the command to the software library.

For DX SDK versions 2.1.0L and later, if a single command in a synchronous mode stateful session fails, the application may re-submit only the failed command if the failure is considered recoverable, such as an overflow error or ring-full error.

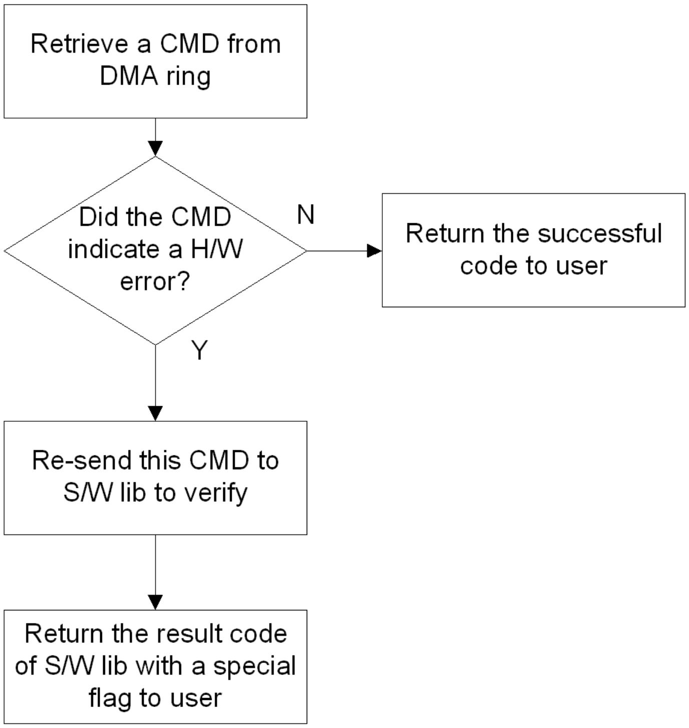

Retrieve a CMD from DMA ring
Did the CMD indicate a H/W error?
N
Return the successful code to user
Y
Re-send this CMD to S/W lib to verify
Return the result code of S/W lib with a special flag to user

Figure 8-1. Single 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.

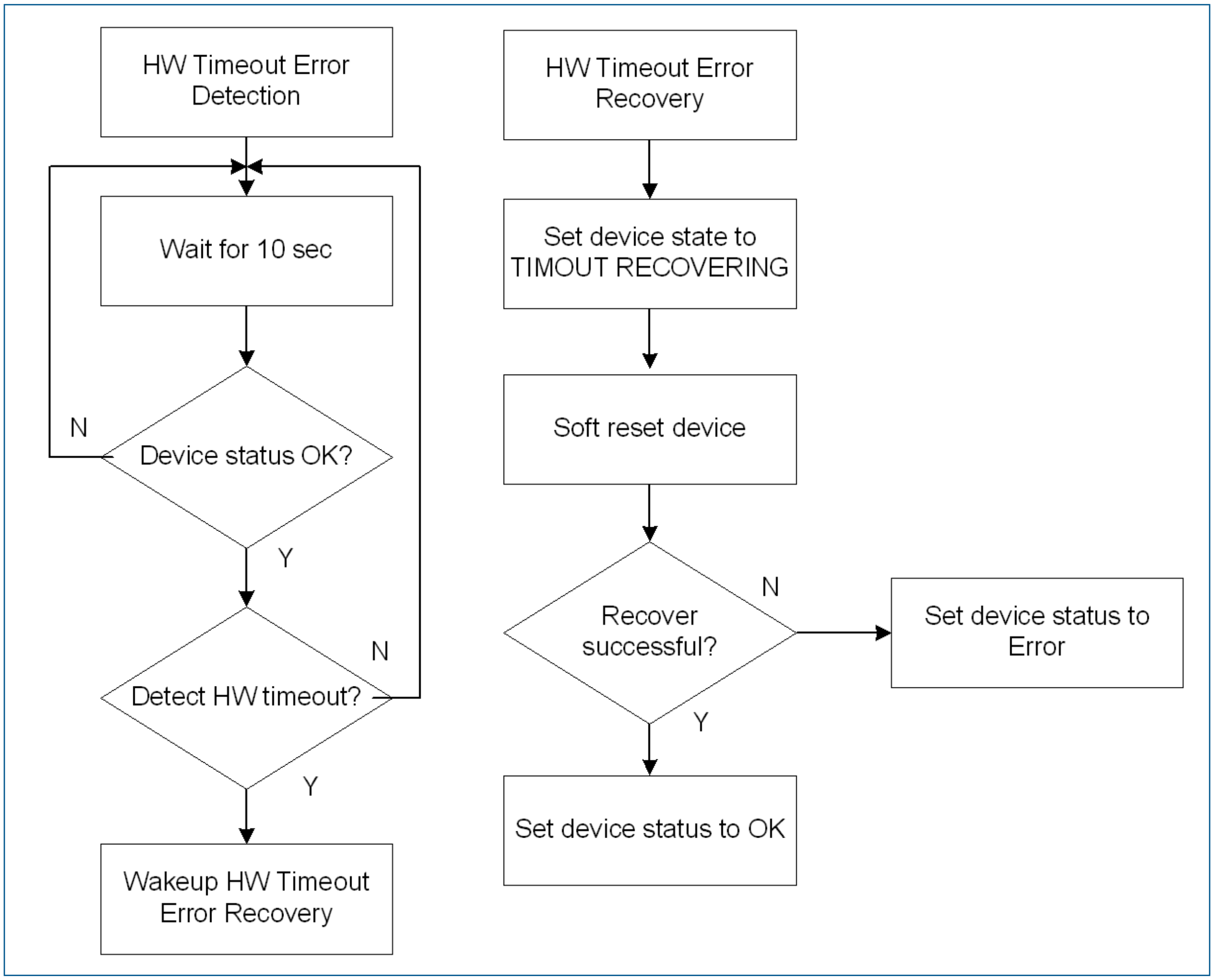

HW Timeout Error Detection
Wait for 10 sec
N
Device status OK?
Y
N
Detect HW timeout?
Y
Wakeup HW Timeout Error Recovery
HW Timeout Error Recovery
Set device state to TIMOUT RECOVERING
Soft reset device
Recover successful?
N
Set device status to Error
Y
Set device status to OK

Figure 8-2. Hardware Timeout Error Processing

8.5 Overheated Condition Error Handling

Figure 8-3 illustrates the error handling by the DX SDK for detecting and recovering an Exar 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.

After the driver is loaded, the DX SDK programs the normal and overheated temperature values set in the driver configuration file, *driver.cfg.xml*, into the XR9240 temperature registers. If the XR9240 detects that its temperature has exceeded the configured overheated temperature, it will trigger an overheat interrupt, causing the SDK to mark the device as overheated, flash the Error LED, and stop submitting commands to that device. Once the XR9240 temperature has returned to a value lower than the configured normal temperature, the device will trigger an overheated recovery interrupt which will cause the SDK to mark the device as normal, turn off the Error LED, and resume submitting commands to that device.

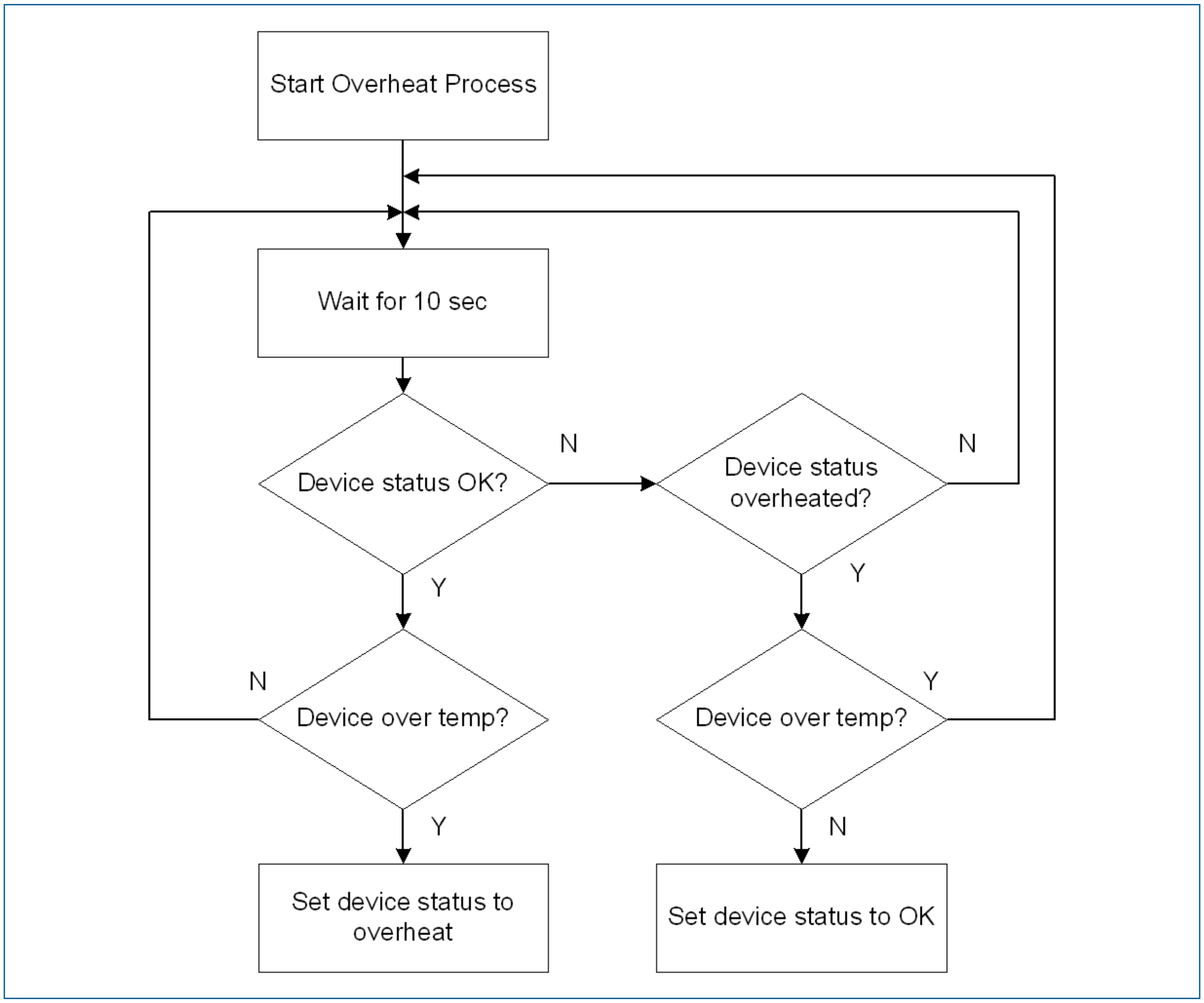


Figure 8-3. Hardware Overheated Error Processing

8.6 PCIe Error Handling

DX SDK versions 2.0.0La and later support the standard PCIe Advanced Error Reporting (AER) mechanisms and functions that are defined by the OS/kernel AER framework.

The DX SDK responds to all data integrity, PCIe error, and ILK interrupts generated by a XR9240 device by attempting to recover the device via a soft reset.

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 Exar device 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 Exar device 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 a register is accessed and trigger the PCIe error handling flow if 0xFFFFFFFF is returned.

8.6.2 Link Speed and Width Degradation

This section describes the DX SDK handling of PCIe link speed and width degradation in more detail. This feature is always enabled, regardless of the setting of the driver configuration parameter "pcie_error_recovery_enable". A PCIe link speed and width degradation error will not increment the pcie_error_interval_threshold counter.

The PCIe error detection module verifies that the DX2040 card is working at Gen3 speed and x8 width when the driver is loaded and again during a configurable interval. The default value for the interval is 300 seconds, and may set to a different value in the configuration file *dre_config.h* using the parameter DRE_NM_LINK_CHECK_INTERVAL_CNT. Setting DRE_NM_LINK_CHECK_INTERVAL_CNT to zero will disables the link speed and width verification. For testing purposes, a value of 20 seconds is recommended.

If the negotiated PCIe link width is not x8, the SDK will trigger a PCIe hot reset process to recover the DX2040 card to x8 width. If the card recovers to x8 width, the SDK will log a message that the recovery was successful. If the card does not recover to a x8 width, the SDK will log a message that the recovery was not successful.

If the negotiated PCIe link speed is Gen1 (2.5GT/s), the SDK will trigger the PCIe link retraining process. If the card recovers to Gen3 speed, the SDK will log a message that the recovery was successful. If the card does not recover to Gen3 speed, the SDK will log a message that the recovery was not successful.

If both the negotiated PCIe link width and speed are degraded, the SDK will trigger the PCIe link retraining process and issue a PCIe hot reset.

It is assumed that the DX2040 card is connected to a slot that supports PCIe Gen3 x8 or better. The SDK does not verify the slot capability. If a DX2040 card is installed in a slot that does not support PCIe Gen3 x8, the recovery process will be triggered continuously on every interval.

8.7 Data Corruption Error Handling

The XR9240 device is able to detect data integrity errors that occur in the PCIe interface or XR9240 internal data path, and will notify the SDK via an interrupt if a data integrity error occurs. The SDK always enabled the fencing mode of the XR9240 device. If a fatal data integrity error occurs, the SDK will block the errored device from sending data to the result buffer to prevent data corruption.

If the hardware reports a fatal ECC error, the SDK will try to recovery the device by issuing a hot reset to the errored card. Commands that were previously submitted to that card will be processed in software using swlib. If the recovery is successful, the SDK will send future commands to that device. If the recovery failed, the SDK will take that card offline, and all future commands will be submitted to other DX cards for processing.

Appendix A: Usage and Standards Compliance of the Random Number Generator

A.1 Overview

Exar's security processors and its 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 Exar 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 Exar device as the initial seed.

The DX SDK provides an API function to retrieve RNG values from the Exar device (see "Retrieve a Nondeterministic RNG Value from an Exar 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 acceleration processors contain 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.

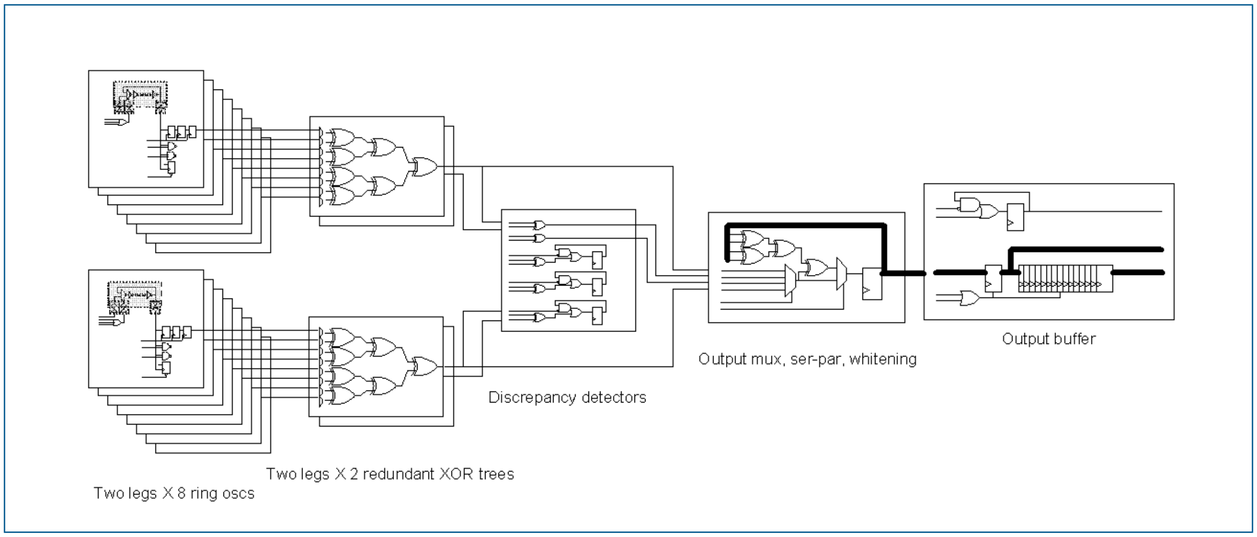

Output buffer
Output mux, ser-par, whitening
Discrepancy detectors
Two legs X 2 redundant XOR trees
Two legs X 8 ring oscs

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 *XR9240 Data Sheet* for a detailed description of the RNG registers. The Raw Acceleration API provides functions to retrieve a random number from the RNG module, precluding the need to access the internal registers.

A.3 Software Implementation

The DX SDK configures and controls the hardware RNG module. The DX SDK uses the default reset values for the Exar device RNG registers, however DX SDK 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 hardware RNG module

The DX SDK supports NIST SP 800-90 (http://csrc.nist.gov/ publications/nistpubs/800-90/ SP800-90revised_March2007.pdf), and implements the CTR_DRBG that supports AES 128, 192, 256 bit keys, and implements the Dual_EC_DRBG that supports 256, 384, 521 bit keys. Refer to the *Raw Acceleration API User Guide*, USR-0040, for details.

In addition, the DX SDK includes a “/proc” entry for “Monitoring the RNG Statistics”.

A.3.1 Retrieve a Nondeterministic RNG Value from an Exar Device

The DX SDK Raw Acceleration API provides a function to retrieve a RNG value from an Exar device. Within the DX SDK API, the nondeterministic RNG value retrieved from an Exar device is referred to as the Raw RNG. Please refer to the *Raw Acceleration Application Programming Interface Reference Guide*, USR-0040, 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 hardware RNG module.

Figure A-2 depicts the flow for how the DX SDK retrieves a RNG value from the Exar device. 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 DX SDK implements one RNG object for each Exar device in the system. Each RNG object may be uniquely seeded by its hardware RNG.

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

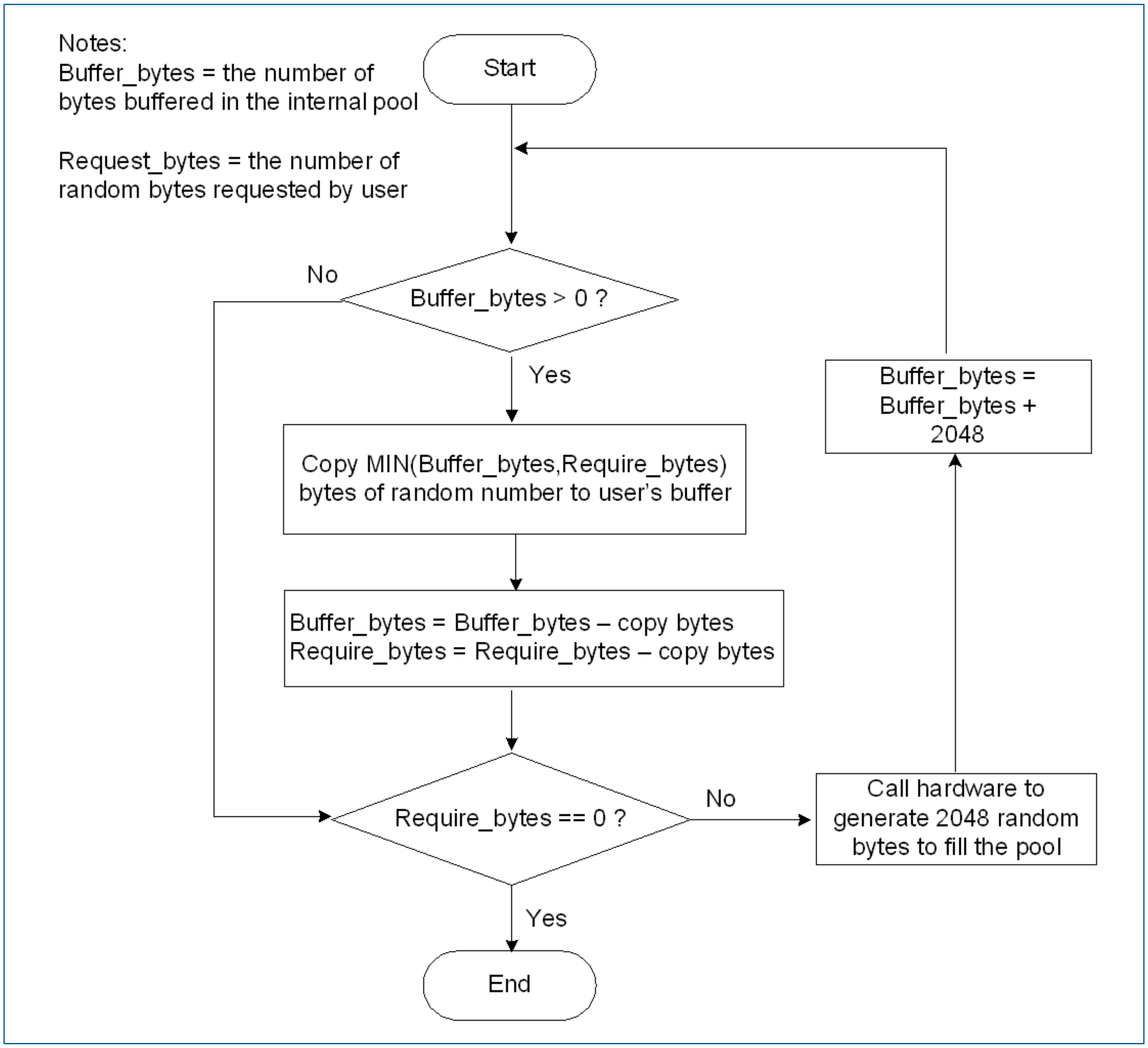

Notes:
Buffer_bytes = the number of bytes buffered in the internal pool
Request_bytes = the number of random bytes requested by user
Start
Buffer_bytes > 0 ?
No
Yes
Copy MIN(Buffer_bytes,Require_bytes) bytes of random number to user's buffer
Buffer_bytes = Buffer_bytes – copy bytes
Require_bytes = Require_bytes – copy bytes
Require_bytes == 0 ?
No
Yes
Call hardware to generate 2048 random bytes to fill the pool
Buffer_bytes = Buffer_bytes + 2048
End

Figure A-2. Flow to Retrieve RNG Number from an Exar Device

A.3.2 RNG Test

The DX SDK meets the FIPS 140-2 Section 4.9.2 requirement for a continuous random number generator test. The 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 hardware RNG and save them as R0, R1,…,R15.

2. Verify that R0 != R1, R1!= R2, …, R14 != 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 Exar device will continue to process commands. The 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

The DX SDK includes a Linux "/proc" entry that allows users to easily monitor the RNG status of all Exar devices in the system. As described in Section A.3.1, a per-device pool buffers the random bytes retrieved from the hardware RNG engine.

To view the RNG status, enter:

```
cat /proc/exar/dx_rng_statistics
```

The example output below was generated on a system with a single DX 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 hardware 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 Exar 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 Exar device 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

Exar plans to validate the XR9240 device 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*.

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 DX SDK's Raw Acceleration API to access the RNG and SHA-256 functions of the Exar device to realize the Hash_DRBG method described in NIST SP 800-90.

The DX SDK version 2.x.x contains a DRBG implementation and API function that when called returns 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

Table B-1. eLZS Compression Flag Definitions

Flag Value	Description
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

Table B-2. Calgary Corpus Performance Test Results

Calgary Filename	Balanced	Max speed	Improved Comp	Optimal Comp
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
PLRABN12.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 ELZS_ERROR_SRC_UNDERRUN error will be returned if there are unused decompressed source bytes after the EOCD token. This flag is ignored if ELZS_FLAG_EOCD_CONTINUE 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 **flag** is set to DRE_PKT_MIDDLE or DRE_PKT_LAST, otherwise the error DRE_ERR_SW_INVALID_ARG will be returned.
ihvLen	Input	Number of bytes in the partial hash buffer. This parameter is required for stateful operations if **flag** is set to DRE_PKT_MIDDLE or DRE_PKT_LAST, otherwise the error DRE_ERR_SW_INVALID_ARG 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 A01

Initial release.

Document Revision A02

Update 1. Updated for DX SDK version 2.0.0Lb thoughout.

Update 2. Section 7.1.8.2 Global Configuration Settings: changed default setting of cmd.

Update 3. Section 7.1.8.3 Raw Acceleration Configuration Settings: changed default setting of src_data_file.

Update 4. Section 7.3.1.3 Synchronous FPGA Mode: added this new section.

Update 5. Section 7.3.1.4 Asynchronous FPGA Mode: added this new section.

Document Revision A03

Update 1. Updated for DX SDK version 2.0.0L thoughout.

Update 2. Section 1 Introduction: updated list of supported OS.

Update 3. Section 2.1.2 Encryption/Decryption Algorithms: added reference to IV replacement feature.

Update 4. Section 2.1.3 Authentication Algorithms: removed reference to slice-hash.

Update 5. Section 4.2.2 Driver Memory Allocation: updated values for required memory.

Update 6. Section 5.1 Initialization Sequence: added step to ignore the first 16 words of the RNG output.

Update 7. Section 5.2.1 Host Initialization Settings: removed caution about degraded performance for pcie_error_recovery_enable.

Update 8. Section 5.2.4 Temperature Sensor Settings: removed statement about erratum in preliminary hardware.

Update 9. Section 5.3.3.4 Flash Access Module: removed note that this feature is not supported for the beta release.

Update 10. Section 8.3 Single Command Error Handling: added text describing overflow error during command processing.

Update 11. Section 8.5 Overheated Condition Error Handling: removed note that this feature is not supported for the beta release.

Update 12. Section 8.6.2 Link Speed and Width Degradation: added this new section.

Update 13. Section 8.7 Data Corruption Error Handling: replaced statement about PCIe data corruption.

Update 14. Section A.3.3 Monitoring the RNG Statistics: renamed the proc entry.

Document Revision A04

Update 1. Updated for DX SDK version 2.1.0L thoughout.

Update 2. Section 1 Introduction: added new supported operating systems. SLES11 SP2 kernel version 3.0.10 and Fedora 19 kernel version 3.9.4

Update 3. Section 3.2.2 Stateful Sessions: added ability to resubmit single stateful commands if the failure was recoverable.

Update 4. Section 4.2.2 Driver Memory Allocation: added formula for calculating the required memory.

Update 5. Section 5.1 Initialization Sequence: added last step of running POST.

Update 6. Section 5.2.1 Driver Host Initialization Settings: changed default setting for notification_mode to interrupt mode using a tasklet. Updated description for how the SDK manages the pp and pk max_key_num tables. For the parameter max_session_num, removed the text about how the table entries are managed and added description of typical session sizes. Added new parameter cpu_dma_zero_latency.

Update 7. Section 5.2.2 Command Structure Settings: changed default setting for cmds_per_ring to 4096 and removed text about large vs small sized packets.

Update 8. Section 5.3.3.4 Flash Access Module: removed note saying that this feature is not currently supported.

Update 9. Section 7.1.8.1 Demo Test Configuration Settings: added description of running several demo tests simultaneously.

Update 10. Section 7.1.8.2 Global Configuration Settings: added parameters stop_sec and sess_per_thread. Added setting of zero to thread and changed default value to zero. Changed default value of max_per_run to 200.

Update 11. Section 7.1.8.3 Raw Acceleration Configuration Settings: changed default value for async to asynchronous. Updated the description and default setting for en_alg_conf. Changed the default setting for comp_algo to DEFLATE. Added the parameter stateful_comp.

Update 12. Section 7.1.8.4 Public Key Configuration Settings: changed default setting for pk_algo to RSA.

Update 13. Section 7.2 sdemo Application: added description of how the source data is handled for stateful comp commands.

Update 14. Section 7.2.1.1 File Settings: corrected default values for both src_data_file and dst_data_file.

Update 15. Section 7.2.1.2 Transform Settings: updated default value for direction. Changed name of parameter algo to op_type and it setting PASSTH to PASSTHRU. Added the new parameters stateful_comp and block_size. Changed valid options for seg_hash_algo to reflect encode only and encode/decode only.

Update 16. Section 7.4.2 Status Tool: removed help option.

Update 17. Section 8.3 Single Command Error Handling: added ability to resubmit single stateful commands if the failure was recoverable.

Update 18. Section 8.5 Overheated Condition Error Handling: added description of overheated error detection and handling based on interrupt instead of polling.

Update 19. Section 8.7 Data Corruption Error Handling: updated description as DRE_92XX_HOT_RESET_ENABLE is enabled with 2.1.0L.

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