SEC 2.0 Reference Device Driver User's Guide

1 Overview

The SEC2 device driver manages the operation of the SEC 2.0 commonly instantiated into PowerQUICC processors. It is a fully functional component, meant to serve as an example of application interaction with the SEC2 core.

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 Section 8, "Porting" addresses them.

Testing has been accomplished on VxWorks 5.5 and LinuxPPC using kernel version 2.4.27.

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 SEC2 core.

Process requests comprise the majority of the requests and all are executed using the same `ioctl()` access point. Structures needed to compose these requests are described in detail in Section 3.3.6, "Process Request Structures."

Throughout the document, the acronyms CHA (crypto hardware accelerator) and EU (execution unit) are used interchangeably.

Contents

1. Overview 1
2. Device Driver Components 3
3. User Interface 5
4. Individual Request Type Descriptions 14
5. Sample Code 36
6. Linux Environment 39
7. VxWorks Environment 40
8. Porting 41

This document contains information on a new product. Specifications and information herein are subject to change without notice.

Both acronyms indicate the device's functional block that performs the crypto functions requested. For further details on the device see the Hardware Reference Manual.

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 1 contains acronyms and abbreviations that are used in this user's guide.

Table 1. Acronyms and Abbreviations

Term	Meaning
AESA	AES accelerator—This term is synonymous with AESU in the *MPC18x User's Manual* and other documentation.
AFHA	ARC-4 hardware accelerator—This term is synonymous with AFEU in the *MPC18x User's Manual* and other 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 *MPC18x User's Manual* and other documentation.
CTX	Context
DESA	DES accelerator—This term is synonymous with DEU in the *MPC18x User's Manual* and other 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
KEA	Kasumi encryption acceleration
MD	Message digest
MDHA	Message digest hardware accelerator—This term is synonymous with MDEU in the *MPC18x User's Manual* and other documentation.
OS	Operating system
PK	Public key
PKHA	Public key hardware accelerator—This term is synonymous with PKEU in the *MPC18x User's Manual* and other documentation.

Table 1. Acronyms and Abbreviations (continued)

Term	Meaning
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 systems provided by VxWorks Company.

2 Device Driver Components

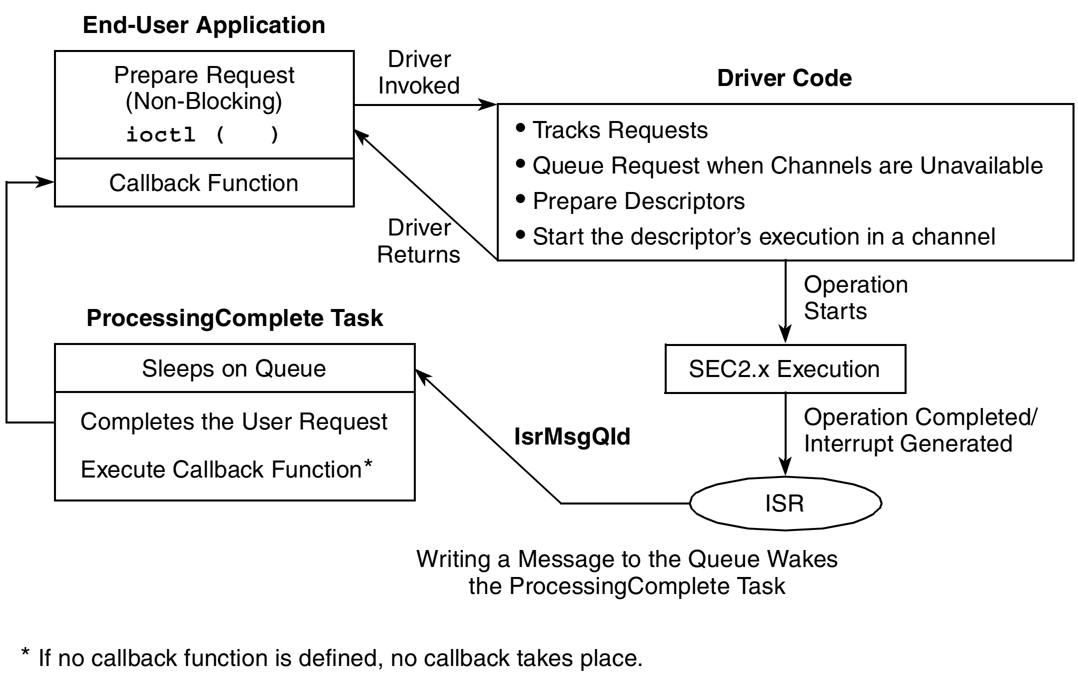
This section is provided to help users understand the internal structure of the device driver.

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.



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 24 requests. The driver will reject requests with an error when the queue is full.
- `ProcessingComplete()` is spawned then pends on the `IsrMsgQId` which 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 (data packet descriptor) 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 SEC2 device, the calling application populates a request structure with information describing the request. 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 I/O control code. This code will specify the driver-specific operation to be performed by the device in question. The third argument is the pointer to the SEC2 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.
- While all requests have a “`channel`” member, it's presence is a holdover from earlier variations of the security engine. For SEC2, it no longer has a valid use, and is retained solely to maintaining request compatibility for applications written for older security engines.
- 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 operation as indicated by the interrupt. The valid values for this status member are `DONE` (normal status) or `ERROR` (error status).
- 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.

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` function. 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 the `ioctl` function, and the second one after the completion of the request. The second level is possible only if the request was done with at least the `notify_on_error` member of the user request structure. If the notification/callback function has not been requested, 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 value of 0x0800.

Table 2. Second and Third Arguments in the `ioctl` Function

I/O Control Code (Second Argument in `ioctl` Function)	Third Argument in `ioctl` Function
SEC2_PROC_REQ	Pointer to user's request structure
SEC2_GET_STATUS	Pointer to a STATUS_REQ
SEC2_MALLOC	Pointer to be assigned to a block of kernel memory for holding caller data to be operated upon
SEC2_FREE	Pointer to free a block originally allocated by SEC2_MALLOC
SEC2_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 SEC2_MALLOC
SEC2_COPYTO	Pointer to type MALLOC_REQ, which will hold information about a user buffer that will be copied from kernel memory space allocated by SEC2_MALLOC back to a user's buffer.

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.

Table 3. Channel Defines

Define	Description
NUM_AFHAS	Number of ARC4 CHAs
NUM_DESAS	Number of DES CHAs
NUM_MDHAS	Number of MD CHAs
NUM_RNGAS	Number of RNG CHAs
NUM_PKHAS	Number of PK CHAs
NUM_AESAS	Number of AESA CHAs

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 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.

Table 4. 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 follows:

Table 5. Callback Error Status Return Code

Define	Description	Value
SEC2_SUCCESS	Successful completion of request	0
SEC2_MEMORY_ALLOCATION	Driver can't 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
SEC2_INVALID_CHA_TYPE	Requested CHA doesn't exist	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

Table 5. Callback Error Status Return Code (continued)

Define	Description	Value
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 DPD's 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
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

Table 5. Callback Error Status Return Code (continued)

Define	Description	Value
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

3.3.5.1 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;
```

3.3.5.2 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 the a copy of the same 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.

```
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;
```

opId	operation Id which identifies what type of request this is. It is normally associated with a specific type of cryptographic operation, see Section 4, “Individual Request Type Descriptions” 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 Section 3.3.7, “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. It exists for legacy compatibility reasons, and is no longer useful for SEC2.
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. Refer back to 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. Refer back to `notifyFlags` for more info.
`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 2.0 processor is 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. It contains these elements:

`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.

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.

Table 6. 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.

Table 7. 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.

Table 8. 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.

Table 9. ARC4_LOADCTX_CRYPT_REQ Valid Descriptor (opId)

Descriptor	Value	Function Description
DPD_RC4_LDCTX_CRYPT_ULCTX	0x3400	Load context, encrypt using RC4, 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.

Table 10. ARC4_LOADKEY_CRYPT_UNLOADCTX_REQ Valid Descriptor (opId)

Descriptor	Value	Function Description
DPD_RC4_LDKEY_CRYPT_ULCTX	0x3500	Load the cipher key, encrypt using RC4 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;
```

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.

Table 11. HASH_REQ Valid Descriptors (0x4400) (opId)

Descriptors	Value	Function Description
DPD_SHA256_LDCTX_HASH_ULCTX	0x4400	Load context, compute digest using SHA-256 hash algorithm, then save the resulting context
DPD_MD5_LDCTX_HASH_ULCTX	0x4401	Load context, compute digest using MD5 hash algorithm, then save the resulting context
DPD_SHA_LDCTX_HASH_ULCTX	0x4402	Load context, compute using SHA-1 hash algorithm, then save the resulting context
DPD_SHA256_LDCTX_IDGS_HASH_ULCTX	0x4403	Load context, compute digest with SHA-256 IDGS hash algorithm, then store the resulting context
DPD_MD5_LDCTX_IDGS_HASH_ULCTX	0x4404	Load context, compute digest with MD5 IDGS hash algorithm, then store the resulting context
DPD_SHA_LDCTX_IDGS_HASH_ULCTX	0x4405	Load context, compute digest with SHA-1 IDGS hash algorithm, then store the resulting context

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.

Table 12. HASH_REQ Valid Descriptors (0x4500) (opId)

Descriptors	Value	Function Description
DPD_SHA256_LDCTX_HASH_PAD_ULCTX	0x4500	Compute digest with pre-padded data using an SHA-256 hash algorithm then store the resulting context
DPD_MD5_LDCTX_HASH_PAD_ULCTX	0x4501	Compute digest with pre-padded data using an MD5 hash algorithm then store the resulting context
DPD_SHA_LDCTX_HASH_PAD_ULCTX	0x4502	Compute digest with pre-padded data using an SHA-1 hash algorithm then store the resulting context
DPD_SHA256_LDCTX_IDGS_HASH_PAD_ULCTX	0x4503	Compute digest with pre-padded data using an SHA-256 IDGS hash algorithm then store the resulting padded context
DPD_MD5_LDCTX_IDGS_HASH_PAD_ULCTX	0x4504	Compute digest with pre-padded data using an MD5 IDGS hash algorithm then store the resulting padded context
DPD_SHA_LDCTX_IDGS_HASH_PAD_ULCTX	0x4505	Compute digest with pre-padded data using an SHA-1 IDGS hash algorithm then store the resulting padded context

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;
```

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.

Table 13. HMAC_PAD_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_SHA256_LDCTX_HMAC_ULCTX	0x4A00	Load context, then use an SHA-256 hash algorithm, then store the resulting HMAC context
DPD_MD5_LDCTX_HMAC_ULCTX	0x4A01	Load context, then use an MD5 hash algorithm, then store the resulting HMAC context
DPD_SHA_LDCTX_HMAC_ULCTX	0x4A02	Load context, then use an SHA-1 hash algorithm, then store the resulting HMAC context
DPD_SHA256_LDCTX_HMAC_PAD_ULCTX	0x4A03	Load context, then use an SHA-256 IDGS hash algorithm, then store the resulting padded HMAC context
DPD_MD5_LDCTX_HMAC_PAD_ULCTX	0x4A04	Load context, then use an MD5 IDGS hash algorithm, then store the resulting padded HMAC context
DPD_SHA_LDCTX_HMAC_PAD_ULCTX	0x4A05	Load context, then use an SHA-1 IDGS hash algorithm, then store the resulting padded HMAC context

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.

Table 14. 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.

Table 15. MOD_EXP_REQ Valid Descriptor (opId)

Descriptors	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.

Table 16. MOD_SS_EXP_REQ Valid Descriptor (opId)

Descriptors	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.

Table 17. 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.

Table 18. 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_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.

Table 19. 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
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

Table 19. MOD_2OP_REQ Valid Descriptors (opId) (continued)

Descriptors	Value	Function Description
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 n 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 20. 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.

Table 21. 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.

Table 22. 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 23. `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.

Table 24. `IPSEC_CBC_REQ` Valid Descriptors (`opId`) Descriptors

Descriptor	Value	Function Description
DPD_IPSEC_CBC_SDES_ENCRYPT_MD5_PAD	0x7000	Perform the IPSec process of encrypting in single DES using CBC mode with MD5 padding
DPD_IPSEC_CBC_SDES_ENCRYPT_SHA_PAD	0x7001	Perform the IPSec process of encrypting in single DES using CBC mode with SHA-1 padding
DPD_IPSEC_CBC_SDES_ENCRYPT_SHA256_PAD	0x7002	Perform the IPSec process of encrypting in single DES using CBC mode with SHA-256 padding
DPD_IPSEC_CBC_SDES_DECRYPT_MD5_PAD	0x7003	Perform the IPSec process of decrypting in single DES using CBC mode with MD5 padding
DPD_IPSEC_CBC_SDES_DECRYPT_SHA_PAD	0x7004	Perform the IPSec process of decrypting in single DES using CBC mode with SHA-1 padding
DPD_IPSEC_CBC_SDES_DECRYPT_SHA256_PAD	0x7005	Perform the IPSec process of decrypting in single DES using CBC mode with SHA-256 padding
DPD_IPSEC_CBC_TDES_ENCRYPT_MD5_PAD	0x7006	Perform the IPSec process of encrypting in triple DES using CBC mode with MD5 padding
DPD_IPSEC_CBC_TDES_ENCRYPT_SHA_PAD	0x7007	Perform the IPSec process of encrypting in triple DES using CBC mode with SHA-1 padding
DPD_IPSEC_CBC_TDES_ENCRYPT_SHA256_PAD	0x7008	Perform the IPSec process of encrypting in triple DES using CBC mode with SHA-256 padding
DPD_IPSEC_CBC_TDES_DECRYPT_MD5_PAD	0x7009	Perform the IPSec process of decrypting in triple DES using CBC mode with MD5 padding
DPD_IPSEC_CBC_TDES_DECRYPT_SHA_PAD	0x700A	Perform the IPSec process of decrypting in triple DES using CBC mode with SHA-1 padding
DPD_IPSEC_CBC_TDES_DECRYPT_SHA256_PAD	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.

Table 25. IPSEC_ECB_REQ Valid Descriptors (opId)

Descriptors	Value	Function Description
DPD_IPSEC_ECB_SDES_ENCRYPT_MD5_PAD	0x7100	Perform the IPSec process of encrypting in single DES using ECB mode with MD5 padding
DPD_IPSEC_ECB_SDES_ENCRYPT_SHA_PAD	0x7101	Perform the IPSec process of encrypting in single DES using ECB mode with SHA-1 padding
DPD_IPSEC_ECB_SDES_ENCRYPT_SHA256_PAD	0x7102	Perform the IPSec process of encrypting in single DES using ECB mode with SHA-256 padding
DPD_IPSEC_ECB_SDES_DECRYPT_MD5_PAD	0x7103	Perform the IPSec process of decrypting in single DES using ECB mode with MD5 padding
DPD_IPSEC_ECB_SDES_DECRYPT_SHA_PAD	0x7104	Perform the IPSec process of decrypting in single DES using ECB mode with SHA-1 padding
DPD_IPSEC_ECB_SDES_DECRYPT_SHA256_PAD	0x7105	Perform the IPSec process of decrypting in single DES using ECB mode with SHA-256 padding
DPD_IPSEC_ECB_TDES_ENCRYPT_MD5_PAD	0x7106	Perform the IPSec process of encrypting in triple DES using ECB mode with MD5 padding
DPD_IPSEC_ECB_TDES_ENCRYPT_SHA_PAD	0x7107	Perform the IPSec process of encrypting in triple DES using ECB mode with SHA-1 padding

Table 25. `IPSEC_ECB_REQ` Valid Descriptors (`opId`) (continued)

DPD_IPSEC_ECB_TDES_ENCRYPT_SHA256_PAD	0x7108	Perform the IPSec process of encrypting in triple DES using ECB mode with SHA-256 padding
DPD_IPSEC_ECB_TDES_DECRYPT_MD5_PAD	0x7109	Perform the IPSec process of decrypting in triple DES using ECB mode with MD5 padding
DPD_IPSEC_ECB_TDES_DECRYPT_SHA_PAD	0x710A	Perform the IPSec process of decrypting in triple DES using ECB mode with SHA-1 padding
DPD_IPSEC_ECB_TDES_DECRYPT_SHA256_PAD	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.

Table 26. `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

Table 26. IPSEC_AES_CBC_REQ Valid Descriptors (opId) (continued)

Descriptors	Value	Function Description
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.

Table 27. 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.

Table 28. `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

Table 28. IPSEC_ESP_REQ Valid Descriptors (opId) (continued)

Descriptors	Value	Function Description
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
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

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.

Table 29. `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	0x8101	Process an inbound CCMP packet

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.

Table 30. 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

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 SEC2 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 SEC2 device 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 SEC2 device 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:

1. 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).
   Note: *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.
2. 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.

6.2.3 Driver Module License Macro

A common necessity of loadable modules for Linux is the inclusion of a license macro (MODULE_LICENSE) that declares a string defining the type of license terms under which the module's code has been published. In the case of the SEC2 driver module, this code is delivered in source form under the terms of a restricted license agreement. Therefore, this macro has been passed a name of "Freescale Restricted" to acknowledge the existence of this agreement.

When loading the driver object, the existence of a non-GPL, non-BSD license string will cause a warning message to be printed to the console, stating that loading a module with a proprietary license will "taint" the kernel. This message is normal, expected, and will not cause any adverse operation of your running system.

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 in:

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 Interface Modules

Throughout the remainder of the installation instructions, the variables provided below are used:

Table 31. VxWorks Interface Module Variables

Variable	Definition
CpuFamily	Specifies the target CPU family, such as PPC85XX
ToolChain	Specifies the tools, such as `gnu`
SecurityProcessor	Specifies the target security processor, should be `SEC2` for this driver

The following steps are used to build drivers and/or the driver test and exercise code:

1. Go to the command prompt or shell
2. Execute `torVars` to set up the Tornado command line build environment.
3. Run make in the driver or test installation directory by use of the following command:
   ```
   make CPU=cpuFamily TOOL=toolChain SP=securityProcessor
   example: make CPU=PPC85XX TOOL=gnu SP=SEC2)
   ```

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 SEC2 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:

- `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:

`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
`__vpa`	Translate a logical address to a physical address for hardware DMA (if both are equivalent, does nothing).

8.2 C Source Files

`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` 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 ORed combination of any of the following bits:

`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.
`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.

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.

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

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
(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 2666 8080
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
(800) 441-2447
303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

SEC2SWUG
Rev. 0
02/2005