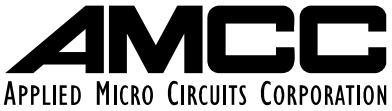
Security Look-aside Driver (SLAD)
for
AMCC Security Co-Processor, v2.2
User's Manual

PowerPC®



Applied Micro Circuits Corporation
215 Moffett Park Drive, Sunnyvale, CA 94089
Phone: (408) 542-8600 — Fax: (408) 542-8601
http://www.amcc.com

Contents

Abstract

This document is derived from the original, Security Look-aside Accelerator Driver (SLAD) User Manual, 1.7, Date: 11 Jan, 2008.

Any errors noted here would be translation errors from the original.

Any additions will be duly marked as such and do not reflect back to the original documentation.

1. Introduction

The Security Look-aside Driver (SLAD) provides a driver that is used by user-mode and kernel-mode applications to communicate with the security engine used in applicable AMCC processor products. This driver implements an Application Programming Interface (API) for communication between applications and the security engine. This manual is primarily intended for a software developers.

For general information about AMCC processors products, please visit the AMCC Web site at:

http://www.amcc.com/Embedded/

1.1 Acronyms

Table 1-1 provides a list of all acronyms used in this manual.

Table 1-1. List of Acronyms

		IP	Internet Protocol
AH	IPSec Authentication Header	IPcomp	IP Compression Protocol
API	Application Programming Interface	IPSec	IP Security Protocol
BM	Byte Memory	IV	Initialization Vector
CBC	Cipher Block Chaining Mode	KCR	Key Cache Register
CC	Crypto Context	KEK	Key Encryption Key
CDR	CGX Descriptor Ring	KRAM	Kernel RAM
CFB	Cipher Feedback Mode	LSV	Local Storage Variable (KEK)
CGX	Cryptographic Extensions Library	MD5	Message Digest 5
CPI	Compression Parameters Index	OFB	Output Feedback Mode
CRT	Chinese Remainder Theorem	PCDB	Program Control Data Bits
DEK	Data Encryption Key	PDR	Packet Descriptor Ring
DES	Data Encryption Standard	PF	Programmable Flags
DH	Diffie-Hellman	PKA	Public Key Acceleration
DKEK	Hash/Encrypt Data Key Protection KEK	PKCP	Public Key Co-Processor
DM	Data Memory	PM	Program Memory
DSA	Digital Signature Algorithm	RAM	Random Access Memory

Table 1-1. List of Acronyms

DSP	Digital Signal Processor	RDR	Result Descriptor Ring
ECB	Electronic Codebook Mode	ROM	Read Only Memory
EMI	External Memory Interface	RNG	Random Number Generator
ESP	IPSec Security Encapsulating Payload	RSA	Rivest, Shamir, Adelman (public key algorithm)
GKEK	Generator KEK	SA	Security Association
HMAC	Hash Message Authentication Code	SHA-1	Secure Hash algorithm, Version 1
IKE	Internet Key Exchange	SPI	Security Parameters Index

1.2 Overview

The Security Look-aside Driver (SLAD) is a very flexible driver that facilitates communication between security applications and the security engine subsystem. It provides services to both kernel-mode and user-mode applications.

AMCC SLAD uses a pair of descriptor rings for communication with the host system. These rings are designated as the Packet Descriptor Ring (PDR) and the CGX Descriptor Ring (CDR).

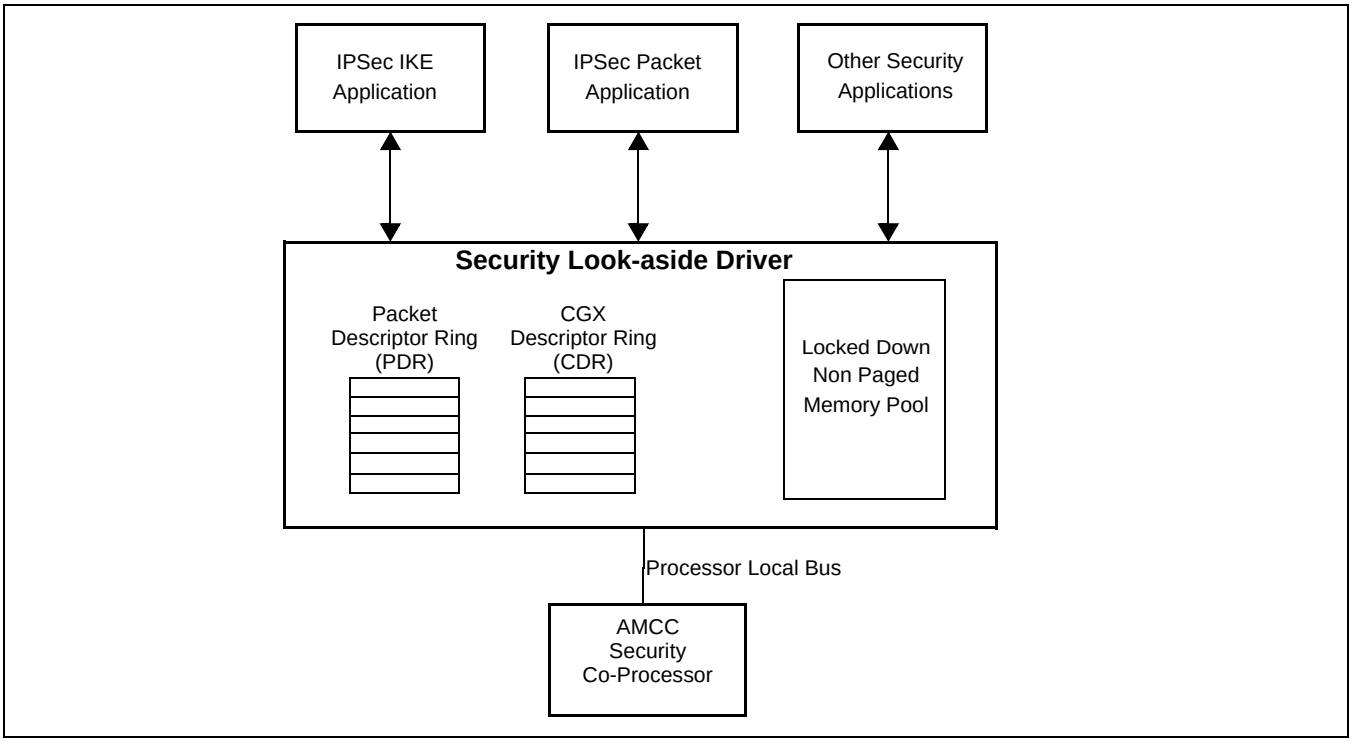
The driver is responsible for setting up and managing these rings on behalf of the host application. For details about the behavior and control of these rings, please refer to the PPC405/PPC460 User Manual. The driver is also responsible for booting, initializing and configuring the hardware device subsystem.

The driver provides the following primary functions:

- Initialization, de-initialization and general management of the hardware subsystem
- Functions to put commands on, and get results back from the CGX Descriptor Ring (CDR)
- Functions to put commands on, and get results back from the Packet Descriptor Ring (PDR)
- Functions to read and write directly to the security co-processor
- Buffer management functions such as allocate, copy in and copy out

Figure 1-1 provides an overview of the driver. To minimize waiting for applications, SLAD maintains two rings-PDR and CDR. Generally applications put commands in PDR while only the CGX library uses the CDR. The SLAD interacts with the security-device to execute the required commands. The SLAD maintains a pool of memory to store the data received from applications and the security-device. Since the security-device may read and write data directly to the memory through DMA, the memory is non-paged and locked.

Figure 1-1. SLAD Communications



1.2.1 Initialization and Configuration Overview

The initialization and configuration functions of the SLAD prepare the security co-processor for operation. This may include the following steps:

1. Bootloading any operating code into the security co-processor device.
2. Passing values from the PE_INIT_BLOCK structure into the hardware device. The PE_INIT_BLOCK is used to define the configuration options for the device as well as the settings for various registers within the device.

1.2.2 Operation after Initialization

Once the hardware device is booted and initialized, the driver manages the descriptor rings and thus the submission of commands to the co-processor. Its operation is asynchronous to the host processor, as descriptor rings are used for inter-process communications.

1.3 General Notes

This section provides miscellaneous information about the SLAD.

1.3.1 Device Numbers

The SLAD supports more than one security co-processor at a time. The current implementation only assumes a single device, numbered zero (0).

1.3.2 Virtual Memory and Physical Memory

When a security co-processor device is accessing data stored in system memory, for example, across a PCI bus, it is always referencing physical memory locations. Linux supports "virtual memory" and implements "paged memory," which allows different banks of memory to be switched in for access. Both virtual memory and paged memory pose a problem, because the physical addressing used by the security co-processor cannot understand virtual or paged addresses.

To avoid this problem, data must be placed in contiguous, non-paged, non-virtual memory (locked). Most kernel-mode applications only operate out of non-paged non-virtual memory, so there is no issue in the kernel space. However, data originating in the user space will generally have to be copied by the driver into non-paged memory. The driver copies the user mode application data into driver allocated/managed bounce buffers.

1.3.3 CGX Command Parsing

In systems with virtual or paged memory, the security co-processor may have to fetch its CGX arguments as a bus master, and it will, of course, use physical addresses. As with packet operations, the arguments will have to be stored in locked, non-paged memory. The driver takes care of this by parsing each CGX command, isolating the pointer type arguments, and then copying the argument data from the virtual location to a physical, locked memory area.

1.3.4 RSA-CRT Modular Exponentiation

The formula for RSA-CRT modular exponentiation is as follows (definitions of the variables used are provided after the formula):

```
/*** Garner Recombination ***/
if(mp >= mq)
   tmp1 = mp – mq
else (where QINV=MODQ^-1 mod MODP)
   tmp1 = mq – mp
   tmp1 = MODP – tmp1
tmp2 = (QINV * tmp1) mod MODP
RESULT = (tmp2 * MODQ) + mq
```

where:

`cp` INPUT mod MODP

`cq` INPUT mod MODQ

`mp` cp^{DP} mod MODP

`mq` cq^{DQ} mod MODQ

1.3.5 Target Mode vs. Autonomous Ring Mode

All the APIs shall use Autonomous ring mode. Target mode is implemented only for debugging purposes and should not be used in any production environment.

2. SLAD API Function Summary

The following table provides a summary of the SLAD API functions (the platform-specific sections also be examined in order to determine if there have been any additions or subtractions to this list):

Table 2-1. List of SLAD API Functions

Function Name	Notes
slad_driver_version	Returns the slad version number
slad_device_info	Returns the SLAD_DEVICEINFO structure for a specified device
slad_setup_pe_initblk	Fills up the default values in the PE_INIT_BLOCK structure
slad_pe_init	Initializes the hardware co-processor using a supplied PE_INIT_BLOCK structure
slad_pe_uninit	Un-initializes the hardware co-processor and de-allocates buffer memory
slad_pka_init	Initializes the security hardware PKA using a supplied PKA_INIT_BLOCK structure
slad_pka_uninit	Un-initializes the security hardware PKA
slad_rng_init	Initializes the security hardware random number generator using a supplied RNG_INIT_BLOCK structure
slad_rng_uninit	Un-initialize the security hardware random number generator
slad_register_sa	Register SA with the driver. After registration, SA can be used in slad_pkt_put/slad_cgx_put functions
slad_register_srec	Register Srec so that the driver may associate it with its corresponding SA
slad_unregister_sa	Unregister SA or state record from the driver
slad_pkt_put	Enqueues a packet processing command onto the Packet Descriptor Ring
slad_pkt_get	Dequeues the next completed Packet from the Packet Descriptor Ring
slad_pkt_sync	Enqueues/dequeues a single packet
slad_pkt_ready	Fetches packet completion status for all active devices in a system
slad_bus_read	Reads data directly from the security co-processor's memory space
slad_bus_write	Writes data directly into the security co-processor's memory space
slad_allocate_buffer	Allocates a physical memory buffer for hardware access
slad_free_buffer	Frees the physical memory allocated by slad_allocate_buffer from the caller-supplied source buffer
slad_buffer_copy_in	Copies data to the buffer allocated by slad_allocate_buffer from the caller-supplied source buffer
slad_buffer_copy_out	Copies data from the buffer allocated by slad_allocate_buffer to the caller-supplied destination buffer
slad_map_addr_range	Map a physical address range to virtual addresses
slad_unmap_addr_range	Unmaps the memory obtained by slad_map_addr_range()
slad_get_random	Generates a true random number of the required size.
slad_expmod	Performs the a^p mod m mathematic calculation
slad_expcrtmod	Performs the RSA-CRT modular exponentiation

The prototypes for all of the driver functions are in the slad.h header file.

2.1 SLAD Function Details

2.1.1 slad_driver_version

int slad_driver_version (UINT32 *vers)

where:

vers pointer to 32-bit version number to be populated by this function

Gets the SLAD version number of the driver. The 32-bit variable pointed to by the vers parameter is filled in by the driver. This command does not require any device to be initialized. The 32-bit version number has the following format:

Bits 24-31 (MSB): major version number (NN.xx.xx)

Bits 16-23: minor version number (xx.NN.xx)

Bits 8-15: very minor version number (xx.xx.NN)

Bits 0-7 (LSB): pre-release number (set to 0 for official release)

2.1.2 slad_device_info

int slad_device_info (int device_num, SLAD_DEVICEINFO *info)

where:

device_num hardware device number

info pointer to SLAD_DEVICEINFO structure to be populated by this function

Gets information about the specified device. The SLAD_DEVICEINFO structure (see *SLAD_DEVICEINFO* on page 26) is filled in by the driver. This command does not require the device to be initialized.

2.1.3 slad_setup_pe_initblk

int slad_setup_pe_initblk (int device_num, PE_INIT_BLOCK *iblk, int *psg_flag)

where:

device_num hardware device number

iblk pointer to the PE_INIT_BLOCK structure

psg_flag pointer to an integer flag, indicating if scatter/gather must be initialized (1) or not (0)

Fills up the default values in the PE_INIT_BLOCK structure for the device specified in slad_device_info(). A user can change the values of PE_ INIT_BLOCK as needed, and initialize the device by calling slad_pe_init().

2.1.4 slad_pe_init

int slad_pe_init (slad_app_id_type * app_id, int device_num, PE_INIT_BLOCK *iblk)

where:

app_id — An opaque output parameter to the caller, used in other API functions; must not be modified by the caller at anytime.

device_num — hardware device number

iblk — pointer to populated PE_INIT_BLOCK structure

Initializes a device using parameters in the caller-supplied PE_INIT_BLOCK structure. If the device has already been initialized, the device will first be un-initialized, and then re-initialized.

2.1.5 slad_pka_init

int slad_pka_init (slad_app_id_type *app_id, int device_num, PKA_INIT_BLOCK *iblk)

where:

app_id — An opaque output parameter to the caller, used in other API functions; must not be modified by the caller at anytime.

device_num — hardware device number

iblk — pointer to populated PKA_INIT_BLOCK structure

Initialize the security device PKA engine using the parameters in the caller supplied PKA_INIT_BLOCK structure

2.1.6 slad_rng_init

int slad_rng_init (slad_app_id_type *app_id, int device_num, RNG_INIT_BLOCK *iblk)

where:

app_id — An opaque output parameter to the caller, used in other API functions; must not be modified by the caller at anytime.

device_num — hardware device number

iblk — pointer to populated RNG_INIT_BLOCK structure

Initialize the security device random number generator engine using the parameters in the caller supplied RNG_INIT_BLOCK structure

2.1.7 slad_pe_uninit

int slad_pe_uninit (slad_app_id_type app_id, int device_num)

where:

app_id — An opaque output parameter to the caller, used in other API functions; must not be modified by the caller at anytime

device_num — hardware device number

Un-initializes a device. This command should especially be used before de-allocating any application memory that was supplied to the driver for DMA access (for example, user-supplied ring space in host memory). All devices will be automatically un-initialized if the driver is unloaded.

2.1.8 slad_pka_uninit

int slad_pka_uninit (slad_app_id_type app_id, int device_num)

where:

app_id — An opaque output parameter to the caller, used in other API functions; must not be modified by the caller at anytime.

device_num — hardware device number

Un-initialize the security device PKA engine. The device will be automatically un-initialized if the driver is unloaded.

2.1.9 slad_rng_uninit

int slad_rng_uninit (slad_app_id_type app_id, int device_num)

where:

app_id — An opaque output parameter to the caller, used in other API functions; must not be modified by the caller at anytime.

device_num — hardware device number

Un-initialize the security device, random number generator engine. The device will be automatically un-initialized if the driver is unloaded.

2.1.10 slad_register_sa

int slad_register_sa(sa_handle *handle, void *sa_buff, slad_bus_addr bus_addr, int len, unsigned flags)

where:

handle — An opaque parameter to the caller which is used in slad_pkt_put/get functions. It should not be modified by the client.

sa_buff — Address of the buffer containing the Security Association (SA)

bus_addr — Bus Address of the SA buffer

len — Size of SA buffer in bytes

flags — Flags specifying properties of the SA buffer.

These may be:

SLAD_CACHE_COHERENT - The buffer is cache coherent

SLAD_NON_CACHE_COHERENT - The buffer is not cache coherent.

Registers an SA with the driver, which maintains mapping between handle and sa_buff for internal house-keeping. It simplifies the driver in the case where the SA is to be re-used. The client must provide this handle in the SLAD_PKT structure in slad_pkt_put/slad_cgx_put functions.

The 'bus_addr' field may be '0' if user does not know the physical address of the SA buffer. If the supplied address is not '0', the driver obtains the physical address of the buffer through OS specific functions. Thus, there are OS specific limitations on what kind of buffers the driver can compute physical addresses. The buffer provided by the user must be exactly of that type, otherwise the user must provide the physical address. Please also see the 'Virtual-to-Physical Address Translation' section in the OS specific appendix.

2.1.11 slad_register_srec

int slad_register_srec(int device_num, sa_handle *handle, void *srec_buff, slad_bus_addr bus_addr, int len, unsigned flags)

where:

device_num hardware device number

handle An opaque parameter to the caller which is used in slad_pkt_put/get functions. It should not be modified by the client.

sa_buff Address of the buffer containing the SA

srec_buff Virtual-address of the buffer storing State Record

bus_addr Bus Address of the State Record buffer

len Size of State Record buffer in bytes

flags Flags specifying properties of the SA buffer.

These may be:

SLAD_CACHE_COHERENT - The buffer is cache coherent

SLAD_NON_CACHE_COHERENT - The buffer is not cache coherent.

This function associates an Srec with an SA. This is done by passing the appropriate SA handle as a second parameter to this function.

A call to this function is not required when the SA does not require a State record.

This association is valid until slad_unregister_sa() is called. For 'bus_addr', similar restrictions apply to this buffer as to the SA buffer in the slad_register_sa() function call.

2.1.12 slad_unregister_sa

void slad_unregister_sa (sa_handle *handle)

where:

handle Handle returned from a successful call to slad_register_sa()

Un-registers the SA and its associated State Record from the driver. After this call, the SA or State Record can not be used in other functions, e.g. slad_pkt_put/get, etc.

2.1.13 slad_pkt_put

int slad_pkt_put (slad_app_id_type app_id, int device_num, SLAD_PKT pkt[], UINT32 *cnt)

where:

app_id	Application-id obtained from successful call to slad_pe_init()
device_num	hardware device number
pkt	pointer to populated array of SLAD_PKT structures
cnt	pointer to maximum/actual packet count

Enqueues packets onto the packet descriptor ring (PDR) of the specified device. Values from the caller supplied SLAD_PKT structures will be used to populate the next available packet descriptors, and those descriptors will be flagged as ready for processing.

A valid 'sa_handle' (which is a handle returned by the slad_register_sa() call, must be passed by the caller in the SLAD_PKT because it is used by the driver to identify the associated SA for the packet(s).

The caller sets cnt to the maximum number of packets to be enqueued, and upon exit this function will set cnt to the actual number of packets enqueued (which may be zero). It is important to check the value of cnt upon return from this function, because if it is less than the caller-supplied value of cnt, that means that some packets were not queued for processing, and they will not be returned by means of the slad_pkt_get() function. The user may either try to enqueue these packets again at a later time, or to discard them as an overrun condition.

Aside from an error condition, this function will return SLAD_DRVSTAT_SUCCESS even if no packets were enqueued (PDR full condition). However, it is possible for this function to return error status even if some of the packets have been successfully enqueued, so it is always important to check the value of cnt after this function returns, no matter what the return status is.

2.1.14 slad_pkt_get

int slad_pkt_get (slad_app_id_type app_id, int device_num, SLAD_PKT pkt[], int *cnt);

where:

app_id	Application-id obtained from successful call to slad_pe_init()
device_num	hardware device number
pkt	pointer to array of SLAD_PKT structures to be populated by this function
cnt	pointer to maximum/actual packet count

Dequeues one or more completed packets from the packet descriptor ring of the specified device.

If packets are ready to be dequeued, the results are retrieved from the completed packet descriptors and placed into the caller supplied SLAD_PKT structures. The caller sets cnt to the maximum number of packets to be dequeued, and upon exit this function will set cnt to the actual number of packets de-queued (which may be zero). The caller's pkt array should be large enough to hold the maximum number of packets specified by cnt.

2.1.15 slad_pkt_sync

int slad_pkt_sync (slad_app_id_type app_id, int device_num, SLAD_PKT *pkt);

where:

app_id Application-id obtained from successful call to slad_pe_init()

device_num hardware device number

pkt pointer to populated SLAD_PKT structure

Enqueues a single packet onto the packet descriptor ring (PDR) of the specified device, then waits for the same packet to be ready, then dequeues the packet. This function will not return until the packet has been processed and dequeued. This function is similar to making consecutive calls to slad_pkt_put() and slad_pkt_get(), except that it guarantees that the dequeued packet is the same one that was enqueued. This function will return SLAD_DRVSTAT_PDR_FULL if there was no room in the PDR to enqueue the packet.

2.1.16 slad_pkt_ready

int slad_pkt_ready (slad_app_id_type app_id, UINT32 *ready);

where:

app_id Application-id obtained from successful call to slad_pe_init()

ready pointer to flags, to be populated by this function

Gets the packet completion status for all devices in the system. The ready variable is a bit-mapped flag, with each bit representing one device in the system. Bit 0 represents device number 0; bit 1 represents device number 1, and so on. A bit set to 1 indicates that one or more packets entries are ready to be removed from the descriptor ring (with the slad_pkt_get() function) for the corresponding device.

Note that this function always returns SLAD_DRVSTAT_SUCCESS whether or not any devices are ready.

2.1.17 slad_bus_read

int slad_bus_read (int device_num, void *buf, int offset, int len);

where:

device_num hardware device number

buf pointer to destination buffer

offset source offset (in bytes) into chip bus memory space

len number of bytes to read

Reads some data directly from the device bus memory space.

The offset and length are not checked for validity, and the actual read operation is not verified. This command does not require the device to be initialized.

2.1.18 slad_bus_write

int slad_bus_write (int device_num, void *buf, int offset, int len);

where:

device_num hardware device number

buf pointer to source buffer

offset destination offset (in bytes) into chip bus memory space

len number of bytes to write

Writes some data directly to the device bus memory space.

The offset and length are not checked for validity, and the actual write operation is not verified. This command does not require the device to be initialized.

2.1.19 slad_allocate_buffer

int slad_allocate_buffer (void **handle, void **buf_addr, void **bus_add, void **buf, int len, int flags)

where:

handle Pointer to the opaque handle of the physical buffer

buf_addr Pointer to the virtual address of the allocated buffer

bus_addr Pointer to the bus address of the physical buffer

len length of buffer (in bytes) to allocate

flags Flags specifying coherency of the buffer to be allocated

SLAD_CACHE_COHERENT - make the buffer cache coherent

SLAD_NON_CACHE_COHERENT - do not make the buffer cache coherent

Allocates a physical (also known as DMA) buffer, which the user can manipulate by means of the slad_buf_copy_in() and slad_buf_copy_out() functions.

It is important to note that the buffer address returned by this function is the "bus address" of the buffer, which is the address used to access the buffer by the device when it is a bus master. Depending on your platform or operating system, this may or may not be the same as the address used to access the buffer by the host processor. For this reason, software should always use the slad_buf_copy_in() and slad_buf_copy_out() functions to read and write to this buffer. However, the bus address can be used to calculate pointers to locations within the buffer that are passed as parameters to the device that will access these locations as a bus master.

Be sure to check the return value of this function. Any value other than SLAD_DRVSTAT_SUCCESS indicates that the buffer was not allocated.

2.1.20 slad_free_buffer

slad_free_buffer (void **handle, int flags)

where:

handle handle to the physical memory to be freed

flags Flags specifying coherency of the buffer to be allocated

SLAD_CACHE_COHERENT - make the buffer cache coherent

SLAD_NON_CACHE_COHERENT - do not make the buffer cache coherent

Frees the physical memory allocated by the slad_buffer_allocate function.

2.1.21 slad_buffer_copy_in

int slad_buffer_copy_in (void *handle, void *in_buf, int offset, int len)

where:

handle Handle to the physical buffer

in_buf pointer to source buffer

offset destination offset (in bytes) into the allocated buffer

len number of bytes to copy

Copies some data from a caller supplied source buffer to the buffer that was previously allocated by the slad_allocated_buffer function. The handle was returned from the slad_allocated_buffer() function is used to refer to the buffer.

The offset and length are not checked for validity, and the actual copy operation is not verified.

2.1.22 slad_buffer_copy_out

int slad_buf_copy_out (void *handle, void *out_buf, int offset, int len)

where:

handle Handle to the physical buffer

out_buf pointer to destination buffer

offset source offset (in bytes) into the allocated buffer

len number of bytes to copy

Copies some data from the buffer that was previously allocated by the slad_buffer_allocate() function to a caller supplied destination buffer.

The offset and length are not checked for validity, and the actual copy operation is not verified.

The handle refers to the handle returned by the slad_allocate_buffer() function call.

2.1.23 slad_map_addr_range

int slad_map_addr_range (void *phy_addr, int len, void **mapped_addr)

where:

phy_addr Physical address to map

len number of bytes to map

mapped_addr Pointer to the virtual address to which the physical address will be mapped

Maps a physical address range to a virtual address so that it can be read and written by the host processor program. The virtual address obtained may be used to access the memory space. It can be useful when on-chip RAM is available, such a RAM can be mapped and used from the host processor program.

2.1.24 slad_unmap_addr_range

int slad_unmap_addr_range (void *mapped_addr)

where:

mapped_addr Mapped virtual address obtained from an earlier successful call to slad_map_addr_range()

Unmaps the memory obtained by mapping created by a call to slad_map_addr_range()

The memory is no longer unable until it is mapped again.

2.1.25 slad_get_random

int slad_get_random(slad_app_id_type app_id, int device_num, RANDOM_PARAM_BLK *arg)

where:

device_num hardware device number, must be '0'

app_id Application-id obtained from a successful call to slad_rng_init().

arg pointer to the RANDOM_PARAM_BLK

Generates a true random number of the required size.

The RANDOM_PARAM_BLK structure will contain the output buffer and size of the number.

2.1.26 slad_expmod

int slad_expmod (slad_app_id_type app_id, int device_num, EXPMOD_PARAM_BLK *info)

where:

device_num hardware device number, must be '0'

app_id Application-id obtained from a successful call to slad_pka_init.

info pointer to the EXPMOD_PARAM_BLK structure

Calculates the a ^ p mod m modular exponentiation.

Note: All operands in the EXPMOD_PARAM_BLK must be supplied in little-endian format. Refer to the PPC405/PPC460 User Manual for the hardware device to understand the input vector requirements/restrictions of the PKCP subsystem.

2.1.27 slad_expcrtmod

int slad_expcrtmod (slad_app_id_type app_id, int device_num, EXPCRTMOD_PARAM_BLK *iblk)

where:

app_id Application-id obtained from a successful call to slad_pka_init()

device_num hardware device number

iblk pointer to the EXPCRTMOD_PARAM_BLK

Calculates the RSA-CRT Modular Exponentiation (see *RSA-CRT Modular Exponentiation* on page 13).

The input data, the half-length modulus vectors (MODP and MODQ), the exponential vectors (DP and DQ), and the Qinv vectors must be supplied by the caller. All operands in the EXPCRTMOD_PARAM_BLK must be supplied in little-endian format.

The result obtained will be returned in the res buffer of the EXPCRT_PARAM_BLK structure. The caller needs to check the first dword-aligned zero in the res buffer to find the actual result.

2.2 Function Return Codes

The possible return values for each of the SLAD functions are indicated by the SLAD_DRVSTAT_xxxx values defined in slad.h. Note that the status returned by the driver functions is not the same as the status placed into the packet descriptor by the device after descriptor processing.

Table 2-2. Function Return Codes

Status Code	Description
SLAD_DRVSTAT_SUCCESS	Command successful
SLAD_DRVSTAT_COMMAND_INVALID	An invalid SLAD_DRVCMD_xxxx value was specified. This applies only when making SLAD calls from a user-mode application.
SLAD_DRVSTAT_DEVICE_INVALID	The value of device_num is out of range, either because it is less than zero, or greater than the maximum number of devices supported by the driver.
SLAD_DRVSTAT_DEVICE_NOT_FOUND	The specified device_num is not present in this system.
SLAD_DRVSTAT_DEVICE_NOT_INIT	The specified device_num has not been initialized.
SLAD_DRVSTAT_PDR_FULL	There is no more room in this device's packet descriptor ring to enqueue another entry.
SLAD_DRVSTAT_MALLOC_ERR	Memory could not be allocated or remapped.
SLAD_DRVSTAT_UPLOAD_ERR	Upload of device firmware failed.
SLAD_DRVSTAT_INIT_FAIL	General device initialization fault.
SLAD_DRVSTAT_PDR_EMPTY	There are no entries in this device's packet descriptor ring (PDR) that are ready to be dequeued.
SLAD_DRVSTAT_GDR_FULL	There is no more room in this device's gather particle descriptor ring to enqueue another entry.
SLAD_DRVSTAT_IOCTL_ERR	An error occurred during command processing through the IOCTL interface. This applies only when making SLAD calls from a user-mode application.
SLAD_DRVSTAT_USERMODE_API_ERR	The file for IOCTL could not be accessed. This applies only when making SLAD calls from a user-mode application.
PE_INIT_BLOCK Parameter Errors (See Note)	
SLAD_DRVSTAT_BAD_PARAM_PDR_BUSID	
SLAD_DRVSTAT_BAD_PARAM_PDR_ENTRIES	
SLAD_DRVSTAT_BAD_PARAM_PDR_POLL_DELAY	
SLAD_DRVSTAT_BAD_PARAM_PDR_DELAY_AFTER	
SLAD_DRVSTAT_BAD_PARAM_PDR_INT_COUNT	
SLAD_DRVSTAT_BAD_PARAM_PDR_OFFSET	
SLAD_DRVSTAT_BAD_PARAM_SA_BUSID	
SLAD_DRVSTAT_BAD_PARAM_SA_ENTRIES	
SLAD_DRVSTAT_BAD_PARAM_SA_CONFIG	
SLAD_DRVSTAT_BAD_PARAM_PAR_SRC_BUSID	
SLAD_DRVSTAT_BAD_PARAM_PAR_SRC_SIZE	
SLAD_DRVSTAT_BAD_PARAM_PAR_DST_BUSID	
SLAD_DRVSTAT_BAD_PARAM_PAR_DST_SIZE	
SLAD_DRVSTAT_BAD_PARAM_PAR_CONFIG	
SLAD_DRVSTAT_BAD_PARAM_OFFSET	
Note: The PE_INIT_BLOCK parameter errors indicate invalid parameters in either the PE_INIT_BLOCK structure or API command parameters. The names should be self-explanatory. Please also see platform specific details in Appendix A.	

3. Data Structures

This chapter describes the various data structures used by the Security Look-aside Driver Module. These structures are defined in the slad.h header file, except for PE_INIT_BLOCK, which is defined in initblk.h. The platform-specific sections should also be examined in order to determine if there have been any changes to the data structures listed in this section.

3.1 SLAD_DEVICEINFO

This data structure is returned in response to a slad_device_info() function call.

```
typedef struct {
   UINT32 device_num;
   UINT32 device_type;      /* one of the SLAD_DEVICETYPE_xxxx defs */
   UINT32 base_addr;        /* base memory address (hardware) */
   VPTR base_addr_mapped    /* base memory address (virtual/mapped) */
   UINT32 addr_len;         /* size of memory space, in bytes */
   UINT32 vendor_id;        /* PCI vendor id */
   UINT32 device_id;        /* PCI device id */
   UINT32 features;         /* bits defined by DEVICEINFO_FEATURES_xxxx */
   UINT32 crypto_algs;      /* bits defined by DEVICEINFO_CRYPTO_ALGS_xxxx */
   UINT32 crypto_mode;      /* bits defined by DEVICEINFO_CRYPTO_MODE_xxxx */
   UINT32 crypto_feedback;  /* bits defined by DEVICEINFO_CRYPTO_FEEDBACK_xxxx */
   UINT32 hash_algs;        /* bits defined by DEVICEINFO_HASH_ALGS_xxxx */
   UINT32 comp_algs;        /* bits defined by DEVICEINFO_COMP_ALGS_xxxx */
   UINT32 pkt_ops;          /* bits defined by DEVICEINFO_PKT_OPS_xxxx */
   UINT32 pkt_features;     /* bits defined by DEVICEINFO_PKT_FEATURES_xxxx */
} SLAD_DEVICEINFO;
```

The SLAD_DEVICETYPE_xxxx values and DEVICEINFO_xxx bit masks are all defined in slad.h.

3.2 PE_INIT_BLOCK

During the device initialization process invoked by slad_device_init(), an Initialization Block must be passed from the application to the driver to configure device options, set-up the locations of descriptor rings, and so on. A common PE_INIT_BLOCK exists for all device types. Be aware that some of the items in the PE_INIT_BLOCK are not used for all device types.

This structure, or any substructures it contains, do not need to persist outside of the call to slad_device_init().

```
typedef struct {
     UINT32 cdr_busid;
     VPTR cdr_addr;
     UINT16 cdr_entries;
     UINT16 hpcdr_entries;
     UINT16 cdr_poll_delay;
     UINT16 cdr_delay_after;
     UINT32 sa_config;
     UINT32 sa_entries;
     UINT32 sa_busid;
     VPTR sa_addr;
```

```
UINT32 token_busid;
VPTR token_addr;
UINT16 cdr_int_count;
UINT16 cdr_int_type;
UINT16 online_int_type;
UINT16 fatalerror_int_type;
UINT16 resetack_int_type;
UINT16 pf_active_low_int;
UINT16 reserved;
UINT16 max_cgx_pci_burst;
UINT32 dma_config;
UINT32 target_read_count;
UINT16 pe_endian_mode;
UINT16 target_endian_mode;
UINT32 pe_dma_config;
VPTR pdr_addr;
VPTR pdrr_addr;
UINT16 pdr_entries;
UINT16 pdr_offset;
UINT16 pe_dma_input_threshold;
UINT16 pe_dma_output_threshold;
UINT16 dram_config;
UINT16 ext_map;
UINT16 ext_memcfg;
UINT16 refresh_timer;
UINT16 ext_mem_wait;
UINT16 pdr_poll_delay;
UINT16 pdr_delay_after;
VPTR part_src_addr;
VPTR part_dst_addr;
UINT16 part_src_entries;
UINT16 part_dst_entries;
UINT32 part_config;
UINT32 device_block_busid;
UINT32 device_block_addr;
UINT32 pdr_int_count;
SLAD_NOTIFY *pdr_notify;
SLAD_NOTIFY *cdr_notify;
SLAD_NOTIFY *exp0_notify;
SLAD_NOTIFY *exp2_notify;
SLAD_NOTIFY *pkcp_notify;
UINT32 int_config;
UINT32 bus_id_config;
UINT16 user_boot_control;
UINT16 user_boot_interrupt_to_force;
UINT32 user_boot_signblock_busid;
VPTR user_boot_signblock_addr;
UINT32 intsrc_mailbox_busid;
VPTR intsrc_mailbox_addr;
UINT32 software_timer_busid;
```

```
VPTR software_timer_addr;
UINT32 target_delay_specified;
UINT32 target_read_interval;
UINT32 target_read_delay;
UINT32 target_write_interval;
UINT32 target_write_delay;
UINT32 misc_options;
interrupt_pid interrupt_callout;
UINT32 pdr_time_out_cnt;
```

Note: (There are additional items in the PE_INIT_BLOCK beyond this point, but they are used only for other initialization and ignored by the SLAD.)

```
}PE_INIT_BLOCK;
```

Table 3-1. PE_INIT_BLOCK Element Values

Parameter	Description
cdr_busid	Bus ID for the CGX Descriptor Ring. One of the SLAD_BUSID_xxxx[4] definitions.
cdr_addr	Base address of CGX Descriptor Ring (CDR). If this value is 0, and cdr_busid is equal to SLAD_BUSID_HOST, the SLAD will allocate the space for the CDR, and the address of the allocated space will be written back here in the PE_INIT_BLOCK. If non-zero, this address must point to contiguous physical memory aligned on a dword boundary.
cdr_entries	Number of CGX Descriptors in the CDR.
cdr_poll_delay	Interval between CGX Descriptor polls, once an empty descriptor is encountered. The units of this timing interval are device-type dependent.
cdr_delay_after	Interval until the next CGX Descriptor poll, immediately after a descriptor is processed. The units of this timing interval are device-type dependent.
sa_config	If set to 0, the host will manage the SA database. If set to 1, the device will manage the SA database by means of CGX commands.
sa_entries	Number of entries in the SA database. Used only if sa_config is set to 1.
sa_busid	Bus ID for the SA database. One of the SLAD_BUSID_xxxx[4] definitions.
sa_addr	Base address of SA database. Used only if sa_config is set to non-zero. If this value is 0, and sa_bus id is equal to SLAD_BUSID_HOST, and sa_config is non-zero, the driver will allocate the space for the SA database, and the address of the allocated space will be written back here in the PE_INIT_BLOCK. If supplied, this address must point to contiguous physical memory aligned on a dword boundary.
token_busid	Bus ID for the token. One of the SLAD_BUSID_xxxx[4] definitions.
token_addr	Base address of the token.
cdr_int_count	Specifies how many CDR entries must be processed before generating a host interrupt.
cdr_int_type	See documentation of device(s) supporting it.
online_int_type	See documentation of device(s) supporting it.
fatalerror_int_type	See documentation of device(s) supporting it.
resetack_int_type	See documentation of device(s) supporting it.
pf_active_low_int	Unused with SLAD applications, although still applicable for legacy applications.
max_cgx_pci_burst	Unused with SLADapplications, although still applicable for legacy applications.

Table 3-1. PE_INIT_BLOCK Element Values (Continued)

dma_config	Value for the device DMA CONFIG register.
target_read_count	Value for the device TARG RDCNT register.
pe_endian_mode	Value for the device PE ENDIAN MODE register.
target_endian_mode	Value for the device TARG ENDIAN MODE register.
pe_dma_config	Value for the device PE DMA CONFIG register. Note that the PDR bus ID is specified by bits 4 and 5 of this value.
pdr_addr	Base address of Packet Descriptor Ring (PDR). If this value is 0, and the PDR bus ID (as specified by bits 4 and 5 of pe_dma_config) is equal to SLAD_BUSID_HOST[4], the driver will allocate the space for the PDR, and the address of the allocated space will be written back here in the PE_INIT_BLOCK. If non-zero, this address must point to contiguous physical memory aligned on a dword boundary.
pdrr_addr	Unused with SLAD applications, although still applicable for legacy applications.
pdr_entries	Number of Packet Descriptors in the PDR.
pdr_offset	Size (in dwords) of each PDR entry.
pe_dma_input_threshold	Value for the device PE DMA INPUT THRESHOLD register.
pe_dma_output_threshold	Value for the device PE DMA OUTPUT THRESHOLD register.
dram_config	Value for the device DRAM CONFIG register.
ext_map	Value for the device EXT MAP register.
ext_memcfg	Value for the device EXT MEM CFG register.
refresh_timer	Value for the device DRAM REFRESH TIMER register.
ext_mem_wait	Value for the device EXT MEM WAIT register.
pdr_poll_delay	Interval between Packet Descriptor polls, immediately after a descriptor is processed. The units of this timing interval are device-type dependent.
pdr_delay_after	Interval until the next Packet Descriptor poll, once an empty descriptor is encountered. The units of this timing interval are device-type dependent.
part_src_addr	Base address of Gather Particle Descriptor Ring (GDR). If this value is 0, the gather feature will not be used. If non-zero, this address must point to a contiguous physical bus memory address aligned on a dword boundary.
part_dst_addr	Base address of Scatter Particle Descriptor Ring (SDR). If this value is 0, the scatter feature will not be used. If non-zero, this address must point to a contiguous physical bus memory address aligned on a dword boundary.
part_src_entries	Number of gather particles in the GDR.
part_dst_entries	Number of scatter particles in the SDR.
part_config	Value for the PE PART CFG register.
pdr_int_count	Specifies how many PDR entries must be processed before generating a host interrupt.
pdr_notify	Pointer to SLAD_NOTIFY structure, which specifies the method to be used for notifying the host that one or more entries are ready to be removed from the PDR.
cdr_notify	Pointer to SLAD_NOTIFY structure, which specifies the method to be used for notifying the host that one or more entries are ready to be removed from the CDR.
exp0_notify	Pointer to SLAD_NOTIFY structure, which specifies the method to be used for notifying the host that the channel 0 exponentiator is finished.
exp2_notify	Pointer to SLAD_NOTIFY structure, which specifies the method to be used for notifying the host that the channel 2 exponentiator is finished.

Table 3-1. PE_INIT_BLOCK Element Values (Continued)

pkcp_notify	Pointer to SLAD_NOTIFY structure, which specifies the method to be used for notifying the host that the public key co-processor is finished.
int_config	Value for the device INT CONFIG register.
bus_id_config	Value for the device BUS ID CONFIG register.
user_boot_control	User Boot is not currently supported by the SLAD.
user_boot_interrupt_to_force	User Boot is not currently supported by the SLAD.
user_boot_signblock_busid	User Boot is not currently supported by the SLAD.
user_boot_signblock_addr	User Boot is not currently supported by the SLAD.
intsrc_mailbox_busid	Bus ID for the intsrc mailbox. One of the SLAD_BUSID_xxxx[4] definitions.
intsrc_mailbox_addr	Base address of intsrc mailbox. If this value is 0, and intsrc_mailbox_busid is equal to SLAD_BUSID_HOST[4], the driver will allocate the space for the intsrc mailbox, and the address of the allocated space will be written back here in the PE_INIT_BLOCK. If non-zero, this address must point to contiguous physical memory aligned on a dword boundary.
software_timer_busid	Unused with SLAD applications, although still applicable for legacy applications.
software_timer_addr	Unused with SLAD applications, although still applicable for legacy applications.
target_delay_specified	This is a mechanism for allowing the application to override the default target access delays (if any) that are built into the SLAD for certain devices. If target_delay_specified is set FALSE, the built in delays are used, and if set TRUE, the specified interval and delay values are used (these are the next four values in the PE_INIT_BLOCK). To specify no delay at all, set both the interval and delay to zero.
target_read_interval	Number of consecutive target reads between each target_read_delay. Ignored if target_delay_specified is FALSE.
target_read_delay	Target read delay (in microseconds). Ignored if target_delay_specified is FALSE.
target_write_interval	Number of consecutive target reads between each target_write_delay. Ignored if target_delay_specified is FALSE.
target_write_delay.	Target write delay (in microseconds). Ignored if target_delay_specified is FALSE
misc_options	A bit map of various SLAD options. The bits are defined by the SLAD_MISC_OPTIONS_xxx [5] definitions in the initblk.h header file. See the comments in initblk.h for an explanation of all of the current options.
pdr_time_out_cnt	Time out counter
pe_mode	Mode configuration settings for the packet engine

Note 1: This bus ID value also includes endian configuration information in the upper byte.

Note 2: This bus ID value must be set to SLAD_BUSID_HOST.

Note 3: For the 2141, which does not physically have this register, bits 4 and 5 of this value will be used to derive the PDR bus ID.

Note 4: SLAD_BUSID_xxx are defined in the source file slad.h.

Note 5: SLAD_MISC_OPTIONS are defined in the source file slad.h

3.3 PKA_INIT_BLOCK

```
typedef struct {
   //to be completed
}PKA_INIT_BLOCK;
```

3.4 RNG_INIT_BLOCK

```
typedef struct {
      //to be completed
   }RNG_INIT_BLOCK;
```

3.5 SLAD_NOTIFY

There are several of these sub-structures within the PE_INIT_BLOCK structure. These O/S and platform-dependent parameters specify the method for notifying the host that a specific event has occurred, e.g., the completed processing of a packet or CGX command.

```
typedef struct {
   UINT32 process_id;
   UINT32 signal_number;
   void (*callback)(int device_num);
} SLAD_NOTIFY;
```

Table 3-2. SLAD Notify

Term	Definition
process_id	The process ID of the process where the signal will be sent
signal_number	Signal number to send for notification to the process identified by process_id. If zero, no signal will be sent.
callback	Function to call for notification. If NULL, no callback will be made.

3.6 SLAD_PKT

```
typedef struct {
   UINT32 control_status;
   VPTR src;
   VPTR dst;
   VPTR sa;
   UNIT32 sa_len
   UINT32 len_control2;
   VPTR user_handle;
   VPTR srec;
   slad_bus_addr  src_bus_addr;
   slad_bus_addr  dst_bus_addr;
   UINT32  dst_len;
   UINT32  flags;
} SLAD_PKT;
```

Table 3-3. SLAD PKT

Term	Definition
control_status	This is the same as the 32-bit Control/Status field as defined in the Packet Engine Descriptor. This value is supplied by the caller when putting packets, and supplied by the driver when getting packets. The use of this field is transform and device dependent.
src	This is a pointer to the packet source address. This value is supplied by the caller when putting packets, and supplied by the driver when getting packets. The value supplied by the driver after the get will be the same as the value originally supplied by the caller during the put of the corresponding packet.
dst	This is a pointer to the packet Destination Address. This value is supplied by the caller when putting packets, and supplied by the driver when getting packets. The value supplied by the driver after the get will be the same as the value originally supplied by the caller during the put of the corresponding packet.
sa	This is a pointer to the Security Association (SA). This value is supplied by the caller when putting packets, and supplied by the driver when getting packets. The value supplied by the driver after the get will be the same as the value originally supplied by the caller during the put of the corresponding packet.
sa_len	Used when specifying "Dynamic SA", this field contains the length in bytes of the "Dynamic SA".
len_control2	This is the same as the 32-bit Length/Control2 field as defined in the Packet Engine Descriptor. This value is supplied by the caller when putting packets, and supplied by the driver when getting packets. The use of this field is transform and device dependent
user_handle	This is a convenient general-purpose variable that may be used by the caller's application. The caller supplies this value when putting a packet; and that same value will be written back here by the driver when getting the same packet. This value is never referenced or altered by the driver.
srec	This is a pointer to the State Record. This value is supplied by the caller when putting packets, and supplied by the driver when getting packets. The value supplied by the driver after the get will be the same as the value originally supplied by the caller during the put of the corresponding packet. This value can remain un-initialized for packets that do not use a state record.
src_bus_addr	Physical address of source buffer or zero. When the user does not know the physical address this should be set to '0'. If the supplied address is not '0', the driver obtains the physical address of the buffer through OS specific functions. So there are OS specific limitations on the buffers that the driver is capable of computing physical addresses. The buffer provided by the user must be of the appropriate type otherwise the user must provide the physical address.
dst_bus_addr	Physical address of the destination buffer or zero. The same restrictions that apply to the above src_bus_addr also apply to this field.
dst_len	The length of the destination buffer (dst).
flags	Unused in the current version of the SLAD.

```
/* Same as SLAD_PKT, but using bitfields for the control words. */
typedef struct {
      UINT32 pad_control:8;                                  /* 31-24 */
      UINT32 status:8;                                       /* 23-16 */
      UINT32 next_header:8;                                  /* 15-08 */
      UINT32 sa_busid:2;                                     /* 07-06 */
      UINT32 chain_sa_cache:1;                               /* 05    */
      UINT32 hash_final:1;                                   /* 04    */
      UINT32 init_stateful_arc4:1;                           /* 03    */
       UINT32 load_sa_digests:1;                              /* 02    */
       UINT32 done1:1;                                        /* 01    */
       UINT32 ready1:1;                                       /* 00    */
       VPTR src;
       VPTR dst;
       VPTR sa;
       UINT32 sa_len
       UINT32 bypass_offset:8;                                /* 31-24 */
       UINT32 done2:1;                                        /* 23    */
       UINT32 ready2:1;                                       /* 22    */
       UINT32 reserved2:2;                                    /* 21-20 */
       UINT32 len:20;                                         /* 19-00 */
       VPTR user_handle;
       VPTR srec;
       slad_bus_addr src_bus_addr;
       slad_bus_addr dst_bus_addr;
       UINT32 dst_len;
       UINT32 flags;
} SLAD_PKT_BITS;
```

Packet Engine Descriptor is the definition of the packets as contained within the Packet Descriptor Ring (PDR)

Note: Note: When descriptors are fetched from the host memory locations the descriptor must be set to little-endian.

3.7 Security Association (SA) Record Format

SLAD_SA

```
typedef struct {
//SA_Command_0
UINT32              output_scatter:1;
UINT32              input_gather:1;
UINT32              save_hash:1;
UINT32              save_iv:1;
UINT32              hash_loading:2;
UINT32              iv_loading:2;

UINT32              digest_len:4;
UINT32              header_proc:1;
UINT32              ext_pad:1;
UINT32              stream_cipher_pad:1;
UINT32              reserved0:1;

UINT32              hash_algo:4;
UINT32              crypto_algo:4;

UINT32              crypto_pad:2;
UINT32              op_code:6;

//SA_COMMAND_1
UINT32              offset:8;

UINT32              rev:2; //00-->rev0, 10-->rev1, 01-->rev2, 11-->reserved
UINT32              byte_offset:1;
UINT32              hmac:1;
UINT32              crypto_feedback:2;
UINT32              crypto_mode:2;

UINT32              ext_seq_num:1;
UINT32              seq_num_mask:1;
UINT32              mutable_bits:1;
UINT32              ipv6:1;
UINT32              copy_pad:1;
UINT32              copy_payload:1;
UINT32              copy_header:1;
UINT32              use_red_keys:1;

BYTE               salt[8];

BYTE               key1[8];
BYTE               key2[8];
BYTE               key3[8];
BYTE               key4[8];
BYTE               inner[20];
BYTE               outer[20];
UINT32             spi;
volatile UINT32    seq;
BYTE               seq_mask[8]
```

```
UINT32              cpi_size;
volatile UINT32     srec;
UINT32              ij;
volatile UINT32     srec_arc4;
volatile UINT32     management0;
volatile UINT32     management1;
}SLAD_SA_REV1;
```

A Security Association record along with the packet descriptor, provides the Packet Engine with all of the necessary information to process an operation.

3.8 State Record

```
typedef struct {
   volatile BYTE IV[16];
   volatile UINT32 HashByteCount;
   volatile BYTE InnerDigest[20];
}  SLAD_STATE_RECORD_REV1
```

Table 3-4. State Record

Term	Definition
IV	Initialization Vectors 0-4[32bit each]
Hash BYTE Count	Starting hash byte count
Inner Digest	Starting hash state

3.9 RANDOM_PARAM_BLK

```
typedef struct {
   unsigned char *output;
   unsigned int size;
} RANDOM_PARAM_BLK;
```

Table 3-5. Random Param Blk

Term	Definition
char *output	Pointer to the buffer where the random number generated will be stored.
int size	Size of the random number generated in bytes. It cannot be longer than 256 bytes.

3.10 EXPMOD_PARAM_BLK

```
typedef struct {
   unsigned int *res;
   unsigned int ressize;
   unsigned int *a;
   unsigned int asize;
   unsigned int *p;
   unsigned int psize;
   unsigned int *m;
   unsigned int msize;
} EXPMOD_PARAM_BLK;
```

where:

res Pointer to the result of the exponential modular calculation obtained by a ^ p mod m.

The a, p, and m are to be represented in little-endian encoding, irrespective of the device. Further, these cannot be longer than 256 bytes.

Note: The caller needs to specify the values for a, p, and m.

Table 3-6. EXPMOD Param Blk

Term	Definition
res	Pointer to the result of the exponential modular calculation obtained by a ^ p mod m.
ressize	Length of block to hold the results of the exponentiation
*a	Pointer to parameter a of the exponential
asize	Length of block that contains the parameter a
*p	Pointer to the parameter p of the exponential
psize	Length of block that contains the parameter p
*m	Pointer to the parameter m of the exponential
msize	Length of block that contains the parameter m

3.11 EXPCRT_PARAM_BLK

```
typedef struct {
   unsigned int *res;
   unsigned int ressize;
   unsigned int *a;
   unsigned int asize;
   unsigned int *p;
   unsigned int psize;
   unsigned int *q;
   unsigned int qsize;
   unsigned int *dp;
   unsigned int dpsize;
   unsigned int *dq;
   unsigned int dqsize;
   unsigned int *qinv;
   unsigned int usize;
} EXPCRTMOD_PARAM_BLK;
```

Table 3-7. EXPCRT Param Blk

Term	Definition
ressize	Length of block to receive the result after exponentiation
*a	Pointer to the parameter a of the exponentiation
asize	Length of the block containing the parameter a
*p	Pointer to the parameter p of the exponentiation
psize	Length of the block containing the parameter p
*q	Pointer to the parameter q of the exponentiation
qsize	Length of the block containing the parameter q
*dp	Pointer to the parameter dq of the exponentiation
dpsize	Length of the block containing the parameter dp
*qinv	Pointer to the parameter qinv of the exponentiation
usize	Length of the block containing the parameter qinv

Note: The caller needs to specify the values for a, p, q, dp, dq, and qinv are to be represented in little-endian encoding, irrespective of the device. Further, these cannot be longer than 256 bytes.

Appendix A. Linux Platform Specifics

A.1 Introduction

In the Linux environment, the SLAD is implemented as a loadable kernel module, compatible with Linux kernel versions 2.6. Kernel versions below 2.6 are not supported. The makefile supplied with the SLAD source code can be used with Linux kernel version 2.6.x only.

After building the Linux version of the SLAD from the source code, the resultant file (named slad.ko) is produced. Two shell scripts are provided to load and unload the driver:

./slad_load.sh

./slad_unload.sh

A.2 Kernel-Mode Interface

Note: The slad_load.sh script processing requires that the system has the /proc filesystem and the awk utility.

The kernel-mode interface is simple. The SLAD API functions are all exported and available for use by any software running in kernel mode.

A.3 Kernel-Mode Interface

The kernel mode interface is simple. The SLAD API functions are all exported and available for use by any software running in kernel mode.

A.4 Kernel-Mode User Application supplied buffers

All buffers supplied by kernel mode applications to SLAD must be DMA safe buffers. Also kernel mode applications must supply the bus address of these buffers in the SLAD_PKT (for the packet source and destination buffers) structure.

A.5 User-Mode Interface

The Linux SLAD uses a file I/O write interface to provide user applications with the API. However, the user-mode helper functions (see "SLAD API Function Summary") make the file I/O interface transparent to the user.

The load/unload scripts mentioned in the introduction to this appendix automatically handle creation of the "special character file" device node /dev/slad, so a mknod command is not necessary.

A.6 Build Instructions

The driver package contains both the driver and test application source code as well as the kernel and user mode binaries. Please refer to the README.TXT files for detailed instructions.

Driver Build Instructions are contained in:

slad/README.TXT

The Makefile for the driver is in

slad/build/Makefile

Please ensure that the KDIR and CROSS_COMPILER_PREFIX variables in the Makefile is set to point to the correct linux kernel (KDIR or kernel directory) and tool chain directory respectively.

A.7 Test Environment

Please refer to the Driver Release Notes for the Linux kernel version and hardware platform used for testing the driver.

A.8 Virtual-to-Physical Address Translation

The driver can obtain the physical address of Source/Destination, SA and State Record buffers, given their virtual addresses; if these buffers are allocated through kmalloc(). For buffers allocated by any other kernal memory allocation function like 'ioremap(), cache-coherent memory allocation by dma_alloc_coherent()' etc, the user should provide the physical address whenever required by the driver API functions.

For user-mode applications, the driver allocates bounce buffers and can compute the required physical addresses of these buffers internally.

A.9 Bounce Buffer Allocation

Since the device accesses physical addresses with the DMA controller, these buffers must be DMA-SAFE. The driver guesses that buffer is DMA-SAFE if it is aligned at cache-line size from beginning to end. If the driver detects that a buffer is not DMA-SAFE, it allocates a cache aligned buffer internally and copies the content of the original buffer. These types of buffers are referred to as "Bounce Buffers".

When the user is certain that the buffers provided to the driver (in slad_pkt_put/get API calls), are already DMA-SAFE, the driver can be configured to not allocate bounce-buffers.

To configure the driver not to allocate bounce buffers ensure the line below is active, not commented out, in the file $(SLAD_INSTALL_DIR)/os/inc/slad_osal.h. Where $(SLAD_INSTALL_DIR) is the directory that the driver source code was installed into.

Appendix B. Security Co-Processor v2.2 Specifics

This appendix describes the SLAD features specific to the Security Co-Processor v2.2 security device.

B.1 Single Device Support

The SLAD supports a single security device. The API functions requiring a device number as a parameter must be supplied with the value zero, '0'.

B.2 PE_INIT_BLOCK Elements

Table 3-1 shows the PE_INIT_BLOCK elements used by the security device.

sa_busid	Bus ID for the SA database. One of the SLAD_BUSID_xxx definitions, see header file slad.h for definitions.
dma_config	Value for the device DMA_CONFIG register
pe_endian_mode	Value for the device PE_ENDIAN_MODE register
pe_dma_config	Value for the device PE_DMA_CONFIG register. Note that the PDR bus ID is specified by bits 4 and 5 of this value.
pdr_addr	Base address of the Packet Descriptor Ring (PDR). If this value is 0, and the PDR bus ID, as specified by bits 4 and 5 of the PE_DMA_CONFIG register, is equal to SLAD_BUSID_HOST, the driver will allocate the space for the PDR, and the address of the allocated space will be written back here to the PE_INIT_BLOCK. If this value is non-zero, this address must point to a contiguous physical block of memory aligned on a dword boundary.
pdr_entries	Number of packet descriptors to be created in the PDR.
pdr_offset	Size, in dwords, of each PDR entry.
pe_dma_input_threshold	Value of the PE_DMA_INPUT_THRESHOLD register
pe_dma_output_threshold	Value of the PE_DMA_OUTPUT_THRESHOLD register
pdr_poll_delay	Interval between packet descriptor polls, immediately after a descriptor is processed. The units of this timing interval are device dependent.
pdr_delay_after	Interval until the next packet descriptor poll, once an empty descriptor is encountered. The units of this timing interval are device dependent.
part_src_addr	Base address of the Gather Particle Descriptor Ring (GDR). If this value is zero, the gather feature will not be used. If non-zero, this address must point to a contiguous block of physical memory aligned on a dword boundary
part_dst_addr	Base address of the Scatter Particle Descriptor Ring (SDR). If this value is zero, the scatter feature will not be used. If non-zero, this address must point to a contiguous block of physical memory aligned on a dword boundary
part_src_entries	Number of gather particles in the GDR
part_dst_entries	Number of scatter particles in the SDR
part_config	Value for the PE_PART_CONFIG register
pdr_init_count	Specifies how many PDR entries must be processed before generating a host interrupt
int_config	Value for the device INT_CONFIG register

target_delay_specified	Method of allowing the application to override the default target access delays (if any) that are built into the SLAD for certain devices. If this value is set to FALSE (0), the built in delays will be used, and if set to TRUE (1), the specified interval and delay values are used (these are the next 4 values in the PE_INIT_BLOCK). To specify no delay at all, set both the interval and delay values to zero
target_read_interval	Number of consecutive target device reads between each target_read_delay. Ignored if target_delay_specified is FALSE (0)
target_read_delay	Target device read delay (in microseconds). Ignored if target_delay_specified is FALSE(0)
target_write_inteval	Number of consecutive target device writes between each target_write_delay. Ignored if target_delay_specified is FALSE(0)
target_write_delay	Target device write delay (in microseconds). Ignored if target_delay_specified is FALSE(0)
misc_options	A bit-map of various SLAD options. The bits are defined by the SLAD_MISC_OPTIONS_xxx definitions in the initblk.h header file. See the comments in that file for an explanation of the current options
pdr_time_out_cnt	Number of clock cycles befora issuing a Timeout interrupt
pe_mode	Mode configuration settings for the packet engine
enable_dynamic_sa	Must be set to '1, if dynamic SA is to be used

Table 3-8. SLAD_BUSID_xxx Definitions

SLAD_BUSID_EMI	0x0000 0000
SLAD_BUSID_HOST	0x0000 0001
SLAD_BUSID_INTERNAL	0x0000 0002
SLAD_BUSID_DISABLED	0x8000 0000

Index

T

U, V, W

Revision Log

Revision Date	Level	Contents of Modification
02/28/2008	1.00	Initial document creation.