Security Engine 2.x Reference Device Driver User's Guide

For SEC 2.x Device Driver Version 1.6

1 Overview

This document is a user's guide for the SEC 2.x Reference Device Driver, version 1.6. SEC 2.x refers to the integrated security engine found in most members of the PowerQUICC™ II Pro and PowerQUICC III device families.

The SEC 2.x Reference Device Driver, version 1.6 is a superset device driver including all the routines necessary to operate the acceleration features available in the SEC 2.1, the most capable SEC core found in PowerQUICC products today. The SEC 2.0, 2.2 and 2.4 cores offer a subset of SEC 2.1 functionality, and consequently, not all the examples of application interaction with the SEC 2.x core will apply to all PowerQUICC devices.

Table 1 shows the SEC core version found in each member of the PowerQUICC product family. Consult the individual PowerQUICC device's reference manual to determine which SEC core and functions are applicable to your environment.

Contents

1. Overview 1
2. Device Driver Components 4
3. User Interface 6
4. Individual Request Type Descriptions 16
5. Sample Code 45
6. Linux Environment 47
7. VxWorks Environment 49
8. Porting 50
9. Revision History 53

freescale™ semiconductor

Table 1. Product and SEC Core Version

Product Family	Device	SEC Core Version	SEC 2.x Driver Features Not Supported in Silicon
PowerQUICC II Pro	MPC8323E	SEC 2.2	Public Key, Kasumi, RNG, ARC-4
	MPC8343E, MPC8347E, MPC8349E (Rev 1.1)	SEC 2.0	Kasumi, XOR, single descriptor SSL/TLS
	MPC8343EA, MPC8347EA, MPC8349EA (Rev 3)	SEC 2.4	Kasumi
	MPC8358E, MPC8360E (Rev 1)	SEC 2.0	Kasumi, XOR, single descriptor SSL/TLS
	MPC8358EA, MPC8360EA (Rev 2)	SEC 2.4	Kasumi
PowerQUICC III	MPC8541E, MPC8555E	SEC 2.0	Kasumi, XOR, single descriptor SSL/TLS
	MPC8543E, MPC8545E, MPC8547E, MPC8548E	SEC 2.1	

The driver is coded in ANSI C. In it's design, an attempt has been made to write a device driver that is as operating system agnostic as practical. Where necessary, operating system dependencies are identified and the "Porting" section, addresses them.

Testing has been accomplished on VxWorks 5.5 and LinuxPPC using kernel version 2.6.11.

Application interfaces to this driver are implemented through the `ioctl()` function call. Requests made through this interface can be broken down into specific components, including miscellaneous requests and process requests. The miscellaneous requests are any requests not related to the direct processing of data by the SEC core.

Process requests comprise the majority of the requests and all are executed using the same `ioctl()` access point. In the section entitled "Process Request Structures," structures needed to compose these requests are described in detail.

Throughout the document, the acronyms CHA (crypto hardware accelerator) and EU (execution unit) are used interchangeably. Both acronyms indicate the device's functional block that performs the crypto functions requested.

The reader should understand that the design of this driver is a legacy holdover from two prior generations of security processors. As applications have already been written for those processors, certain aspects of the interface for this driver have been designed so as to maintain source-level application portability with prior driver/processor versions. Where relevant in this document, prior-version compatibility features will be indicated to the reader.

Table 2 contains acronyms and abbreviations that are used in this user's manual.

Table 2. Acronyms and Abbreviations

Term	Meaning
AESA	AES accelerator—This term is synonymous with AESU in the Security Engine 2.x chapter of the PowerQUICC device reference manuals and other related documentation.
AFHA	ARC-4 hardware accelerator—This term is synonymous with AFEU in the Security Engine 2.x chapter of the PowerQUICC device reference manuals and other related documentation.
APAD	Autopad—The MDHA will automatically pad incomplete message blocks out to 512 bits when APAD is enabled.
ARC-4	Encryption algorithm compatible with the RC-4 algorithm developed by RSA, Inc.
Auth	Authentication
CBC	Cipher block chaining—An encryption mode commonly used with block ciphers.
CHA	Crypto hardware accelerator—This term is synonymous with 'execution unit' in the Security Engine 2.x chapter of the PowerQUICC device reference manuals and other related documentation.
CTX	Context
DESA	DES accelerator—This term is synonymous with DEU in the Security Engine 2.x chapter of the PowerQUICC device reference manuals and other related documentation.
DPD	Data packet descriptor
ECB	Electronic code book—An encryption mode less commonly used with block ciphers.
EU	Execution unit
HMAC	Hashed message authentication code
IDGS	Initialize digest
IPSec	Internet protocol security
ISR	Interrupt service routine
MD	Message digest, synonymous with hash
MDHA	Message digest hardware accelerator—This term is synonymous with MDEU in the Security Engine 2.x chapter of the PowerQUICC device reference manuals and other related documentation.
OS	Operating system
PK	Public key
PKHA	Public key hardware accelerator—This term is synonymous with PKEU in the Security Engine 2.x chapter of the PowerQUICC device reference manuals and other related documentation.
RDK	Restore decrypt key—An AESA option to re-use an existing expanded AES decryption key.
RNGA	Random number generator accelerator
SDES	Single DES
TEA	Transfer error acknowledge
TDES	Triple DES
VxWorks	Operating system provided by Wind River Systems.

2 Device Driver Components

This section is provided to help users understand the internal structure of the device driver.

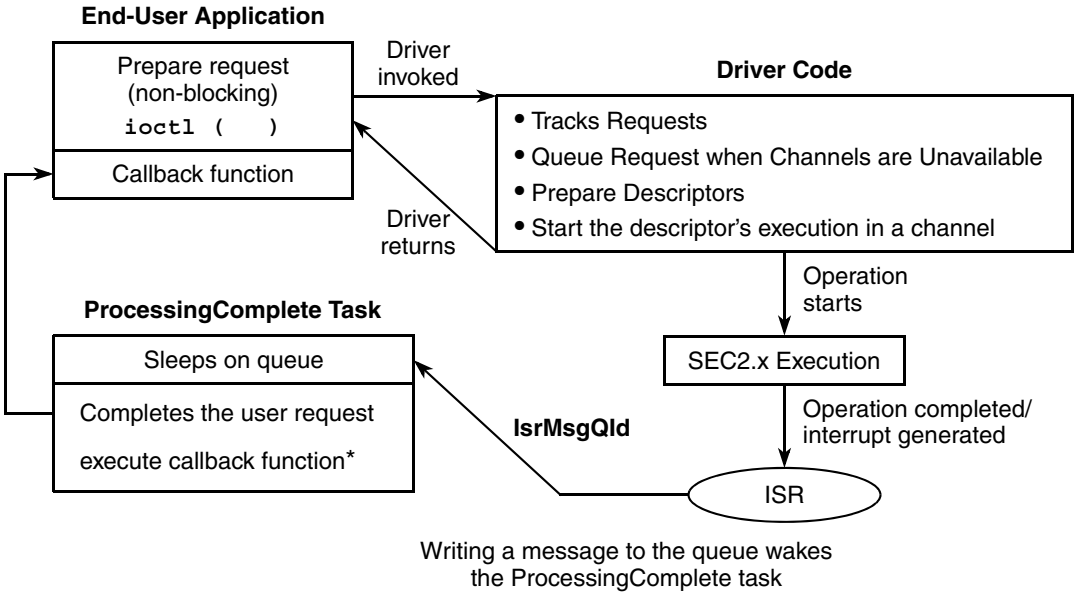
2.1 Device Driver Structure

Internally, the driver is structured in four basic components:

- Driver Initialization and Setup
- Application Request Processing
- Interrupt Service Routine
- Deferred Service Routine

While executing a request, the driver runs in system/kernel state for all components with the exception of the ISR, which runs in the operating system's standard interrupt processing context.

Figure 1 shows the SEC Reference Device Driver structure.



* If no callback function is defined, no callback takes place.

Figure 1. SEC Device Driver Structure

2.1.1 Driver Initialization Routine

The driver initialization routine includes both OS-specific and hardware-specific initialization. The steps taken by the driver initialization routine are as follows:

- Finds the security engine core and sets the device memory map starting address in `IOBaseAddress`.
- Initialize the security engine's registers
  - — Controller registers
  - — Channel registers

  - EU registers
- Initializes driver internal variables
- Initializes the channel assignment table
  - The device driver will maintain this structure with state information for each channel and user request. A mutual-exclusion semaphore protects this structure so multiple tasks are prevented from interfering with each other.
- Initializes the internal request queue
  - This queue holds requests to be dispatched when channels become available. The queue can hold up to 16 requests. The driver will reject requests with an error when the queue is full.
- `ProcessingComplete()` is enabled, which then pends on `IsrMsgQId`. This serves as the interface between the interrupt service routine and this deferred task.

2.1.2 Request Dispatch Routine

The request dispatch routine provides the `ioctl()` interface to the device driver. It uses the callers request code to identify which function is to execute and dispatches the appropriate handler to process the request. The driver performs a number of tasks that include tracking requests, queuing requests when the requested channel is unavailable, preparing data packet descriptors, and writing said descriptor's address to the appropriate channel; in effect giving the security engine the direction to begin processing the request. The `ioctl()` function returns to the end-user application without waiting for the security engine to complete, assuming that once a DPD is initiated for processing by the hardware, interrupt service may invoke a handler to provide completion notification

2.1.3 Process Request Routine

The process request routine translates the request into a sequence of one or more data packet descriptors (DPD) and feeds it to the security engine core to initiate processing. If no channels are available to handle the request, the request is queued.

2.1.4 Interrupt Service Routine

When processing is completed by the security engine, an interrupt is generated. The interrupt service routine handles the interrupt and queues the result of the operation in the `IsrMsgQId` queue for deferred processing by the `ProcessingComplete()` deferred service routine.

2.1.5 Deferred Service Routine

The `ProcessingComplete()` routine completes the request outside of the interrupt service routine, and runs in a non-ISR context. This routine depends on the `IsrMsgQId` queue and processes messages written to the queue by the interrupt service routine. This function will determine which request is complete, and notify the calling task using any handler specified by that calling task. It will then check the remaining content of the process request queue, and schedule any queued requests.

3 User Interface

3.1 Application Interface

In order to make a request of the SEC, the calling application must populate a request structure with the appropriate information to pass to the driver for processing. These structures are described in Section 4, “Individual Request Type Descriptions,” and include items such as operation ID, channel, callback routines (success and error), and data.

Once the request is prepared, the application calls `ioctl()` with the prepared request. This function is a standard system call used by operating system I/O subsystems to implement special-purpose functions. It typically follows the format:

```
int ioctl(int fd, /* file descriptor */
          int function, /* function code */
          int arg /* arbitrary argument (driver dependent) */
```

The function code (second argument) is defined as the Section 3.3.1, “I/O Control Codes.” This code will specify the driver-specific operation to be performed by the device in question. The third argument is the pointer to the SEC user request structure, which contains information needed by the driver to perform the function requested.

The following is a list of guidelines to be followed by the end-user application when preparing a request structure:

- The first member of every request structure is an operation ID (`opID`). The operation ID is used by the device driver to determine the format of the request structure.
- All requests have a "`channel`" member. It should normally be specified as zero, and as such, the driver will place the request on the “next available” channel from the pool of channels in use. If the requesting task has reserved a static channel for dedicated use (numbered 1 or higher), this member may be filled in with the number of the reserved channel.
- All process request structures have a `status` member. This value is filled in by the device driver when the interrupt for the operation occurs and it reflects the status of the completed operation as determined by the deferred service routine.
- All process request structures have two notify members, `notify` and `notify_on_error`. These notify members can be used by the device driver to notify the application when its request has been completed. They may be the same function, or different, as required by the caller's operational requirements.
- All process request structures have a `next` request member. This allows the application to chain multiple process requests together.
- It is the application's choice to use a notifier function or to poll the status member to detect completion of the request.

3.2 Error Handling

Due to the asynchronous nature of the device/driver, there are two primary sources of errors:

- Syntax or logic. These are returned in the `status` member of the 'user request' argument and as a return code from `ioctl()`. Errors of this type are detected by the driver, not by hardware.
- Protocol/procedure. These errors are returned only in the `status` member of the user request argument. Errors of this type are detected by hardware in the course of their execution.

Consequently, the end-user application needs two levels of error checking, the first one after the return from `ioctl()`, and the second after the completion of the request. The second level is possible only if the request was done with a valid`notify_on_error` handler. If the handler has not been specified, this level of error will be lost.

A code example of the two levels of errors are as follows, using an AES request as an example:

```
AESA_CRYPT_REQ aesdynReq;

  ..
  aesdynReq.opId            = DPD_AESA_CBC_ENCRYPT_CRYPT;
  aesdynReq.channel         = 0;
  aesdynReq.notify          = (void *) notifAes;
  aesdynReq.notify_on_error = (void *) notifAes;
  aesdynReq.status          = 0;
  aesdynReq.inIvBytes       = 16;
  aesdynReq.inIvData        = iv_in;
  aesdynReq.keyBytes        = 32;
  aesdynReq.keyData         = AesKey;
  aesdynReq.inBytes         = packet_length;
  aesdynReq.inData          = aesData;
  aesdynReq.outData         = aesResult;
  aesdynReq.outIvBytes      = 16;
  aesdynReq.outIvData       = iv_out;
  aesdynReq.nextReq         = 0;
  status = Ioctl(device, IOCTL_PROC_REQ, &aesdynReq);


  if (status != 0) {
    printf ("Syntax-Logic Error in dynamic descriptor 0x%x\n", status); .
    .
    .
  }.
```

```
/* in callback function notifAes */
if (aesdynReq.status != 0) {
  printf ("Error detected by HW 0x%x\n", aesdynReq.status) ;
  .
  .
}
```

3.3 Global Definitions

3.3.1 I/O Control Codes

The I/O control code is the second argument in the `ioctl` function. Definitions of these control codes are defined in `Sec2.h`.

Internally, these values are used in conjunction with a base index to create the I/O control codes. The macro for this base index is defined by `SEC2_IOCTL_INDEX` and has a default value of 0x0800. See Table 3.

Table 3. Second and Third Arguments to `ioctl()`

I/O Control Code (Second Argument in `ioctl` Function)	Purpose of `ioctl` Function
`IOCTL_PROC_REQ`	Primary form of making a request of the security engine. Passes a pointer to a detailed request block, specific to the type of request being made.
`IOCTL_GET_STATUS`	Get detailed internal status after execution. Passes pointer to a `STATUS_REQ`
`IOCTL_RESERVE_CHANNEL_STATIC`	Reserve a channel for exclusive use by a task, such that it will not be shared with other requestors. Is passed a pointer to a “channel” argument (see the “channel” element in `GENERIC_REQ`) that will be updated with a channel number for use in subsequent requests to that channel.
`IOCTL_RELEASE_CHANNEL`	Release a channel from exclusive use by a task. Is passed a pointer to a channel same as for `IOCTL_RESERVE_CHANNEL_STATIC`
`IOCTL_MALLOC`	Allocate a contiguous block of kernel memory for use in the processing of a request. Parameter is a pointer to the allocated block. Note that this is only valid on systems that support privileged memory access,
`IOCTL_FREE`	Free a block of memory originally allocated by `IOCTL_MALLOC`. Parameter is a pointer to the block to free.
`IOCTL_COPYFROM`	Pointer to type `MALLOC_REQ`, which will hold information about a user buffer that will be copied from user memory space to kernel memory space allocated by `IOCTL_MALLOC`.
`IOCTL_COPYTO`	Pointer to type `MALLOC_REQ`, which will hold information about a user buffer that will be copied from kernel memory space allocated by `IOCTL_MALLOC` back to a user's buffer.
`IOCTL_INSTALL_AUX_HANDLER`	Install an auxiliary deferred service handler for a channel that was reserved by `IOCTL_RESERVE_CHANNEL_STATIC`. This exists so that the application using a reserved channel may implement it’s own channel completion “interrupt” handler to be invoked at the completion of operations on a channel. Will be passed an argument to an `AUX_HANDLER_SPEC` which will include the channel number, and a pointer to the handler to be installed (or `NULL` if the handler is to be disabled).

3.3.2 Channel Definitions

The NUM_CHANNELS definition is used to specify the number of channels implemented in the SEC2 device. If not specified it will be set to a value of 4 as a default. See Table 4.

Table 4. Channel Defines

Define	Description
NUM_AFHAS	Number of ARC4 accelerators
NUM_DESAS	Number of DES accelerators
NUM_MDHAS	Number of message digest accelerators
NUM_RNGAS	Number of random number generators
NUM_PKHAS	Number of public key accelerators
NUM_AESAS	Number of AES accelerators
NUM_KEAS	Number of Kasumi accelerators

The NUM_CHAS definition contains the total number of crypto hardware accelerators (CHAs) in SEC2 and is simply defined as the sum of the individual channels.

The device name is defined by default as /dev/sec2.

3.3.3 Operation ID (opId) Masks

Operation Ids can be broken down into two parts, the group or type of request and the request index or descriptor within a group or type. This is provided to help understand the structuring of the opIds. It is not specifically needed within a user application. See Table 5.

Table 5. Request Operation ID Mask

Define	Description	Value
DESC_TYPE_MASK	The mask for the group or type of an opId	0xFF00
DESC_NUM_MASK	The mask for the request index or descriptor within that group or type	0x00FF

3.3.4 Return Codes

A complete list of the error status results that may be returned to the callback routines is shown in Table 6.

Table 6. Callback Error Status Return Code

Code	Description	Value
SEC2_SUCCESS	Successful completion of request	0
SEC2_MEMORY_ALLOCATION	Driver cannot obtain memory from the host operating system.	0xE004FFFF
SEC2_INVALID_CHANNEL	Channel specification was out of range. This exists for legacy compatibility and has no relevance for SEC2.	0xE004FFFE

Table 6. Callback Error Status Return Code (continued)

SEC2_INVALID_CHA_TYPE	Requested CHA does not exist in this version of the hardware.	0xE004FFFD
SEC2_INVALID_OPERATION_ID	Requested opID is out of range for this request type.	0xE004FFFC
SEC2_CHANNEL_NOT_AVAILABLE	Requested channel was not available. This error exists for legacy compatibility reasons, and has no relevance for SEC2.	0xE004FFFB
SEC2_CHA_NOT_AVAILABLE	Requested CHA was not available at the time the request was being processed.	0xE004FFFA
SEC2_INVALID_LENGTH	Length of requested data item is incompatible with request type, or data alignment incompatible.	0xE004FFF9
SEC2_OUTPUT_BUFFER_ALIGNMENT	Output buffer alignment incompatible with request type	0xE004FFF8
SEC2_ADDRESS_PROBLEM	Driver could not translate argued address into a physical address.	0xE004FFF6
SEC2_INSUFFICIENT_REQS	Request entry pool exhausted at the time of request processing; try again later.	0xE004FFF5
SEC2_CHA_ERROR	CHA flagged an error during processing; check the error notification context if one was provided to the request.	0xE004FFF2
SEC2_NULL_REQUEST	Request pointer was argued NULL.	0xE004FFF1
SEC2_REQUEST_TIMED_OUT	Timeout in request processing	0xE004FFF0
SEC2_MALLOC_FAILED	Direct kernel memory buffer request failed.	0xE004FFEF
SEC2_FREE_FAILED	Direct kernel memory free failed.	0xE004FFEE
SEC2_PARITY_SYSTEM_ERROR	Parity Error detected on the bus	0xE004FFED
SEC2_INCOMPLETE_POINTER	Error due to partial pointer	0xE004FFEC
SEC2_TEA_ERROR	A transfer error has occurred.	0xE004FFEB
SEC2_FRAGMENT_POOL_EXHAUSTED	The internal scatter-gather buffer descriptor pool is full.	0xE004FFEA
SEC2_FETCH_FIFO_OVERFLOW	Too many DPDs written to a channel (indicates an internal driver problem)	0xE004FFE9
SEC2_BUS_MASTER_ERROR	Processor could not acquire the bus for a data transfer.	0xE004FFE8
SEC2_SCATTER_LIST_ERROR	Caller's list describing a scatter-gather buffer is corrupt.	0xE004FFE7
SEC2_UNKNOWN_ERROR	Any other unrecognized error	0xE004FFE6
SEC2_IO_CARD_NOT_FOUND	Error due to device hardware not being found	-1000
SEC2_IO_MEMORY_ALLOCATE_ERROR	Error due to insufficient resources	-1001
SEC2_IO_IO_ERROR	Error due to I/O configuration	-1002

Table 6. Callback Error Status Return Code (continued)

SEC2_IO_VXWORKS_DRIVER_TABLE_ADD_ERROR	Error due to VxWorks not being able to add driver to table.	-1003
SEC2_IO_INTERRUPT_ALLOCATE_ERROR	Error due to interrupt allocation error.	-1004
SEC2_VXWORKS_CANNOT_CREATE_QUEUE	Error due to VxWorks not being able to create the ISR queue in IOInitQs()	-1009
SEC2_CANCELLED_REQUEST	Error due to canceled request	-1010
SEC2_INVALID_ADDRESS	Error due to a NULL request	-1011

3.3.5 Miscellaneous Request Structures

STATUS_REQ Structure

Used to indicate the internal state of the SEC2 core as well as the driver after the occurrence of an event. Returned as a pointer by GetStatus() and embedded in all requests. This structure is defined in Sec2Notify.h

Each element is a copy of the contents of the same register in the SEC2 driver. This structure is also known as SEC2_STATUS through a typedef.

```
unsigned long ChaAssignmentStatusRegister[2];
unsigned long InterruptControlRegister[2];
unsigned long InterruptStatusRegister[2];
unsigned long IdRegister;
unsigned long ChannelStatusRegister[NUM_CHANNELS][2];
unsigned long ChannelConfigurationRegister[NUM_CHANNELS][2];
unsigned long CHAInterruptStatusRegister[NUM_CHAS][2];
unsigned long QueueEntryDepth;
unsigned long FreeChannels;
unsigned long FreeAfhas;
unsigned long FreeDesas;
unsigned long FreeMdhas;
unsigned long FreePkhas;
unsigned long FreeAesas;
unsigned long FreeKeas;
unsigned long BlockSize;
```

SEC2_NOTIFY_ON_ERROR_CTX Structure

Structure returned to the notify_on_error callback routine that was setup in the initial process request. This structure contains the original request structure as well as an error and driver status.

```
unsigned long errorcode;     // Error that the request generated
void         *request;       // Pointer to original request
```

```
  STATUS_REQ    driverstatus;  // Detailed information as to the state of the
                               // hardware and the driver at the time of an error
```

3.3.6 Process Request Structures

All process request structures contain a copy of identical request header information, which is defined by the COMMON_REQ_PREAMBLE macro. The members of this header must be filled in as needed by the user prior to the issue of the user's request. Descriptions of the process-request structures are shown in Table 7.

```
    unsigned long                opId;
    unsigned char                scatterBufs;
    unsigned char                notifyFlags;
    unsigned char                reserved;
    unsigned char                channel;
    PSEC2_NOTIFY_ROUTINE         notify;
    PSEC2_NOTIFY_CTX             pNotifyCtx;
    PSEC2_NOTIFY_ON_ERROR_ROUTINE notify_on_error;
    SEC2_NOTIFY_ON_ERROR_CTX     ctxNotifyOnErr;
    int                          status;
    void                        *nextReq;
```

Table 7. Process Request Structures

Process-Request Structure	Description
opId	operation Id which identifies what type of request this is. It is normally associated with a specific type of cryptographic operation, see Individual Requests for all supported request types.
scatterBufs	A bitmask that specifies which of the argued buffers are mapped through a scatter-gather list. The mask is filled out via the driver's helper function MarkScatterBuffer(), described in the discussion on scatter-gather buffer management.
notifyFlags	If a POSIX-style signal handler will be responsible for request completion notification, then it can contain ORed bits of NOTIFY_IS_PID and/or NOTIFY_ERROR_IS_PID, signifying that the notify or notify_on_error pointers are instead the process ID's (i.e. getpid()) of the task requesting a signal upon request completion.
channel	identifies the channel to be used for the request. Should normally be zero, and as such, the driver places the request on the “next available” channel, or queues the request. If a static channel has been reserved, pass it’s number here to ensure that the request executes on this channel only.
notify	pointer to a notification callback routine that will be called when the request has completed successfully. May instead be a process ID if a user-state signal handler will flag completion, see notifyFlags for more info.
pNotifyCtx	pointer to context area to be passed back through the notification routine.
notify_on_error	pointer to the notify on error routine that will be called when the request has completed unsuccessfully. May instead be a process ID if a user-state signal handler will flag completion, see notifyFlags for more info.

Table 7. Process Request Structures (continued)

ctxNotifyOnErr	context area that is filled in by the driver when there is an error.
status	will contain the returned status of request.
nextReq	pointer to next request which allows for multiple request to be linked together and sent via a single ioctl function call.

The additional data in the process request structures is specific to each request; refer to the specific structure for this information.

3.3.7 Scatter-Gather Buffer Management

A unique feature of the SEC 2.x core is the presence of the hardware's ability to read and act on a scatter-gather description list for a data buffer. This allows the hardware to more efficiently deal with buffers located in memory belonging to a non-privileged process; memory which may not be contiguous, but instead may be at scattered locations determined by the memory management scheme of the host system. Any data buffer in any request may be "marked" as a scattered memory buffer by the requestor as needed.

For the requestor to do so, two actions must be taken:

- A linked list of structures of type EXT_SCATTER_ELEMENT, one per memory fragment, must be constructed to describe the whole of the buffer's content.
- The buffer pointer shall reference the head of this list, not the data itself. The buffers containing scatter references shall be marked in the request's scatterBufs element. Which bits get marked shall be determined by a helper function that understands the mapping used on an individual request basis.

3.3.7.1 Building the Local Scatter/gather List with EXT_SCATTER_ELEMENT

Since individual operating systems shall have their own internal means defining memory mapping constructs, the driver cannot be designed with specific knowledge of one particular mapping method. Therefore, a generic memory fragment definition structure, EXT_SCATTER_ELEMENT is defined for this purpose.

Each EXT_SCATTER_ELEMENT describes one contiguous fragment of user memory, and is designed so that multiple fragments can be tied together into a single linked list. Each element has the aspects shown in Table 8:

Table 8. Scatter_Gather_Elements

Aspects of Scatter_Gather_Elements	Description
void *next;	Pointer to next fragment in list, NULL if at end of list
void *fragment;	Pointer to contiguous data fragment
unsigned short size;	Size of this fragment in bytes

With this, the caller must construct the list of all the fragments needed to describe the buffer, NULL terminate the end of the list, and pass the head as the buffer pointer argument. This list must remain intact until completion of the request.

3.3.7.2 Scatter Buffer Marking

For reasons of legacy compatibility, the structure of all driver request types maintains the same size and form as prior versions, with a minor change in that a size-compatible scatterBufs element was added as a modification to the channel element in other versions. This allows the caller a means of indicating which buffers in the request are scatter-composed, as opposed to direct, contiguous memory (for instance, key data could be in contiguous system memory, while ciphertext data will be in fragmented user memory).

A problem with marking buffers using this method is that there is no means for the caller to clearly identify which bit in scatterBufs matches any given pointer in the request, since the data description portion of different requests cannot be consistent or of any particular order.

A helper function, MarkScatterBuffer(), is therefore made available by the driver to make the bit/pointer association logic in the driver accessible to the caller. It's form is:

```
MarkScatterBuffer(void *request, void *buffer);
```

where request points to the request block being built (the opId element must be set prior to call), and buffer points to the element within the request which references a scattered buffer. It will then mark the necessary bit in scatterBufs that defines this buffer for this specific request type.

3.3.7.3 Direct Scatter-Gather Usage Example

In order to make this usage clear, an example is presented. Assume that a triple DES encryption operation is to be constructed, where the input and output buffers are located in fragmented user memory, and the cipher keys and IV are contained in system memory. A DES_LOADCTX_CRYPT_REQ is zero-allocated as encReq, and constructed:

```
/* set up encryption operation */
encReq.opId            = DPD_TDES_CBC_CTX_ENCRYPT;
encReq.notify          = notifier;
encReq.notify_on_error = notifier;
encReq.inIvBytes       = 8;
encReq.keyBytes        = 24;
encReq.inBytes         = bufsize;
encReq.inIvData        = iv;
encReq.keyData         = cipherKey;
encReq.inData          = (unsigned char *)input; /* this buffer is scattered */
encReq.outIvBytes      = 8;
encReq.outIvData       = ctx;
encReq.outData         = (unsigned char *)output; /* this buffer is scattered */

MarkScatterBuffer(&encReq, &encReq.input);
MarkScatterBuffer(&encReq, &encReq.output);
```

Upon completion of the two mark calls, encReq.scatterBufs will have two bits set within it that the driver knows how to interpret as meaning that the intended buffers have scatter lists defined for them, and will process them accordingly as the DPD is built for the hardware.

3.3.8 Reserved Channels

Beginning with driver version 1.5, a capability is added to allow a task the ability to reserve a channel and use it separately from other requests (i.e. keeps that channel out of the “pool” of channels available for general processing). This feature may be desirable in the instance that a channel may be needed for rapid servicing of data peculiar to a specific protocol with high real-time demand requirements.

Requesting a channel is done with the IOCTL_RESERVE_CHANNEL_STATIC call. It’s argument is a pointer to a channel byte that will be used in the channel argument for all subsequent requests. Once the static channel is no longer needed, it can be returned to the free channel pool with IOCTL_RELEASE_CHANNEL.

3.3.8.1 Reserved Channel Specification

Normal requests to the driver take place through the COMMON_REQ_PREAMBLE to the detailed request block. The channel argument in this block is normally left at zero and if specified as such, the driver “allocates” a channel on a round-robin basis for each request as it comes in (extra requests become placed in a queue for each channel).

If the channel specification is nonzero, (for example, from 1 to 4 for all SEC 2.x devices other that are not 2.2) then the driver will post the request to that channel if the channel is reserved to that requestor.

3.3.8.2 Auxiliary Channel Service Handlers

Should a reserved channel need the ability to invoke a special handler at the end of processing for a given request, then an auxiliary handler routine may be installed to service end-of-processing events for that channel. That handler may be installed or disabled through the use of IOCTL_INSTALL_AUX_HANDLER, which is passed a pointer to a AUX_HANDLER_SPEC which specifies the channel to service, and a pointer to the user’s handler to execute the service.

```
typedef struct {
    int   (*auxHandler)(int ch, void *regs);
    char   channel;
} AUX_HANDLER_SPEC;
```

In any case, channel should always be the channel that is reserved for use. If it is not reserved, an error will be returned.

The auxHandler pointer references the user handler function. When dispatched by the driver, it will be passed two items, (1) the number of the channel whose interrupt triggered this handler, and (2) a pointer to a copy of a DEVICE_REGS as saved from the ISR as it’s argument, so that the handler can act on any error information that was saved. Although the auxiliary handler is cast to return an int as a status value, this return is ignored by the driver at this time.

If auxHandler is passed as a NULL value, then an existing handler will be disabled. This should be accomplished before a channel is released.

NOTE

Auxiliary handlers run in a deferred-service context, not in the context of the calling application. For this reason, they need to complete in a reasonable amount of time to prevent starving other channels from service. Also, because they are not in the user's context, auxiliary handlers should never be used from a Linux user-state application, since the user-state handler cannot be directly invoked.

4 Individual Request Type Descriptions

4.1 Random Number Requests

4.1.1 RNG_REQ

```
COMMON_REQ_PREAMBLE
unsigned long rngBytes;
unsigned char* rngData;
```

NUM_RNGA_DESC defines the number of descriptors within the DPD_RNG_GROUP that use this request.

DPD_RNG_GROUP (0x1000) defines the group for all descriptors within this request. See Table 9.

Table 9. RNG_REQ Valid Descriptor (opId)

Descriptor	Value	Function Description
DPD_RNG_GETRN	0x1000	Generate a series of random values

4.2 DES Requests

4.2.1 DES_CBC_CRYPT_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  inIvBytes;  /* 0 or 8 bytes */
unsigned char *inIvData;
unsigned long  keyBytes;   /* 8, 16, or 24 bytes */
unsigned char *keyData;
unsigned long  inBytes;    /* multiple of 8 bytes */
unsigned char *inData;
unsigned char *outData;    /* output length = input length */
unsigned long  outIvBytes; /* 0 or 8 bytes */
unsigned char *outIvData;
```

NUM_DES_LOADCTX_DESC defines the number of descriptors within the DPD_DES_CBC_CTX_GROUP that use this request.

DPD_DES_CBC_CTX_GROUP (0x2500) defines the group for all descriptors within this request. See Table 10.

Table 10. DES_CBC_CRYPT_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_SDES_CBC_CTX_ENCRYPT	0x2500	Load encrypted context from a dynamic channel to encrypt in single DES using CBC mode
DPD_SDES_CBC_CTX_DECRYPT	0x2501	Load encrypted context from a dynamic channel to decrypt in single DES using CBC mode
DPD_TDES_CBC_CTX_ENCRYPT	0x2502	Load encrypted context from a dynamic channel to encrypt in triple DES using CBC mode
DPD_TDES_CBC_CTX_DECRYPT	0x2503	Load encrypted context from a dynamic channel to decrypt in triple DES using CBC mode

4.2.2 DES_CRYPT_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  keyBytes; /* 8, 16, or 24 bytes */
unsigned char *keyData;
unsigned long  inBytes;  /* multiple of 8 bytes */
unsigned char *inData;
unsigned char *outData;  /* output length = input length */
```

NUM_DES_DESC defines the number of descriptors within the DPD_DES_ECB_GROUP that use this request.

DPD_DES_ECB_GROUP (0x2600) defines the group for all descriptors within this request. See Table 11.

Table 11. DES_CRYPT_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_SDES_ECB_ENCRYPT	0x2600	Encrypt data in single DES using ECB mode
DPD_SDES_ECB_DECRYPT	0x2601	Decrypt data in single DES using ECB mode
DPD_TDES_ECB_ENCRYPT	0x2602	Encrypt data in triple DES using ECB mode
DPD_TDES_ECB_DECRYPT	0x2603	Decrypt data in triple DES using ECB mode

4.3 ARC4 Requests

4.3.1 ARC4_LOADCTX_CRYPT_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  inCtxBytes;  /* 257 bytes */
unsigned char *inCtxData;
unsigned long  inBytes;
```

```
unsigned char *inData;
unsigned char *outData;     /* output length = input length */
unsigned long  outCtxBytes; /* 257 bytes */
unsigned char *outCtxData;
```

NUM_RC4_LOADCTX_UNLOADCTX_DESC defines the number of descriptors within the DPD_RC4_LDCTX_CRYPT_ULCTX_GROUP that use this request.

DPD_RC4_LDCTX_CRYPT_ULCTX_GROUP (0x3400) defines the group for all descriptors within this request. See Table 12.

Table 12. ARC4_LOADCTX_CRYPT_REQ Valid Descriptor (opId)

Descriptor	Value	Function Description
DPD_RC4_LDCTX_CRYPT_ULCTX	0x3400	Load context, encrypt using ARC-4, and store the resulting context.

4.3.2 ARC4_LOADKEY_CRYPT_UNLOADCTX_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  keyBytes;
unsigned char *keyData;
unsigned long  inBytes;
unsigned char *inData;
unsigned char *outData;     /* output length = input length */
unsigned long  outCtxBytes; /* 257 bytes */
unsigned char* outCtxData;
```

NUM_RC4_LOADKEY_UNLOADCTX_DESC defines the number of descriptors within the DPD_RC4_LDKEY_CRYPT_ULCTX_GROUP that use this request.

DPD_RC4_LDKEY_CRYPT_ULCTX_GROUP (0x3500) defines the group for all descriptors within this request. See Table 13.

Table 13. ARC4_LOADKEY_CRYPT_UNLOADCTX_REQ Valid Descriptor (opId)

Descriptor	Value	Function Description
DPD_RC4_LDKEY_CRYPT_ULCTX	0x3500	Load the cipher key, encrypt using ARC-4, then save the resulting context.

4.4 Hash Requests

4.4.1 HASH_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  ctxBytes;
unsigned char *ctxData;
unsigned long  inBytes;
unsigned char *inData;
unsigned long  outBytes; /* length is fixed by algorithm */
```

```
unsigned char *outData;
unsigned char *cmpData; /* digest auto-compare value */
```

NUM_MDHA_DESC defines the number of descriptors within the DPD_HASH_LDCTX_HASH_ULCTX_GROUP that use this request.

DPD_HASH_LDCTX_HASH_ULCTX_GROUP (0x4400) defines the group for all descriptors within this request. See Table 14.

Table 14. HASH_REQ Valid Descriptors (0x4400) (opId)

Descriptors	Value	Function Description
DPD_SHA256_LDCTX_HASH_ULCTX	0x4400	Compute a final SHA256 digest using autopadding and an external context.
DPD_MD5_LDCTX_HASH_ULCTX	0x4401	Compute a final MD5 digest using autopadding and an external context.
DPD_SHA_LDCTX_HASH_ULCTX	0x4402	Compute a final SHA1 digest using autopadding and an external context.
DPD_SHA256_LDCTX_IDGS_HASH_ULCTX	0x4403	Compute an interim SHA256 digest using autopadding, and use the algorithm's standard initialization.
DPD_MD5_LDCTX_IDGS_HASH_ULCTX	0x4404	Compute an interim MD5 digest using autopadding, and use the algorithm's standard initialization.
DPD_SHA_LDCTX_IDGS_HASH_ULCTX	0x4405	Compute an interim SHA1 digest using autopadding, and use the algorithm's standard initialization.
DPD_SHA256_CONT_HASH_ULCTX	0x4406	Compute an interim SHA256 digest using autopadding, and use an external context.
DPD_MD5_CONT_HASH_ULCTX	0x4407	Compute an interim MD5 digest using autopadding, and use an external context.
DPD_SHA_CONT_HASH_ULCTX	0x4408	Compute an interim SHA1 digest using autopadding, and use an external context.
DPD_SHA224_LDCTX_HASH_ULCTX	0x4409	Compute a final SHA224 digest with autopadding and an external context (Does not apply to SEC 2.0).
DPD_SHA224_LDCTX_IDGS_HASH_ULCTX	0x440a	Compute an interim SHA224 digest using autopadding, and use the algorithm's standard initialization (Does not apply to SEC 2.0).
DPD_SHA224_CONT_HASH_ULCTX	0x440b	Compute an interim SHA224 digest using autopadding, and use an external context (Does not apply to SEC 2.0).
DPD_SHA256_LDCTX_HASH_ULCTX_CMP	0x440c	Compute a final SHA256 digest using autopadding and an external context. Compare the digest to an expected value, and flag an error if they do not compare (Does not apply to SEC 2.0).
DPD_MD5_LDCTX_HASH_ULCTX_CMP	0x440d	Compute a final MD5 digest using autopadding and an external context. Compare the digest to an expected value, and flag an error if they do not compare (Does not apply to SEC 2.0).
DPD_SHA_LDCTX_HASH_ULCTX_CMP	0x440e	Compute a final SHA1 digest using autopadding and an external context. Compare the digest to an expected value, and flag an error if they do not compare (Does not apply to SEC 2.0).

Table 14. HASH_REQ Valid Descriptors (0x4400) (opId) (continued)

DPD_SHA256_LDCTX_IDGS_HASH_ULCTX_CMP	0x440f	Compute an interim SHA256 digest using autopadding, and use the algorithm's standard initialization. Compare the digest to an expected value, and flag an error if they do not compare (Does not apply to SEC 2.0).
DPD_MD5_LDCTX_IDGS_HASH_ULCTX_CMP	0x4410	Compute an interim MD5 digest using autopadding, and use the algorithm's standard initialization. Compare the digest to an expected value, and flag an error if they do not compare (Does not apply to SEC 2.0)
DPD_SHA_LDCTX_IDGS_HASH_ULCTX_CMP	0x4411	Compute an interim SHA1 digest using autopadding, and use the algorithm's standard initialization. Compare the digest to an expected value, and flag an error if they do not compare (Does not apply to SEC 2.0)
DPD_SHA224_LDCTX_HASH_ULCTX_CMP	0x4412	Compute a final SHA224 digest using autopadding and an external context. Compare the digest to an expected value, and flag an error if they do not compare (Does not apply to SEC 2.0)
DPD_SHA224_LDCTX_IDGS_HASH_ULCTX_CMP	0x4413	Compute an interim SHA224 digest using autopadding, and use the algorithm's standard initialization. Compare the digest to an expected value, and flag an error if they do not compare (Does not apply to SEC 2.0)

NUM_MDHA_PAD_DESC defines the number of descriptors within the DPD_HASH_LDCTX_HASH_PAD_ULCTX_GROUP that use this request.

DPD_HASH_LDCTX_HASH_PAD_ULCTX_GROUP (0x4500) defines the group for all descriptors within this request. See Table 15.

Table 15. HASH_REQ Valid Descriptors (0x4500) (opId)

Descriptors	Value	Function Description
DPD_SHA256_LDCTX_HASH_PAD_ULCTX	0x4500	Compute digest over pre-padded data using a SHA-256 hash algorithm with an external context.
DPD_MD5_LDCTX_HASH_PAD_ULCTX	0x4501	Compute digest over pre-padded data using an MD5 hash algorithm with an external context.
DPD_SHA_LDCTX_HASH_PAD_ULCTX	0x4502	Compute digest over pre-padded data using a SHA-1 hash algorithm with an external context.
DPD_SHA256_LDCTX_IDGS_HASH_PAD_ULCTX	0x4503	Compute digest over pre-padded data using SHA-256 with its standard initialization.
DPD_MD5_LDCTX_IDGS_HASH_PAD_ULCTX	0x4504	Compute digest over pre-padded data using MD5 with its standard initialization.
DPD_SHA_LDCTX_IDGS_HASH_PAD_ULCTX	0x4505	Compute digest over pre-padded data using SHA1 with its standard initialization.
DPD_SHA224_LDCTX_HASH_PAD_ULCTX	0x4506	Compute digest over pre-padded data using a SHA224 hash algorithm with an external context (Does not apply to SEC 2.0).
DPD_SHA224_LDCTX_IDGS_HASH_PAD_ULCTX	0x4507	Compute digest over pre-padded data using SHA224 with its standard initialization (Does not apply to SEC 2.0).

Table 15. HASH_REQ Valid Descriptors (0x4500) (opId) (continued)

DPD_SHA256_LDCTX_HASH_PAD_ULCTX_CMP	0x4508	Compute digest over pre-padded data using a SHA-256 hash algorithm with an external context. Compare resulting digest to external value and return an error if they do not match (Does not apply to SEC 2.0).
DPD_MD5_LDCTX_HASH_PAD_ULCTX_CMP	0x4509	Compute digest over pre-padded data using an MD5 hash algorithm with an external context. Compare resulting digest to external value and return an error if they do not match (Does not apply to SEC 2.0)
DPD_SHA_LDCTX_HASH_PAD_ULCTX_CMP	0x450a	Compute digest over pre-padded data using a SHA-1 hash algorithm with an external context. Compare resulting digest to external value and return an error if they do not match (Does not apply to SEC 2.0)
DPD_SHA256_LDCTX_IDGS_HASH_PAD_ULCTX_CMP	0x450b	Compute digest over pre-padded data using SHA-256 with its standard initialization. Compare resulting digest to external value and return an error if they do not match (Does not apply to SEC 2.0)
DPD_MD5_LDCTX_IDGS_HASH_PAD_ULCTX_CMP	0x450c	Compute digest over pre-padded data using MD5 with its standard initialization. Compare resulting digest to external value and return an error if they do not match (Does not apply to SEC 2.0)
DPD_SHA_LDCTX_IDGS_HASH_PAD_ULCTX_CMP	0x450d	Compute digest over pre-padded data using SHA1 with its standard initialization. Compare resulting digest to external value and return an error if they do not match (Does not apply to SEC 2.0)
DPD_SHA224_LDCTX_HASH_PAD_ULCTX_CMP	0x450e	Compute digest over pre-padded data using a SHA224 hash algorithm using an external context. Compare resulting digest to external value and return an error if they do not match (Does not apply to SEC 2.0)
DPD_SHA224_LDCTX_IDGS_HASH_PAD_ULCTX_CMP	0x450f	Compute digest over pre-padded data using SHA224 with its standard initialization. Compare resulting digest to external value and return an error if they do not match (Does not apply to SEC 2.0)

4.5 HMAC Requests

4.5.1 HMAC_PAD_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  keyBytes;
unsigned char *keyData;
unsigned long  inBytes;
unsigned char *inData;
unsigned long  outBytes; /* length is fixed by algorithm */
unsigned char *outData;
unsigned char *cmpData; /* digest auto-compare value */
```

NUM_HMAC_PAD_DESC defines the number of descriptors within the DPD_HASH_LDCTX_HMAC_ULCTX_GROUP that use this request.

DPD_HASH_LDCTX_HMAC_ULCTX_GROUP (0x4A00) defines the group for all descriptors within this request. See Table 16.

Table 16. HMAC_PAD_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_SHA256_LDCTX_HMAC_ULCTX	0x4A00	Compute a SHA256 HMAC digest over a non-padded message
DPD_MD5_LDCTX_HMAC_ULCTX	0x4A01	Compute an MD5 HMAC digest over a non-padded message
DPD_SHA_LDCTX_HMAC_ULCTX	0x4A02	Compute a SHA1 HMAC digest over a non-padded message
DPD_SHA256_LDCTX_HMAC_PAD_ULCTX	0x4A03	Compute a SHA256 HMAC digest over a pre-padded message
DPD_MD5_LDCTX_HMAC_PAD_ULCTX	0x4A04	Compute an MD5 HMAC digest over a pre-padded message
DPD_SHA_LDCTX_HMAC_PAD_ULCTX	0x4A05	Compute a SHA1 HMAC digest over a pre-padded message
DPD_SHA224_LDCTX_HMAC_ULCTX	0x40a6	Compute a SHA224 HMAC digest over a non-padded message (Does not apply to SEC 2.0)
DPD_SHA224_LDCTX_HMAC_PAD_ULCTX	0x4a07	Compute a SHA224 HMAC digest over a pre-padded message (Does not apply to SEC 2.0)
DPD_SHA256_LDCTX_HMAC_ULCTX_CMP	0x4a08	Compute a SHA256 HMAC digest over a non-padded message. Compare the resulting digest to an expected value, and return an error if they do not compare (Does not apply to SEC 2.0)
DPD_MD5_LDCTX_HMAC_ULCTX_CMP	0x4a09	Compute an MD5 HMAC digest over a non-padded message. Compare the resulting digest to an expected value, and return an error if they do not compare (Does not apply to SEC 2.0)
DPD_SHA_LDCTX_HMAC_ULCTX_CMP	0x4a0a	Compute a SHA1 HMAC digest over a non-padded message. Compare the resulting digest to an expected value, and return an error if they do not compare (Does not apply to SEC 2.0)
DPD_SHA256_LDCTX_HMAC_PAD_ULCTX_CMP	0x4a0b	Compute a SHA256 HMAC digest over a pre-padded message. Compare the resulting digest to an expected value, and return an error if they do not compare (Does not apply to SEC 2.0)
DPD_MD5_LDCTX_HMAC_PAD_ULCTX_CMP	0x4a0c	Compute an MD5 HMAC digest over a pre-padded message. Compare the resulting digest to an expected value, and return an error if they do not compare (Does not apply to SEC 2.0)
DPD_SHA_LDCTX_HMAC_PAD_ULCTX_CMP	0x4a0d	Compute a SHA1 HMAC digest over a pre-padded message. Compare the resulting digest to an expected value, and return an error if they do not compare (Does not apply to SEC 2.0)
DPD_SHA224_LDCTX_HMAC_ULCTX_CMP	0x40ae	Compute a SHA224 HMAC digest over a non-padded message. Compare the resulting digest to an expected value, and return an error if they do not compare (Does not apply to SEC 2.0)
DPD_SHA224_LDCTX_HMAC_PAD_ULCTX_CMP	0x4a0f	Compute a SHA224 HMAC digest over a pre-padded message. Compare the resulting digest to an expected value, and return an error if they do not compare (Does not apply to SEC 2.0)

4.6 AES Requests

4.6.1 AESA_CRYPT_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  keyBytes;    /* 16, 24, or 32 bytes */
unsigned char *keyData;
unsigned long  inIvBytes;   /* 0 or 16 bytes */
unsigned char *inIvData;
unsigned long  inBytes;     /* multiple of 8 bytes */
unsigned char *inData;
unsigned char *outData;     /* output length = input length */
unsigned long  outCtxBytes; /* 0 or 8 bytes */
unsigned char *outCtxData;
```

NUM_AESA_CRYPT_DESC defines the number of descriptors within the DPD_AESA_CRYPT_GROUP that use this request.

DPD_AESA_CRYPT_GROUP (0x6000) defines the group for all descriptors within this request. See Table 17.

Table 17. AESA_CRYPT_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_AESA_CBC_ENCRYPT_CRYPT	0x6000	Perform encryption in AESA using CBC mode
DPD_AESA_CBC_DECRYPT_CRYPT	0x6001	Perform decryption in AESA using CBC mode
DPD_AESA_CBC_DECRYPT_CRYPT_RDK	0x6002	Perform decryption in AESA using CBC mode with RDK
DPD_AESA_ECB_ENCRYPT_CRYPT	0x6003	Perform encryption in AESA using ECB mode
DPD_AESA_ECB_DECRYPT_CRYPT	0x6004	Perform decryption in AESA using ECB mode
DPD_AESA_ECB_DECRYPT_CRYPT_RDK	0x6005	Perform decryption in AESA using ECB mode with RDK
DPD_AESA_CTR_CRYPT	0x6006	Perform CTR in AESA
DPD_AESA_CTR_HMAC	0x6007	Perform AES CTR-mode cipher operation with integrated authentication as part of the operation

4.7 Integer Public Key Requests

4.7.1 MOD_EXP_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  aDataBytes;
unsigned char *aData;
unsigned long  expBytes;
unsigned char *expData;
unsigned long  modBytes;
unsigned char *modData;
```

```
unsigned long  outBytes;
unsigned char *outData;
```

NUM_MM_EXP_DESC defines the number of descriptors within the DPD_MM_LDCTX_EXP_ULCTX_GROUP that use this request.

DPD_MM_LDCTX_EXP_ULCTX_GROUP (0x5100) defines the group for all descriptors within this request. See Table 18.

Table 18. MOD_EXP_REQ Valid Descriptor (opId)

Descriptor	Value	Function Description
DPD_MM_LDCTX_EXP_ULCTX	0x5100	Perform a modular exponentiation operation

4.7.2 MOD_SS_EXP_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  expBytes;
unsigned char *expData;
unsigned long  modBytes;
unsigned char *modData;
unsigned long  aDataBytes;
unsigned char *aData;
unsigned long  bDataBytes;
unsigned char *bData;
```

NUM_MM_SS_EXP_DESC defines the number of descriptors within the DPD_MM_SS_EXP_GROUP that use this request.

DPD_MM_SS_EXP_GROUP (0x5B00) defines the group for all descriptors within this request. See Table 19.

Table 19. MOD_SS_EXP_REQ Valid Descriptor (opId)

Descriptor	Value	Function Description
DPD_MM_SS_RSA_EXP	0x5B00	Perform a single-stage RSA exponentiation operation

4.7.3 MOD_R2MODN_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  modBytes;
unsigned char *modData;
unsigned long  outBytes;
unsigned char *outData;
```

NUM_MM_R2MODN_DESC defines the number of descriptors within the DPD_MM_LDCTX_R2MODN_ULCTX_GROUP that use this request.

DPD_MM_LDCTX_R2MODN_ULCTX_GROUP (0x5200) defines the group for all descriptors within this request.See Table 20.

Table 20. MOD_R2MODN_REQ Valid Descriptor (opId)

Descriptor	Value	Function Description
DPD_MM_LDCTX_R2MODN_ULCTX	0x5200	Perform a R2MOD operation upon a public key.

4.7.4 MOD_RRMODP_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  nBytes;
unsigned long  pBytes;
unsigned char *pData;
unsigned long  outBytes;
unsigned char *outData;
```

NUM_MM_RRMODP_DESC defines the number of descriptors within the DPD_MM_LDCTX_RRMODP_ULCTX_GROUP that use this request.

DPD_MM_LDCTX_RRMODP_ULCTX_GROUP (0x5300) defines the group for all descriptors within this request. See Table 21.

Table 21. MOD_RRMODP_REQ Valid Descriptor (opId)

Descriptor	Value	Function Description
DPD_MM_LDCTX_RRMODP_ULCTX	0x5300	Compute the result of an RRMODP operation

4.7.5 MOD_INV_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  nBytes;
unsigned char *inData;
unsigned long  inBytes;
unsigned char *inData;
unsigned long  outBytes;
unsigned char *outData;
```

NUM_MM_MOD_INV_DESC defines the number of descriptors within the DPD_MM_MOD_INV_ULCTX_GROUP that use this request.

DPD_MM_MOD_INV_ULCTX_GROUP (0x5500) defines the group for all descriptors within this request. See Table 22.

Table 22. MOD_INV_REQ Valid Descriptor (opId)

Descriptor	Value	Function Description
DPD_MM_MOD_INV_ULCTX	0x5500	Compute the modular inverse of inData using the modulus specified in inData

4.7.6 MOD_2OP_REQ

```
unsigned long  bDataBytes;
unsigned char *bData;
unsigned long  aDataBytes;
unsigned char *aData;
unsigned long  modBytes;
unsigned char *modData;
unsigned long  outBytes;
unsigned char *outData;
```

NUM_MM_2OP_DESC defines the number of descriptors within the DPD_MM_LDCTX_2OP_ULCTX_GROUP that use this request.

DPD_MM_LDCTX_2OP_ULCTX_GROUP (0x5400) defines the group for all descriptors within this request. See Table 23.

Table 23. MOD_2OP_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_MM_LDCTX_MUL1_ULCTX	0x5400	Perform a modular MUL1 operation
DPD_MM_LDCTX_MUL2_ULCTX	0x5401	Perform a modular MUL2 operation
DPD_MM_LDCTX_ADD_ULCTX	0x5402	Perform a modular ADD operation
DPD_MM_LDCTX_SUB_ULCTX	0x5403	Perform a modular SUB operation
DPD_POLY_LDCTX_A0_B0_MUL1_ULCTX	0x5404	Perform a modular A0-to-B0 MUL1 operation
DPD_POLY_LDCTX_A0_B0_MUL2_ULCTX	0x5405	Perform a modular A0-to-B0 MUL2 operation
DPD_POLY_LDCTX_A0_B0_ADD_ULCTX	0x5406	Perform a modular A0-to-B0 ADD operation
DPD_POLY_LDCTX_A1_B0_MUL1_ULCTX	0x5407	Perform a modular A1-to-B0 MUL1 operation
DPD_POLY_LDCTX_A1_B0_MUL2_ULCTX	0x5408	Perform a modular A1-to-B0 MUL2 operation
DPD_POLY_LDCTX_A1_B0_ADD_ULCTX	0x5409	Perform a modular A1-to-B0 ADD operation
DPD_POLY_LDCTX_A2_B0_MUL1_ULCTX	0x540A	Perform a modular A2-to-B0 MUL1 operation
DPD_POLY_LDCTX_A2_B0_MUL2_ULCTX	0x540B	Perform a modular A2-to-B0 MUL2 operation
DPD_POLY_LDCTX_A2_B0_ADD_ULCTX	0x540C	Perform a modular A2-to-B0 ADD operation
DPD_POLY_LDCTX_A3_B0_MUL1_ULCTX	0x540D	Perform a modular A3-to-B0 MUL1 operation
DPD_POLY_LDCTX_A3_B0_MUL2_ULCTX	0x540E	Perform a modular A3-to-B0 MUL2 operation
DPD_POLY_LDCTX_A3_B0_ADD_ULCTX	0x540F	Perform a modular A3-to-B0 ADD operation
DPD_POLY_LDCTX_A0_B1_MUL1_ULCTX	0x5410	Perform a modular A0-to-B1 MUL1 operation
DPD_POLY_LDCTX_A0_B1_MUL2_ULCTX	0x5411	Perform a modular A-to-B MUL2 operation
DPD_POLY_LDCTX_A0_B1_ADD_ULCTX	0x5412	Perform a modular A0-to-B1 ADD operation
DPD_POLY_LDCTX_A1_B1_MUL1_ULCTX	0x5413	Perform a modular A1-to-B1 MUL1 operation
DPD_POLY_LDCTX_A1_B1_MUL2_ULCTX	0x5414	Perform a modular A1-to-B1 MUL2 operation

Table 23. MOD_2OP_REQ Valid Descriptors (opId) (continued)

Descriptors	Value	Function Description
DPD_POLY_LDCTX_A1_B1_ADD_ULCTX	0x5415	Perform a modular A1-to-B1 ADD operation
DPD_POLY_LDCTX_A2_B1_MUL1_ULCTX	0x5416	Perform a modular A2-to-B1 MUL1 operation
DPD_POLY_LDCTX_A2_B1_MUL2_ULCTX	0x5417	Perform a modular A2-to-B1 MUL2 operation
DPD_POLY_LDCTX_A2_B1_ADD_ULCTX	0x5418	Perform a modular A2-to-B1 ADD operation
DPD_POLY_LDCTX_A3_B1_MUL1_ULCTX	0x5419	Perform a modular A3-to-B1 MUL1 operation
DPD_POLY_LDCTX_A3_B1_MUL2_ULCTX	0x541A	Perform a modular A3-to-B1 MUL2 operation
DPD_POLY_LDCTX_A3_B1_ADD_ULCTX	0x541B	Perform a modular A3-to-B1 ADD operation
DPD_POLY_LDCTX_A0_B2_MUL1_ULCTX	0x541C	Perform a modular A0-to-B2 MUL1 operation
DPD_POLY_LDCTX_A0_B2_MUL2_ULCTX	0x541D	Perform a modular A0-to-B2 MUL2 operation
DPD_POLY_LDCTX_A0_B2_ADD_ULCTX	0x541E	Perform a modular A0-to-B2ADD operation
DPD_POLY_LDCTX_A1_B2_MUL1_ULCTX	0x541F	Perform a modular A1-to-B2 MUL1 operation
DPD_POLY_LDCTX_A1_B2_MUL2_ULCTX	0x5420	Perform a modular A1-to-B2 MUL2 operation
DPD_POLY_LDCTX_A1_B2_ADD_ULCTX	0x5421	Perform a modular A1-to-B2 ADD operation
DPD_POLY_LDCTX_A2_B2_MUL1_ULCTX	0x5422	Perform a modular A2-to-B2 MUL1 operation
DPD_POLY_LDCTX_A2_B2_MUL2_ULCTX	0x5423	Perform a modular A2-to-B2 MUL2 operation
DPD_POLY_LDCTX_A2_B2_ADD_ULCTX	0x5424	Perform a modular A2-to-B2 ADD operation
DPD_POLY_LDCTX_A3_B2_MUL1_ULCTX	0x5425	Perform a modular A3-to-B2 MUL1 operation
DPD_POLY_LDCTX_A3_B2_MUL2_ULCTX	0x5426	Perform a modular A3-to-B2 MUL2 operation
DPD_POLY_LDCTX_A3_B2_ADD_ULCTX	0x5427	Perform a modular A3-to-B2 ADD operation
DPD_POLY_LDCTX_A0_B3_MUL1_ULCTX	0x5428	Perform a modular A0-to-B3 MUL1 operation
DPD_POLY_LDCTX_A0_B3_MUL2_ULCTX	0x5429	Perform a modular A0-to-B3 MUL2 operation
DPD_POLY_LDCTX_A0_B3_ADD_ULCTX	0x542A	Perform a modular A0-to-B3 ADD operation
DPD_POLY_LDCTX_A1_B3_MUL1_ULCTX	0x542B	Perform a modular A1-to-B3 MUL1 operation
DPD_POLY_LDCTX_A1_B3_MUL2_ULCTX	0x542C	Perform a modular A1-to-B3 MUL2 operation
DPD_POLY_LDCTX_A1_B3_ADD_ULCTX	0x542D	Perform a modular A1-to-B3 ADD operation
DPD_POLY_LDCTX_A2_B3_MUL1_ULCTX	0x542E	Perform a modular A2-to-B3 MUL1 operation
DPD_POLY_LDCTX_A2_B3_MUL2_ULCTX	0x542F	Perform a modular A2-to-B3 MUL2 operation
DPD_POLY_LDCTX_A2_B3_ADD_ULCTX	0x5430	Perform a modular A2-to-B3 ADD operation
DPD_POLY_LDCTX_A3_B3_MUL1_ULCTX	0x5431	Perform a modular A3-to-B3 MUL1 operation
DPD_POLY_LDCTX_A3_B3_MUL2_ULCTX	0x5432	Perform a modular A3-to-B3 MUL2 operation
DPD_POLY_LDCTX_A3_B3_ADD_ULCTX	0x5433	Perform a modular A3-to-B3 ADD operation

4.8 ECC Public Key Requests

4.8.1 ECC_POINT_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  nDataBytes;
unsigned char *nData;
unsigned long  eDataBytes;
unsigned char *eData;
unsigned long  buildDataBytes;
unsigned char *buildData;
unsigned long  b1DataBytes;
unsigned char *b1Data;
unsigned long  b2DataBytes;
unsigned char *b2Data;
unsigned long  b3OutDataBytes;
unsigned char *b3OutData;
```

NUM_EC_POINT_DESC defines the number of descriptors within the DPD_EC_LDCTX_kP_ULCTX_GROUP that use this request.

DPD_EC_LDCTX_kP_ULCTX_GROUP (0x5800) defines the group for all descriptors within this request. Table 24.

Table 24. ECC_POINT_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_EC_FP_AFF_PT_MULT	0x5800	Perform a PT_MULT operation in an affine system
DPD_EC_FP_PROJ_PT_MULT	0x5801	Perform a PT_MULT operation in a projective system
DPD_EC_F2M_AFF_PT_MULT	0x5802	Perform an F2M PT_MULT operation in an affine system
DPD_EC_F2M_PROJ_PT_MULT	0x5803	Perform an F2M PT_MULT operation in a projective system
DPD_EC_FP_LDCTX_ADD_ULCTX	0x5804	Perform an FP add operation
DPD_EC_FP_LDCTX_DOUBLE_ULCTX	0x5805	Perform an FP double operation
DPD_EC_F2M_LDCTX_ADD_ULCTX	0x5806	Perform an F2M add operation
DPD_EC_F2M_LDCTX_DOUBLE_ULCTX	0x5807	Perform an F2M double operation

4.8.2 ECC_2OP_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  bDataBytes;
unsigned char *bData;
unsigned long  aDataBytes;
unsigned char *aData;
unsigned long  modBytes;
```

```
unsigned char *modData;
unsigned long  outBytes;
unsigned char *outData;
```

NUM_EC_2OP_DESC defines the number of descriptors within the DPD_EC_2OP_GROUP that use this request.

DPD_EC_2OP_GROUP (0x5900) defines the group for all descriptors within this request. See Table 25.

Table 25. ECC_2OP_REQ Valid Descriptors (opId)

Descriptor	Value	Function Description
DPD_EC_F2M_LDCTX_MUL1_ULCTX	0x5900	Perform an F2M MULT1 operation

4.8.3 ECC_SPKBUILD_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  a0DataBytes;
unsigned char *a0Data;
unsigned long  a1DataBytes;
unsigned char *a1Data;
unsigned long  a2DataBytes;
unsigned char *a2Data;
unsigned long  a3DataBytes;
unsigned char *a3Data;
unsigned long  b0DataBytes;
unsigned char *b0Data;
unsigned long  b1DataBytes;
unsigned char *b1Data;
unsigned long  buildDataBytes;
unsigned char *buildData;
```

NUM_EC_SPKBUILD_DESC defines the number of descriptors within the DPD_EC_SPKBUILD_GROUP that use this request.

DPD_EC_SPKBUILD_GROUP (0x5a00) defines the group for all descriptors within this request. See Table 26.

Table 26. ECC_SPKBUILD_REQ Valid Descriptor (opId)

Descriptor	Value	Function Description
DPD_EC_SPKBUILD_ULCTX	0x5A00	Using separate values for a0-a3 and b0-b1, build a uniform data block that can be used to condense data to a point that allow it to be used with ECC operational requests.

4.8.4 ECC_PTADD_DBL_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  modBytes;
unsigned char *modData;
unsigned long  buildDataBytes;
```

```
unsigned char *buildData;
unsigned long  b2DataBytes;
unsigned char *b2Data;
unsigned long  b3DataBytes;
unsigned char *b3Data;
unsigned long  b1DataBytes;
unsigned char *b2Data;
unsigned long  b2DataBytes;
unsigned char *b2Data;
unsigned long  b3DataBytes;
unsigned char *b3Data;
```

Table 27. ECC_PTADD_DBL_REQ Valid Descriptor (opId)

Descriptor	Value	Function Description
DPD_EC_FPADD	0x5d00	Perform an FP add operation
DPD_EC_FPDBL	0x5d01	Perform an FP double operation
DPD_EC_F2MADD	0x5d02	Perform an F2M add operation
DPD_EC_F2MDBL	0x5d03	Perform an F2M double operation

4.9 IPSec Requests

4.9.1 IPSEC_CBC_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  hashKeyBytes;
unsigned char *hashKeyData;
unsigned long  cryptKeyBytes;
unsigned char *cryptKeyData;
unsigned long  cryptCtxInBytes;
unsigned char *cryptCtxInData;
unsigned long  hashInDataBytes;
unsigned char *hashInData;
unsigned long  inDataBytes;
unsigned char *inData;
unsigned char *cryptDataOut;
unsigned long  hashDataOutBytes;
unsigned char *hashDataOut;
```

NUM_IPSEC_CBC_DESC defines the number of descriptors within the DPD_IPSEC_CBC_GROUP that use this request.

DPD_IPSEC_CBC_GROUP (0x7000) defines the group for all descriptors within this request. See Table 28.

Table 28. IPSEC_CBC_REQ Valid Descriptors (opId) Descriptors

Descriptor	Value	Function Description
DPD_IPSEC_CBC_SDES_ENCRYPT_MD5	0x7000	Perform the IPSec process of encrypting in single DES using CBC mode with MD5 padding.
DPD_IPSEC_CBC_SDES_ENCRYPT_SHA	0x7001	Perform the IPSec process of encrypting in single DES using CBC mode with SHA-1 padding.
DPD_IPSEC_CBC_SDES_ENCRYPT_SHA256	0x7002	Perform the IPSec process of encrypting in single DES using CBC mode with SHA-256 padding.
DPD_IPSEC_CBC_SDES_DECRYPT_MD5	0x7003	Perform the IPSec process of decrypting in single DES using CBC mode with MD5 padding.
DPD_IPSEC_CBC_SDES_DECRYPT_SHA	0x7004	Perform the IPSec process of decrypting in single DES using CBC mode with SHA-1 padding.
DPD_IPSEC_CBC_SDES_DECRYPT_SHA256	0x7005	Perform the IPSec process of decrypting in single DES using CBC mode with SHA-256 padding.
DPD_IPSEC_CBC_TDES_ENCRYPT_MD5	0x7006	Perform the IPSec process of encrypting in triple DES using CBC mode with MD5 padding.
DPD_IPSEC_CBC_TDES_ENCRYPT_SHA	0x7007	Perform the IPSec process of encrypting in triple DES using CBC mode with SHA-1 padding.
DPD_IPSEC_CBC_TDES_ENCRYPT_SHA256	0x7008	Perform the IPSec process of encrypting in triple DES using CBC mode with SHA-256 padding.
DPD_IPSEC_CBC_TDES_DECRYPT_MD5	0x7009	Perform the IPSec process of decrypting in triple DES using CBC mode with MD5 padding.
DPD_IPSEC_CBC_TDES_DECRYPT_SHA	0x700A	Perform the IPSec process of decrypting in triple DES using CBC mode with SHA-1 padding.
DPD_IPSEC_CBC_TDES_DECRYPT_SHA256	0x700B	Perform the IPSec process of decrypting in triple DES using CBC mode with SHA-256 padding.

4.9.2 IPSEC_ECB_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  hashKeyBytes;
unsigned char *hashKeyData;
unsigned long  cryptKeyBytes;
unsigned char *cryptKeyData;
unsigned long  hashInDataBytes;
unsigned char *hashInData;
unsigned long  inDataBytes;
unsigned char *inData;
unsigned long  hashDataOutBytes;
unsigned char *hashDataOut;
unsigned char *cryptDataOut;
```

NUM_IPSEC_ECB_DESC defines the number of descriptors within the DPD_IPSEC_ECB_GROUP that use this request.

DPD_IPSEC_ECB_GROUP (0x7100) defines the group for all descriptors within this request. See Table 29.

Table 29. IPSEC_ECB_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_IPSEC_ECB_SDES_ENCRYPT_MD5	0x7100	Perform the IPSec process of encrypting in single DES using ECB mode with MD5 padding.
DPD_IPSEC_ECB_SDES_ENCRYPT_SHA	0x7101	Perform the IPSec process of encrypting in single DES using ECB mode with SHA-1 padding.
DPD_IPSEC_ECB_SDES_ENCRYPT_SHA256	0x7102	Perform the IPSec process of encrypting in single DES using ECB mode with SHA-256 padding.
DPD_IPSEC_ECB_SDES_DECRYPT_MD5	0x7103	Perform the IPSec process of decrypting in single DES using ECB mode with MD5 padding.
DPD_IPSEC_ECB_SDES_DECRYPT_SHA	0x7104	Perform the IPSec process of decrypting in single DES using ECB mode with SHA-1 padding.
DPD_IPSEC_ECB_SDES_DECRYPT_SHA256	0x7105	Perform the IPSec process of decrypting in single DES using ECB mode with SHA-256 padding.
DPD_IPSEC_ECB_TDES_ENCRYPT_MD5	0x7106	Perform the IPSec process of encrypting in triple DES using ECB mode with MD5 padding.
DPD_IPSEC_ECB_TDES_ENCRYPT_SHA	0x7107	Perform the IPSec process of encrypting in triple DES using ECB mode with SHA-1 padding.
DPD_IPSEC_ECB_TDES_ENCRYPT_SHA256	0x7108	Perform the IPSec process of encrypting in triple DES using ECB mode with SHA-256 padding.
DPD_IPSEC_ECB_TDES_DECRYPT_MD5	0x7109	Perform the IPSec process of decrypting in triple DES using ECB mode with MD5 padding.
DPD_IPSEC_ECB_TDES_DECRYPT_SHA	0x710A	Perform the IPSec process of decrypting in triple DES using ECB mode with SHA-1 padding.
DPD_IPSEC_ECB_TDES_DECRYPT_SHA256	0x710B	Perform the IPSec process of decrypting in triple DES using ECB mode with SHA-256 padding.

4.9.3 IPSEC_AES_CBC_REQ

```
unsigned long  hashKeyBytes;
unsigned char *hashKeyData;
unsigned long  cryptKeyBytes;
unsigned char *cryptKeyData;
unsigned long  cryptCtxInBytes;
unsigned char *cryptCtxInData;
unsigned long  hashInDataBytes;
unsigned char *hashInData;
unsigned long  inDataBytes;
unsigned char *inData;
unsigned char *cryptDataOut;
```

```
unsigned long  hashDataOutBytes;
unsigned char *hashDataOut;
```

NUM_IPSEC_AES_CBC_DESC defines the number of descriptors within the DPD_IPSEC_AES_CBC_GROUP that use this request.

DPD_IPSEC_AES_CBC_GROUP (0x8000) defines the group for all descriptors within this request. See Table 30.

Table 30. IPSEC_AES_CBC_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_IPSEC_AES_CBC_ENCRYPT_MD5_APAD	0x8000	Perform the IPSec process of encrypting in AES using CBC mode with MD5 auto padding.
DPD_IPSEC_AES_CBC_ENCRYPT_SHA_APAD	0x8001	Perform the IPSec process of encrypting in AES using CBC mode with SHA-1 auto padding.
DPD_IPSEC_AES_CBC_ENCRYPT_SHA256_APAD	0x8002	Perform the IPSec process of encrypting in AES using CBC mode with SHA-256 auto padding.
DPD_IPSEC_AES_CBC_ENCRYPT_MD5	0x8003	Perform the IPSec process of encrypting in AES using CBC mode with MD5.
DPD_IPSEC_AES_CBC_ENCRYPT_SHA	0x8004	Perform the IPSec process of encrypting in AES using CBC mode with SHA-1.
DPD_IPSEC_AES_CBC_ENCRYPT_SHA256	0x8005	Perform the IPSec process of encrypting in AES using CBC mode with SHA-256.
DPD_IPSEC_AES_CBC_DECRYPT_MD5_APAD	0x8006	Perform the IPSec process of decrypting in AES using CBC mode with MD5 auto padding.
DPD_IPSEC_AES_CBC_DECRYPT_SHA_APAD	0x8007	Perform the IPSec process of decrypting in AES using CBC mode with SHA-1 auto padding.
DPD_IPSEC_AES_CBC_DECRYPT_SHA256_APAD	0x8008	Perform the IPSec process of decrypting in AES using CBC mode with SHA-256 auto padding.
DPD_IPSEC_AES_CBC_DECRYPT_MD5	0x8009	Perform the IPSec process of decrypting in AES using CBC mode with MD5.
DPD_IPSEC_AES_CBC_DECRYPT_SHA	0x800A	Perform the IPSec process of decrypting in AES using CBC mode with SHA-1.
DPD_IPSEC_AES_CBC_DECRYPT_SHA256	0x800B	Perform the IPSec process of decrypting in AES using CBC mode with SHA-256.

4.9.4 IPSEC_AES_ECB_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  hashKeyBytes;
unsigned char *hashKeyData;
unsigned long  cryptKeyBytes;
unsigned char *cryptKeyData;
unsigned long  hashInDataBytes;
unsigned char *hashInData;
```

```
unsigned long  inDataBytes;
unsigned char *inData;
unsigned char *cryptDataOut;
unsigned long  hashDataOutBytes;
unsigned char *hashDataOut;
```

NUM_IPSEC_AES_ECB_DESC defines the number of descriptors within the DPD_IPSEC_AES_ECB_GROUP that use this request.

DPD_IPSEC_AES_ECB_GROUP (0x8100) defines the group for all descriptors within this request. See Table 31.

Table 31. IPSEC_AES_ECB_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_IPSEC_AES_ECB_ENCRYPT_MD5_APAD	0x8100	Perform the IPSec process of encrypting in AES using ECB mode with MD5 auto padding.
DPD_IPSEC_AES_ECB_ENCRYPT_SHA_APAD	0x8101	Perform the IPSec process of encrypting in AES using ECB mode with SHA-1 auto padding.
DPD_IPSEC_AES_ECB_ENCRYPT_SHA256_APAD	0x8102	Perform the IPSec process of encrypting in AES using ECB mode with SHA-256 auto padding.
DPD_IPSEC_AES_ECB_ENCRYPT_MD5	0x8103	Perform the IPSec process of encrypting in AES using ECB mode with MD5.
DPD_IPSEC_AES_ECB_ENCRYPT_SHA	0x8104	Perform the IPSec process of encrypting in AES using ECB mode with SHA-1.
DPD_IPSEC_AES_ECB_ENCRYPT_SHA256	0x8105	Perform the IPSec process of encrypting in AES using ECB mode with SHA-256.
DPD_IPSEC_AES_ECB_DECRYPT_MD5_APAD	0x8106	Perform the IPSec process of decrypting in AES using ECB mode with MD5 auto padding.
DPD_IPSEC_AES_ECB_DECRYPT_SHA_APAD	0x8107	Perform the IPSec process of decrypting in AES using ECB mode with SHA-1 auto padding.
DPD_IPSEC_AES_ECB_DECRYPT_SHA256_APAD	0x8108	Perform the IPSec process of decrypting in AES using ECB mode with SHA-256 auto padding.
DPD_IPSEC_AES_ECB_DECRYPT_MD5	0x8109	Perform the IPSec process of decrypting in AES using ECB mode with MD5.
DPD_IPSEC_AES_ECB_DECRYPT_SHA	0x810A	Perform the IPSec process of decrypting in AES using ECB mode with SHA-1.
DPD_IPSEC_AES_ECB_DECRYPT_SHA256	0x810B	Perform the IPSec process of decrypting in AES using ECB mode with SHA-256.

4.9.5 IPSEC_ESP_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  hashKeyBytes;
unsigned char *hashKeyData;
unsigned long  cryptKeyBytes;
```

```
unsigned char *cryptKeyData;
unsigned long  cryptCtxInBytes;
unsigned char *cryptCtxInData;
unsigned long  hashInDataBytes;
unsigned char *hashInData;
unsigned long  inDataBytes;
unsigned char *inData;
unsigned char *cryptDataOut;
unsigned long  hashDataOutBytes;
unsigned char *hashDataOut;
unsigned long  cryptCtxOutBytes;
unsigned char *cryptCtxOutData;
```

NUM_IPSEC_ESP_DESC defines the number of descriptors within the DPD_IPSEC_ESP_GROUP that use this request.

DPD_IPSEC_ESP_GROUP (0x7500) defines the group for all descriptors within this request. See Table 32.

Table 32. IPSEC_ESP_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_IPSEC_ESP_OUT_SDES_ECB_CRPT_MD5_PAD	0x7500	Process an outbound IPSec encapsulated system payload packet using single DES in ECB mode and MD5 with auto padding
DPD_IPSEC_ESP_OUT_SDES_ECB_CRPT_SHA_PAD	0x7501	Process an outbound IPSec encapsulated system payload packet using single DES in ECB mode, and SHA1 with auto padding
DPD_IPSEC_ESP_OUT_SDES_ECB_CRPT_SHA256_PAD	0x7502	Process an outbound IPSec encapsulated system payload packet using single DES in ECB mode, and SHA256 with auto padding
DPD_IPSEC_ESP_IN_SDES_ECB_DCRPT_MD5_PAD	0x7503	Process an inbound IPSec encapsulated system payload packet using single DES in ECB mode, and MD5 with auto padding
DPD_IPSEC_ESP_IN_SDES_ECB_DCRPT_SHA_PAD	0x7504	Process an inbound IPSec encapsulated system payload packet using single DES in ECB mode, and SHA1 with auto padding
DPD_IPSEC_ESP_IN_SDES_ECB_DCRPT_SHA256_PAD	0x7505	Process an inbound IPSec encapsulated system payload packet using single DES in ECB mode, and SHA256 with auto padding
DPD_IPSEC_ESP_OUT_SDES_CBC_CRPT_MD5_PAD	0x7506	Process an outbound IPSec encapsulated system payload packet using single DES in CBC mode, and MD5 with auto padding
DPD_IPSEC_ESP_OUT_SDES_CBC_CRPT_SHA_PAD	0x7507	Process an outbound IPSec encapsulated system payload packet using single DES in CBC mode, and SHA1 with auto padding
DPD_IPSEC_ESP_OUT_SDES_CBC_CRPT_SHA256_PAD	0x7508	Process an outbound IPSec encapsulated system payload packet using single DES in CBC mode, and SHA256 with auto padding
DPD_IPSEC_ESP_IN_SDES_CBC_DCRPT_MD5_PAD	0x7509	Process an inbound IPSec encapsulated system payload packet using single DES in CBC mode, and MD5 with auto padding
DPD_IPSEC_ESP_IN_SDES_CBC_DCRPT_SHA_PAD	0x750A	Process an inbound IPSec encapsulated system payload packet using single DES in CBC mode, and SHA1 with auto padding
DPD_IPSEC_ESP_IN_SDES_CBC_DCRPT_SHA256_PAD	0x750B	Process an inbound IPSec encapsulated system payload packet using single DES in CBC mode, and SHA256 with auto padding
DPD_IPSEC_ESP_OUT_TDES_CBC_CRPT_MD5_PAD	0x750C	Process an outbound IPSec encapsulated system payload packet using triple DES in CBC mode, and MD5 with auto padding

Table 32. IPSEC_ESP_REQ Valid Descriptors (opId) (continued)

DPD_IPSEC_ESP_OUT_TDES_CBC_CRPT_SHA_PAD	0x750D	Process an outbound IPSec encapsulated system payload packet using triple DES in CBC mode, and SHA1 with auto padding
DPD_IPSEC_ESP_OUT_TDES_CBC_CRPT_SHA256_PAD	0x750E	Process an outbound IPSec encapsulated system payload packet using triple DES in CBC mode, and SHA256 with auto padding
DPD_IPSEC_ESP_IN_TDES_CBC_DCRPT_MD5_PAD	0x750F	Process an inbound IPSec encapsulated system payload packet using triple DES in CBC mode, and MD5 with auto padding
DPD_IPSEC_ESP_IN_TDES_CBC_DCRPT_SHA_PAD	0x7510	Process an inbound IPSec encapsulated system payload packet using triple DES in CBC mode, and SHA1 with auto padding
DPD_IPSEC_ESP_IN_TDES_CBC_DCRPT_SHA256_PAD	0x7511	Process an inbound IPSec encapsulated system payload packet using triple DES in CBC mode, and SHA256 with auto padding
DPD_IPSEC_ESP_OUT_TDES_ECB_CRPT_MD5_PAD	0x7512	Process an outbound IPSec encapsulated system payload packet using triple DES in ECB mode, and MD5 with auto padding
DPD_IPSEC_ESP_OUT_TDES_ECB_CRPT_SHA_PAD	0x7513	Process an outbound IPSec encapsulated system payload packet using triple DES in ECB mode, and SHA1 with auto padding
DPD_IPSEC_ESP_OUT_TDES_ECB_CRPT_SHA256_PAD	0x7514	Process an outbound IPSec encapsulated system payload packet using triple DES in ECB mode, and SHA256 with auto padding
DPD_IPSEC_ESP_IN_TDES_ECB_DCRPT_MD5_PAD	0x7515	Process an inbound IPSec encapsulated system payload packet using triple DES in ECB mode, and MD5 with auto padding
DPD_IPSEC_ESP_IN_TDES_ECB_DCRPT_SHA_PAD	0x7516	Process an inbound IPSec encapsulated system payload packet using triple DES in ECB mode, and SHA1 with auto padding
DPD_IPSEC_ESP_IN_TDES_ECB_DCRPT_SHA256_PAD	0x7517	Process an inbound IPSec encapsulated system payload packet using triple DES in ECB mode, and SHA256 with auto padding
DPD_IPSEC_ESP_IN_SDES_ECB_DCRPT_MD5_PAD_CMP	0x7518	Process an inbound IPSec encapsulated system payload packet using single DES in ECB mode, and MD5 with auto padding. If the packet signature does not match the expected signature, and error will be returned (Does not apply to SEC 2.0).
DPD_IPSEC_ESP_IN_SDES_ECB_DCRPT_SHA_PAD_CMP	0x7519	Process an inbound IPSec encapsulated system payload packet using single DES in ECB mode, and SHA1 with auto padding. If the packet signature does not match the expected signature, and error will be returned (Does not apply to SEC 2.0)
DPD_IPSEC_ESP_IN_SDES_ECB_DCRPT_SHA256_PAD_CMP	0x751a	Process an inbound IPSec encapsulated system payload packet using single DES in ECB mode, andSHA256 with auto padding. If the packet signature does not match the expected signature, and error will be returned (Does not apply to SEC 2.0).
DPD_IPSEC_ESP_IN_SDES_CBC_DCRPT_MD5_PAD_CMP	0x751b	Process an inbound IPSec encapsulated system payload packet using single DES in CBC mode, and MD5 with auto padding. If the packet signature does not match the expected signature, and error will be returned (Does not apply to SEC 2.0).
DPD_IPSEC_ESP_IN_SDES_CBC_DCRPT_SHA_PAD_CMP	0x751c	Process an inbound IPSec encapsulated system payload packet using single DES in CBC mode, and SHA1 with auto padding. If the packet signature does not match the expected signature, and error will be returned (Does not apply to SEC 2.0).

Table 32. IPSEC_ESP_REQ Valid Descriptors (opId) (continued)

DPD_IPSEC_ESP_IN_SDES_CBC_DCRPT_SHA256_PAD_CMP	0x751d	Process an inbound IPSec encapsulated system payload packet using single DES in CBC mode, and SHA256 with auto padding. If the packet signature does not match the expected signature, and error will be returned (Does not apply to SEC 2.0).
DPD_IPSEC_ESP_IN_TDES_ECB_DCRPT_MD5_PAD_CMP	0x751e	Process an inbound IPSec encapsulated system payload packet using triple DES in ECB mode, and MD5 with auto padding. If the packet signature does not match the expected signature, and error will be returned (Does not apply to SEC 2.0).
DPD_IPSEC_ESP_IN_TDES_ECB_DCRPT_SHA_PAD_CMP	0x751f	Process an inbound IPSec encapsulated system payload packet using triple DES in ECB mode, and SHA1 with auto padding. If the packet signature does not match the expected signature, and error will be returned (Does not apply to SEC 2.0).
DPD_IPSEC_ESP_IN_TDES_ECB_DCRPT_SHA256_PAD_CMP	0x7520	Process an inbound IPSec encapsulated system payload packet using triple DES in ECB mode, andSHA256 with auto padding. If the packet signature does not match the expected signature, and error will be returned (Does not apply to SEC 2.0).
DPD_IPSEC_ESP_IN_TDES_ECB_DCRPT_MD5_PAD_CMP	0x7521	Process an inbound IPSec encapsulated system payload packet using triple DES in CBC mode, and MD5 with auto padding. If the packet signature does not match the expected signature, and error will be returned (Does not apply to SEC 2.0).
DPD_IPSEC_ESP_IN_TDES_ECB_DCRPT_SHA_PAD_CMP	0x7512	Process an inbound IPSec encapsulated system payload packet using triple DES in CBC mode, and SHA1 with auto padding. If the packet signature does not match the expected signature, and error will be returned (Does not apply to SEC 2.0).
DPD_IPSEC_ESP_IN_TDES_ECB_DCRPT_SHA256_PAD_CMP	0x7513	Process an inbound IPSec encapsulated system payload packet using single DES in CBC mode, and SHA256 with auto padding. If the packet signature does not match the expected signature, and error will be returned (Does not apply to SEC 2.0).

4.10 802.11 Protocol Requests

4.10.1 CCMP_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  keyBytes;
unsigned char *keyData;
unsigned long  ctxBytes;
unsigned char *context;
unsigned long  FrameDataBytes;
unsigned char *FrameData;
unsigned long  AADBytes;
unsigned char *AADData;
unsigned long  cryptDataBytes;
unsigned char *cryptDataOut;
unsigned long  MICBytes;
unsigned char *MICData;
```

NUM_CCMP_DESC defines the number of descriptors within the DPD_CCMP_GROUP that use this request.

DPD_CCMP_GROUP (0x6500) defines the group for all descriptors within this request. See Table 33.

Table 33. CCMP_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_802_11_CCMP_OUTBOUND	0x6500	Process an outbound CCMP packet.
DPD_802_11_CCMP_INBOUND	0x6501	Process an inbound CCMP packet.
DPD_802_11_CCMP_INBOUND_CMP	0x6502	Process an inbound CCMP packet and compare the packet signature with an expected value. If they don't compare, return an error (Does not apply to SEC 2.0).

4.11 SRTP Protocol Requests

4.11.1 SRTP_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  hashKeyBytes;
unsigned char *hashKeyData;
unsigned long  keyBytes;
unsigned char *keyData;
unsigned long  ivBytes;
unsigned char *ivData;
unsigned long  HeaderBytes;
unsigned long  inBytes;
unsigned char *inData;
unsigned long  ROCBytes;
unsigned long  cryptDataBytes;
unsigned char *cryptDataOut;
unsigned long  digestBytes;
unsigned char *digestData;
unsigned long  outIvBytes;
unsigned char *outIvData;
```

NUM_SRTP_DESC defines the number of descriptors within the DPD_SRTP_GROUP that use this request.

DPD_SRTP_GROUP (0x8500) defines the group for all descriptors within this request. See Table 34.

Table 34. SRTP_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_SRTP_OUTBOUND	0x8500	Process an outbound SRTP packet.
DPD_SRTP_INBOUND	0x8501	Process an inbound SRTP packet.
DPD_SRTP_INBOUND_CMP	0x8502	Process an inbound SRTP packet and compare the signature with an expected value. If the compared values differ, return an error (Does not apply to SEC 2.0).

4.12 RAID XOR Requests (Does not apply to SEC 2.0)

4.12.1 RAID_XOR_REQ

```
COMMON_REQ_PREAMBLE
unsigned char *inDataA;
unsigned char *inDataB;
unsigned char *inDataC;
unsigned char *outData;
unsigned long  opSize;
```

NUM_RAID_XOR_DESC defines the number of descriptors within the DPD_RAID_XOR_GROUP that use this request.

DPD_RAID_XOR_GROUP (0x6100) defines the group for all descriptors within this request. See Table 35.

Table 35. RAID_XOR_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_RAID_XOR	0x6100	Perform an XOR operation of either 2 or 3 of the input data blocks, and write the output.

4.13 Kasumi Cipher Requests (SEC 2.1 only)

4.13.1 KEA_CRYPT_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  ivBytes;
unsigned char *ivData
unsigned long  keyBytes;
unsigned char *keyData;
unsigned long  inBytes; /* multiple of 8 bytes */
unsigned char *inData;
unsigned long  outBytes;
unsigned char *outData; /* output length = input length */
                        /*if f9_CMP, this becomes compare value */
unsigned long  outIvBytes;
unsigned char *outIvData;
unsigned long  ctxBytes;
unsigned char *ctxData;  /* f9 integrity digest/context */
```

NUM_KEA_CRYPT_DESC defines the number of descriptors within the DPD_KEA_CRYPT_GROUP that use this request.

DPD_KEA_CRYPT_GROUP (0xa000) defines the group for all descriptors within this request. See Table 36.

Table 36. KEA_CRYPT_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_KEA_f8_CIPHER_INIT	0x6100	Perform f8 cipher function with initial cipher state set.
DPD_KEA_f8_CIPHER	0x6101	Perform f8 cipher function using cipher state remaining from previous operation.
DPD_KEA_f9_CIPHER_INIT	0x6102	Perform f9 authentication function with initial state set.
DPD_KEA_f9_CIPHER	0x6103	Perform f9 authentication function using existing state.
DPD_KEA_f9_CIPHER_FINAL	0x6104	Perform f9 authentication and finalize digest using existing state.
DPD_KEA_f9_CIPHER_INIT_FINAL	0x6105	Perform f9 authentication and finalize digest using initialized cipher state.
DPD_KEA_GSM_A53_CIPHER	0x6106	Perform single-pass 3GPP GSM A5/3 message processing.
DPD_KEA_EDGE_A53_CIPHER	0x6107	Perform single-pass 3GPP EDGE A5/2 message processing.
DPD_KEA_f9_CIPHER_FINAL_CMP	0x6108	Perform f9 authentication and finalize digest using existing cipher state. Compare final digest with expected value, and flag error if they do not match.
DPD_KEA_f9_CIPHER_INIT_FINAL_CMP	0x6109	Perform f9 authentication and finalize digest using initial cipher state. Compare final digest with expected value, and flag error if they do not match.

4.14 SSL/TLS Processing Requests (Does not apply to SEC 2.0)

Note that there are 4 different classes of requests for SSL/TLS message types; block or stream cipher, either inbound or outbound. Since each type has significantly differing content, and data mapping into descriptors differs for each type, the four separate request types have their own request structure.

4.14.1 TLS_BLOCK_INBOUND_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  hashKeyBytes;
unsigned char *hashKeyData;
unsigned long  hashOnlyBytes;
unsigned char *hashOnlyData;
unsigned long  ivBytes;
unsigned char *ivData;
unsigned long  cipherKeyBytes;
unsigned char *cipherKeyData;
unsigned long  inBytes;
unsigned char *inData;
unsigned long  MACcmpBytesl
unsigned long  outBytes;
unsigned char *outData;
unsigned long  MACoutBytes;
unsigned long  ivOutBytes;
```

```
unsigned char *ivOutData;
```

NUM_TLS_BLOCK_INBOUND_DESC defines the number of descriptors within the DPD_TLS_BLOCK_INBOUND_GROUP that use this request.

DPD_TLS_BLOCK_INBOUND_GROUP (0x9000) defines the group for all descriptors within this request. See Table 37.

Table 37. TLS_BLOCK_INBOUND_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_TLS_BLOCK_INBOUND_SDES_MD5	0x9000	Process inbound message using single DES and MD5
DPD_TLS_BLOCK_INBOUND_SDES_MD5_CMP	0x9001	Process inbound message using single DES and MD5, and comparing signatures with the expected value, returning an error if a mismatch occurs.
DPD_TLS_BLOCK_INBOUND_SDES_SHA1	0x9002	Process inbound message using single DES and SHA1.
DPD_TLS_BLOCK_INBOUND_SDES_SHA1_CMP	0x9003	Process inbound message using single DES and SHA1, and comparing signatures with the expected value, returning an error if a mismatch occurs.
DPD_TLS_BLOCK_INBOUND_SDES_SHA256	0x9004	Process inbound message using single DES and SHA256.
DPD_TLS_BLOCK_INBOUND_SDES_SHA256_CMP	0x9005	Process inbound message using single DES and SHA256, and comparing signatures with the expected value, returning an error if a mismatch occurs.
DPD_TLS_BLOCK_INBOUND_TDES_MD5	0x9006	Process inbound message using triple DES and MD5.
DPD_TLS_BLOCK_INBOUND_TDES_MD5_CMP	0x9007	Process inbound message using triple DES and MD5, and comparing signatures with the expected value, returning an error if a mismatch occurs.
DPD_TLS_BLOCK_INBOUND_TDES_SHA1	0x9008	Process inbound message using triple DES and SHA1.
DPD_TLS_BLOCK_INBOUND_TDES_SHA1_CMP	0x9009	Process inbound message using triple DES and SHA1, and comparing signatures with the expected value, returning an error if a mismatch occurs.
DPD_TLS_BLOCK_INBOUND_TDES_SHA256	0x900a	Process inbound message using triple DES and SHA256.
DPD_TLS_BLOCK_INBOUND_TDES_SHA256_CMP	0x900b	Process inbound message using triple DES and SHA256, and comparing signatures with the expected value, returning an error if a mismatch occurs.
DPD_TLS_BLOCK_INBOUND_SDES_MD5_SMAC	0x900c	Process inbound message using single DES and MD5 SMAC.
DPD_TLS_BLOCK_INBOUND_SDES_MD5_SMAC_CMP	0x900d	Process inbound message using single DES and MD5 SMAC, and comparing signatures with the expected value, returning an error if a mismatch occurs.
DPD_TLS_BLOCK_INBOUND_SDES_SHA1_SMAC	0x900e	Process inbound message using single DES and SHA1 SMAC.
DPD_TLS_BLOCK_INBOUND_SDES_SHA1_SMAC_CMP	0x900f	Process inbound message using single DES and SHA1 SMAC, and comparing signatures with the expected value, returning an error if a mismatch occurs.

Table 37. TLS_BLOCK_INBOUND_REQ Valid Descriptors (opId) (continued)

DPD_TLS_BLOCK_INBOUND_TDES_MD5_SMAC	0x9010	Process inbound message using triple DES and MD5 SMAC.
DPD_TLS_BLOCK_INBOUND_TDES_MD5_SMAC_CMP	0x9011	Process inbound message using triple DES and MD5 SMAC, and comparing signatures with the expected value, returning an error if a mismatch occurs.
DPD_TLS_BLOCK_INBOUND_TDES_SHA1_SMAC	0x9012	Process inbound message using triple DES and SHA1 SMAC.
DPD_TLS_BLOCK_INBOUND_TDES_SH1_SMAC_CMP	0x9013	Process inbound message using triple DES and SHA1 SMAC, and comparing signatures with the expected value, returning an error if a mismatch occurs.
DPD_TLS_BLOCK_INBOUND_AES_MD5	0x9014	Process inbound message using AES and MD5.
DPD_TLS_BLOCK_INBOUND_AES_MD5_CMP	0x9015	Process inbound message using single AES and MD5, and comparing signatures with the expected value, returning an error if a mismatch occurs.
DPD_TLS_BLOCK_INBOUND_AES_SHA1	0x9016	Process inbound message using single AES and SHA1.
DPD_TLS_BLOCK_INBOUND_AES_SHA1_CMP	0x9017	Process inbound message using single AES and SHA1, and comparing signatures with the expected value, returning an error if a mismatch occurs.
DPD_TLS_BLOCK_INBOUND_AES_SHA256	0x9018	Process inbound message using single AES and SHA256.
DPD_TLS_BLOCK_INBOUND_AES_SHA256_CMP	0x9019	Process inbound message using single AES and SHA256, and comparing signatures with the expected value, returning an error if a mismatch occurs.
DPD_TLS_BLOCK_INBOUND_AES_MD5_SMAC	0x901a	Process inbound message using single AES and MD5 SMAC.
DPD_TLS_BLOCK_INBOUND_AES_MD5_SMAC_CMP	0x901b	Process inbound message using single AES and MD5 SMAC, and comparing signatures with the expected value, returning an error if a mismatch occurs.
DPD_TLS_BLOCK_INBOUND_AES_SHA1_SMAC	0x901c	Process inbound message using single AES and SHA1 SMAC.
DPD_TLS_BLOCK_INBOUND_AES_SHA1_SMAC_CMP	0x901d	Process inbound message using single AES and SHA1 SMAC, and comparing signatures with the expected value, returning an error if a mismatch occurs.

4.14.2 TLS_BLOCK_OUTBOUND_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  hashKeyBytes;
unsigned char *hashKeyData;
unsigned long  ivBytes;
unsigned char *ivData;
unsigned long  cipherKeyBytes;
unsigned char *cipherKeyData;
unsigned long  hashOnlyBytes;
unsigned long *hashOnlyData;
```

```
unsigned long  mainDataBytes;
unsigned long  outBytes;
unsigned long *outData;
unsigned long  MACbytes;
unsigned long  cipherOnlyBytes;
unsigned long *cipherOnlyData;
unsigned long  ivOutBytes;
unsigned long *ivOutData;
```

NUM_TLS_BLOCK_OUTBOUND_DESC defines the number of descriptors within the DPD_TLS_BLOCK_OUTBOUND_GROUP that use this request.

DPD_TLS_BLOCK_OUTBOUND_GROUP (0x9100) defines the group for all descriptors within this request. See Table 38.

Table 38. TLS_BLOCK_OUTBOUND_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_TLS_BLOCK_OUTBOUND_SDES_MD5	0x9100	Prepare an outbound message using single DES and MD5.
DPD_TLS_BLOCK_OUTBOUND_SDES_SHA1	0x9101	Prepare an outbound message using single DES and SHA1.
DPD_TLS_BLOCK_OUTBOUND_SDES_SHA256	0x9102	Prepare an outbound message using single DES and SHA256.
DPD_TLS_BLOCK_OUTBOUND_TDES_MD5	0x9103	Prepare an outbound message using triple DES and MD5.
DPD_TLS_BLOCK_OUTBOUND_TDES_SHA1	0x9104	Prepare an outbound message using triple DES and SHA1.
DPD_TLS_BLOCK_OUTBOUND_TDES_SHA256	0x9105	Prepare an outbound message using triple DES and SHA256.
DPD_TLS_BLOCK_OUTBOUND_AES_MD5	0x9106	Prepare an outbound message using AES and MD5.
DPD_TLS_BLOCK_OUTBOUND_AES_SHA1	0x9107	Prepare an outbound message using AES and SHA1.
DPD_TLS_BLOCK_OUTBOUND_AES_SHA256	0x9108	Prepare an outbound message using AES and SHA256.

4.14.3 TLS_STREAM_INBOUND_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  hashKeyBytes;
unsigned char *hashKeyData;
unsigned long  hashOnlyBytes;
unsigned char *hashOnlyData;
unsigned long  ivBytes;
unsigned char *ivData;
unsigned long  cipherKeyBytes;
unsigned char *cipherKeyData;
unsigned long  inBytes;
unsigned char *inData;
unsigned long  MACcmpBytes;
unsigned long  outBytes;
unsigned char *outData;
unsigned long  MACoutBytes;
```

```
unsigned long  ivOutBytes;
unsigned char *ivOutData;
unsigned char *cmpData;
```

NUM_TLS_STREAM_INBOUND_DESC defines the number of descriptors within the DPD_TLS_STREAM_INBOUND_GROUP that use this request.

DPD_TLS_STREAM_INBOUND_GROUP (0x9200) defines the group for all descriptors within this request. See Table 39.

Table 39. TLS_BLOCK_INBOUND_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_TLS_STREAM_INBOUND_MD5	0x9200	Process an inbound message using MD5 and RC4 initial context.
DPD_TLS_STREAM_INBOUND_CTX_MD5	0x9201	Process an inbound message using MD5 and RC4 external context.
DPD_TLS_STREAM_INBOUNDS_SHA1	0x9202	Process an inbound message using SHA1 and RC4 initial context.
DPD_TLS_STREAM_INBOUND_CTX_SHA1	0x9203	Process an inbound message using SHA1 and RC4 external context.
DPD_TLS_STREAM_INBOUND_SHA256	0x9204	Process an inbound message using SHA256 and RC4 initial context.
DPD_TLS_STREAM_INBOUND_SHA256	0x9205	Process an inbound message using SHA256 and RC5 external context.
DPD_TLS_STREAM_INBOUND_MD5_CMP	0x9206	Process an inbound message using MD5 and RC4 initial context. Compare digest with external value, and return error if digests do not match.
DPD_TLS_STREAM_INBOUND_CTX_MD5_CMP	0x9207	Process an inbound message using MD5 and RC4 external context. Compare digest with external value, and return error if digests do not match.
DPD_TLS_STREAM_INBOUNDS_SHA1_CMP	0x9208	Process an inbound message using SHA1 and RC4 initial context. Compare digest with external value, and return error if digests do not match.
DPD_TLS_STREAM_INBOUND_CTX_SHA1_CMP	0x9209	Process an inbound message using SHA1 and RC4 external context. Compare digest with external value, and return error if digests do not match.
DPD_TLS_STREAM_INBOUND_SHA256_CMP	0x920a	Process an inbound message using SHA256 and RC4 initial context. Compare digest with external value, and return error if digests do not match.
DPD_TLS_STREAM_INBOUND_SHA256_CMP	0x920b	Process an inbound message using SHA256 and RC4 external context. Compare digest with external value, and return error if digests do not match.

4.14.4 TLS_STREAM_OUTBOUND_REQ

```
COMMON_REQ_PREAMBLE
unsigned long  hashKeyBytes;
```

```
unsigned char *hashKeyData;
unsigned long  ivBytes;
unsigned char *ivData;
unsigned long  cipherKeyBytes;
unsigned char *cipherKeyData;
unsigned long  hashOnlyBytes;
unsigned char *hashOnlyData;
unsigned long  mainDataBytes;
unsigned long  outBytes;
unsigned char *outData;
unsigned long  MACbytes;
unsigned long  ivOutBytes;
unsigned char *ivOutData;
unsigned char *cmpData;
```

NUM_TLS_STREAM_OUTBOUND_DESC defines the number of descriptors within the DPD_TLS_STREAM_OUTBOUND_GROUP that use this request.

DPD_TLS_STREAM_OUTBOUND_GROUP (0x9300) defines the group for all descriptors within this request. See Table 40.

Table 40. TLS_STREAM_OUTBOUND_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_TLS_STREAM_OUTBOUND_MD5	0x9300	Prepare an outbound message using MD5 and an RC4 initial context.
DPD_TLS_STREAM_OUTBOUND_SHA1	0x9301	Prepare an outbound message using SHA1 and an RC4 initial context.
DPD_TLS_STREAM_OUTBOUND_SHA256	0x9302	Prepare an outbound message using SHA256 and an RC4 initial context.
DPD_TLS_STREAM_OUTBOUND_CTX_MD5	0x9303	Prepare an outbound message using MD5 and an RC4 external context.
DPD_TLS_STREAM_OUTBOUND_CTX_SHA1	0x9304	Prepare an outbound message using SHA1 and an RC4 external context.
DPD_TLS_STREAM_OUTBOUND_CTX_SHA256	0x9305	Prepare an outbound message using SHA256 and an RC4 external context.

5 Sample Code

The following sections provide sample codes for DES and IPSec.

5.1 DES Sample

```
/* define the User Structure */
DES_LOADCTX_CRYPT_REQ desencReq;
...
```

```
/* fill the User Request structure with appropriate pointers */
desencReq.opId = DPD_TDES_CBC_ENCRYPT_SA_LDCTX_CRYPT ;
desencReq.channel = 0; /* dynamic channel */
desencReq.notify = (void*) notifyDes; /* callback function */
desencReq.notify_on_error = (void*) notifyDes; /* callback in case of
errors only */
desencReq.status = 0;
desencReq.ivBytes = 8; /* input iv length */
desencReq.ivData = iv_in; /* pointer to input iv */
desencReq.keyBytes = 24; /* key length */
desencReq.keyData = DesKey; /* pointer to key */
desencReq.inBytes = packet_length; /* data length */
desencReq.inData = DesData; /* pointer to data */
desencReq.outData = desEncResult; /* pointer to results */
desencReq.nextReq = 0; /* no descriptor chained */
/* call the driver */
status = Ioctl(device, IOCTL_PROC_REQ, &desencReq);
/* First Level Error Checking */
if (status != 0) {
..
}
...
void notifyDes (void)
{
/* Second Level Error Checking */
if (desencReq.status != 0) {
..
}
..)
```

5.2 IPSEC Sample

```
/* define User Requests structures */
IPSEC_CBC_REQ ipsecReq;
....
/* Ipsec dynamic descriptor triple DES with SHA-1 authentication */
ipsecReq.opId = DPD_IPSEC_CBC_TDES_ENCRYPT_SHA_PAD;
ipsecReq.channel = 0;
ipsecReq.notify = (void *) notifyFunc;
ipsecReq.notify_on_error = (void *) notifyFunc;
ipsecReq.status = 0;
ipsecReq.hashKeyBytes = 16; /* key length for HMAC SHA-1 */
ipsecReq.hashKeyData = authKey; /* pointer to HMAC Key */
ipsecReq.cryptCtxInBytes = 8; /* length of input iv */
ipsecReq.cryptCtxInData = in_iv; /* pointer to input iv */
ipsecReq.cryptKeyBytes = 24; /* DES key length */
ipsecReq.cryptKeyData = EncKey; /* pointer to DES key */
```

```
ipsecReq.hashInDataBytes = 8; /* length of data to be hashed only */
ipsecReq.hashInData = PlainText; /* pointer to data to be
hashed only */
ipsecReq.inDataBytes = packet_length-8; /* length of data to be
hashed and encrypted */
ipsecReq.inData = &PlainText[8]; /* pointer to data to be
hashed and encrypted */
ipsecReq.cryptDataOut = Result; /* pointer to encrypted results */
ipsecReq.hashDataOutBytes = 20; /* length of output digest */
ipsecReq.hashDataOut = digest; /* pointer to output digest */
ipsecReq.nextReq = 0; /* no chained requests */
/* call the driver */
status = Ioctl(device, IOCTL_PROC_REQ, &ipsecReq);
/* First Level Error Checking */
if (status != 0) {
...
}
...
void notifyFunc (void)
{
/* Second Level Error Checking */
if (ipsecReq.status != 0) {
...
}
..)
```

6 Linux Environment

This section describes the driver's adaptation to and interaction with the Linux operating system as applied to PPC processors

6.1 Installation

6.1.1 Driver Source

The SEC 2.x driver installs into Linux as a loadable module. To build the driver as a module, it must be installed into the kernel source tree to be included in the kernel build process. The makefile included with the distribution assumes this inclusion. As delivered, this directory is defined as `[kernelroot]/drivers/sec2`

Once the driver source is installed, and the kernel source (and modules) are built, module dependency lists updated, and the built objects are installed in the target filesystem, the driver, (named `sec2drv.o`) is ready for loading when needed.

6.1.2 Device Inode

Kernel processes may call the driver's functionality directly. On the other hand, user processes must use the kernel's I/O interface to make driver requests. The only way for user processes to do this it to open the device as a file with the `open()` system call to get a file descriptor, and then make requests through `ioctl()`. Thus the system will need a device file created to assign a name to the device.

The driver functions as a `char` device in the target system. As shipped, the driver assumes that the device major number will be assigned dynamically, and that the minor number will always be zero, since only one instance of the driver is supported.

Creation of the device's naming inode may be done manually in a development setting, or may be driven by a script that runs after the driver module loads, and before any user attempts to open a path to the driver. Assuming the module loaded with a dynamically assigned major number of 254 (look for `sec2` in `/proc/devices`), then the shell command to accomplish this would normally appear as:

```
$ mknod c 254 0 /dev/sec2
```

Once this is done, user tasks can make requests to the driver under the device name `/dev/sec2`.

6.2 Operation

6.2.1 Driver Operation in Kernel Mode

Operation of the SEC 2x core under kernel mode is relatively straightforward. Once the driver module has loaded, which will initialize the device, direct calls to the `ioctl()` entry (named `SEC2_ioctl` in the driver) can be made, the first two arguments may effectively be ignored.

In kernel mode, request completion may be handled through the standard use of notification callbacks in the request. The example suite available with the driver shows how this may be accomplished; this suite uses a mutex that the callback will release in order to allow the request to complete, although the caller may make use of any other type of event mechanism that suits their preference.

Logical to physical memory space translation is handled internal to the driver.

6.2.2 Driver Operation in User Mode

Operation of the SEC 2x core in user mode is slightly more complex than in kernel mode. In particular, the transition from user to kernel memory space creates two complications for user mode operation:

- User memory buffers can't be passed directly to the driver; instead, in this driver edition, the user must allocate and place data in kernel memory buffer for operation. This can be accomplished via `SEC2_MALLOC`, `SEC2_FREE`, `SEC2_COPYFROM`, and `SEC2_COPYTO` requests (see Section 3.3.1, “I/O Control Codes,” for more information).

CAUTION

Extreme caution must be exercised by the user in transferring memory in this fashion; kernel memory space may easily be corrupted by the caller, causing target system instability.

- Standard notification callbacks cannot work, since the routines to perform the callback are in user memory space, and cannot safely execute from kernel mode. In their place, standard POSIX signals can be used to indicate I/O completion by placing the process ID of the user task in the notification members of the request, and flagging `NOTIFY_IS_PID` in the `notifyFlags` member. The driver uses `SIGUSR1` to indicate normal request completions, and `SIGUSR2` to indicate error completions.

The example suite available with the driver illustrates the contrast between the two different application environments. Within the `testAll.c` file, there is a set of functions that shows the difference between the two operations. Building the example testing application with `__KERNEL__` on (building a kernel mode test) shows the installation and usage of standard completion callbacks and a mutex used for interlock. Conversely, building the example testing application with `USERMODE` turned on shows the installation of signal handlers and their proper setup.

In `USERMODE`, this example also shows one possible means for handling the user to kernel memory transition via the use of three functions for transferring user buffers to and from kernel memory.

7 VxWorks Environment

The following sections describe the installation of the SEC2 security processor software drivers, BSP integration, and distribution archives.

7.1 Installation

To install the software drivers, extract the archive containing the driver source files into a suitable installation directory. If you want the driver and tests to be part of a standard VxWorks source tree, place them as follows:

Driver:	`$(WIND_BASE)/target/src/drv/crypto`
Tests:	`$(WIND_BASE)/target/src/drv/crypto/test`

Once the modules are installed, the driver image may be built per the following instructions.

7.2 Building the Driver Modules

Specific makefiles that can support building of the driver and it's example application are no longer provided, due to the differences between the build environment between VxWorks Base-5 and Base-6. For this reason, it is assumed that the builder will create a workbench project that builds the driver as a downloadable kernel module, and is familiar with this process.

The driver project will need it's build properties amended to build the driver properly. Under "Build Properties", select the "Build Macros" tab. Within the "DEFINES" variable, add "-DVXWORKS" as one of the definitions. Also, adding "-DDBG" will assist in the initial driver integration process.

Once built, the driver may simply be downloaded to a running BSP for testing.

7.3 BSP Integration

Once the modules are built, they should be linked directly with the user's board support package, to become integral part of the board image.

In VxWorks, the file `sysLib.c` contains the initialization functions, the memory/address space functions, and the bus interrupt functions. It is recommended to call the function `SEC2DriverInit` directly from `sysLib.c`.

In the process of initialization, the driver calls a specialized function name `sysGetPeripheralBase()`, which returns a pointer to the base location of the peripheral device block in the processor (often defined by the CCSBAR register in some PowerQUICC III processors). The driver uses this address and an offset to locate the SEC core on the system bus. This is not a standard BSP function, the integrator will need to provide it, or a substitute method for locating CCSBAR.

The security processor will be initialized at board start-up, with all the other devices present on the board.

8 Porting

This section describes probable areas of developer concern with respect to porting the driver to other operating systems or environments.

At this time, this driver has been ported to function on both VxWorks and Linux operating systems. Most of the internal functionality is independent of the constructs of a specific operating system, but there necessarily are interface boundaries between them where things must be addressed.

Only a few of the files in the driver's source distribution contain specific dependencies on operating system components; this is intentional. Those specific files are as follows:

- `Sec2Driver.h`
- `sec2_init.c`
- `sec2_io.c`

8.1 Header Files

`Sec2Driver.h`

This header file is meant to be local (private) to the driver itself, and as such, is responsible for including all needed operating system header files, and casts a series of macros for specific system calls

Of particular interest, this header casts local equivalents macros for the following:

`malloc`	Allocate a block of system memory with the operating system's heap allocation mechanism.
`free`	Return a block of memory to the system heap
`semGive`	Release a mutex semaphore
`semTake`	Capture and hold a mutex semaphore

8.2 C Source Files

sec2_init.c:

sec2_init.c performs the basic initialization of the device and the driver. It is responsible for finding the base address of the hardware and saving it in IOBaseAddress for later reference.

For Linux, this file also contains references to register/unregister the driver as a kernel module, and to manage it's usage/link count

sec2_io.c:

sec2_io.c contains functions to establish:

- Channel interlock semaphores (IOInitSemaphores)
- The ISR message queue (IOInitQs)
- Driver service function registration with the operating system (IORegisterDriver)
- ISR connection/disconnection (IOConnectInterrupt)

8.3 Interrupt Service Routine

The ISR will queue processing completion result messages onto the IsrMsgQId queue. ProcessingComplete() pends on this message queue. When a message is received, the completion task will execute the appropriate callback routine based on the result of the processing. When the end-user application prepares the request to be executed, callback functions can be defined for nominal processing as well as error case processing. If the callback function was set to NULL when the request was prepared then no callback function will be executed. These routines will be executed as part of the device driver so any constraints placed on the device driver will also be placed on the callback routines.

8.4 Conditional Compilation

See the makefile for specifics on the default build of the driver

8.5 Debug Messaging

The driver includes a DBG define that allows for debug message output to the developer's console. If defined in the driver build, debug messages will be sent from various components in the driver to the console.

Messages come from various sections of the driver, and a bitmask is kept in a driver global variable so that the developer can turn message sources on or off as required. This global is named SEC2DebugLevel and contains an OR'ed combination of any of the bits shown in Table 41:

Table 41. Debug Messages

Bit	Description
DBGTXT_SETRQ	Messages from request setup operations (new requests inbound from the application)
DBGTXT_SVCRQ	Messages from servicing device responses (ISR/deferred service routine handlers) outbound to the application
DBGTXT_INITDEV	Messages from the device/driver initialization process

Table 41. Debug Messages (continued)

Bit	Description
DBGTXT_DPDSHOW	Shows the content of a constructed DPD before it is handed to the security core
DBGTXT_INFO	Shows a short banner at device initialization describing the driver and hardware version
DBGTXT_DATASHOW	Displays the content of all DPD-referenced data buffers before the request is passed to the hardware
DBGTXT_DESCBUF	Displays contents of the descriptor buffer after processing to show what was fetched by the hardware

In normal driver operation (not in a development setting), the DBG definition should be left undefined for best performance.

8.6 Distribution Archive

For this release, the distribution archive consists of the source files listed in this section. Note that the user may wish to reorganize header file locations consistent with the file location conventions appropriate for their system configuration. See Table 42.

Table 42. Reference Driver Files

Header	Description
Sec2.h	Primary public header file for all users of the driver
Sec2Driver.h	Driver/hardware interfaces, private to the driver itself
Sec2Descriptors.h	DPD-type definitions
Sec2Notify.h	Structures for ISR/main thread communication
sec2_dpd_Table.h	DPD construction constants
sec2_cha.c	CHA mapping and management
sec2_dpd.c	DPD construction functionality
sec2_init.c	Device/driver initialization code
sec2_io.c	Basic register I/O primitives
sec2_ioctl.c	Operating system interfaces
sec2_request.c	Request/response management
sec2_sctrMap.c	Scatter buffer identification and mapping
sec2isr.c	Interrupt service routine

9 Revision History

Table 43 provides a revision history for this document.

Table 43. Revision History

Rev. Number	Date	Substantive Change(s)
0	10/18/2005	Initial Release
1	08/17/2006	Tested support for 2.2 core. Full dynamic configuration for the SEC. Added support for PKEU modular inverse operations. Added two descriptor AES-CMAC examples.

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

How to Reach Us:

Home Page:
www.freescale.com

email:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
1-800-521-6274
480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064, Japan
0120 191014
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate,
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447
303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: SEC2XSWUG
Rev. 1
08/2006

freescale™
semiconductor