Freescale Semiconductor
Application Note

Document Number: AN2755
Rev. 1, 04/2010

SEC 2.0 Descriptor Programmer's Guide

by *Networking and Multimedia Group*
Freescale Semiconductor, Inc.
Austin, TX

This application note is offered as a supplement to the reference manuals of the PowerQUICC II Pro and PowerQUICC III integrated communications processors that incorporate the SEC 2.0. The purpose is to assist the user in understanding and creating descriptors in the event that the user has more specific requirements than those covered by the SEC 2.0 reference device driver. This application note assumes the reader is already basically familiar with the SEC 2.0 architecture, as explained in the applicable device reference manual.

Contents

1 SEC 2.0 Data Packet Descriptor Overview

The SEC 2.0 has DMA capability to off-load data movement and encryption operations from the PowerQUICC CPU core. As the system controller, the CPU core of the PowerQUICC maintains a record of current secure sessions, as well as the corresponding keys and contexts of those sessions. Once the CPU core has determined a security operation is required, it can either directly write keys, context, and data to the SEC (SEC in target mode), or the CPU core can create a “data packet descriptor” to guide the SEC through the security operation, with the SEC acting as an internal bus master. The descriptor can be created in main memory, any memory local to the SEC, or written directly to the data packet descriptor buffer in the SEC crypto-channel.

2 Descriptor Structure

SEC descriptors are conceptually similar to descriptors used by most devices with DMA capability. The descriptors have a fixed length of 64 bytes, that is, eight long-words, consisting of one “header dword” and seven “pointer dwords.” Figure 1 shows the descriptor format.

	0 15	16	17 23	24 31	32 63
Header Dword	Header				Reserved
Pointer Dword 0	Length0	J1	Extent0	-	Pointer0
Pointer Dword 1	Length1	J2	Extent1	-	Pointer1
Pointer Dword 2	Length2	J3	Extent2	-	Pointer2
Pointer Dword 3	Length3	J4	Extent3	-	Pointer3
Pointer Dword 4	Length4	J5	Extent4	-	Pointer4
Pointer Dword 5	Length5	J6	Extent5	-	Pointer5
Pointer Dword 6	Length6	J7	Extent6	-	Pointer6

Figure 1. Descriptor Format

The header dword specifies the security operation to be performed, the execution unit(s) needed, and the modes for each execution unit. The pointer dwords, all of which have the same format, contain pointer and length information for locating input or output data parcels (such as keys, context, or text-data). The large number of pointers provided in the descriptor allows for multi-algorithm operations that require fetching of multiple keys, as well as fetch and return of contexts. Any pointer dword that is not needed can be given a length of zero, and the channel will skip over the corresponding operations.

SEC descriptors include scatter/gather capability, which means that each pointer in a descriptor can be either a direct pointer to a contiguous parcel of data, or can be a pointer to a “link table” which is a list of pointers and lengths used to assemble the data parcel. When a link table is used to read input data, this is referred to as a “gather” operation; when used to write output data, it is referred to as a “scatter” operation.

3 Descriptor Header

Descriptors are created by the host to guide the SEC through required cryptographic operations. The descriptor header defines the operations to be performed, the mode for each operation, and the ordering of the inputs and outputs in the body of the descriptor. The SEC 2.0 device drivers allow the host to create proper headers for each cryptographic operation. Figure 2 shows the descriptor header.

	0 3	4 11	12 15	16 23	24 28	29	30	31
Field	Op_0		Op_1		Desc_Type	—	DIR	DN
Subfield	EU_SELECT	MODE_DATA	EU_SELECT	MODE_DATA	—	—	—	—
Reset	0x0000_0000							
R/W	R/W							
Addr	Channel_X 0x0_080							

Figure 2. Descriptor Header

Table 1 defines the descriptor header fields.

Table 1. Descriptor Header Field Descriptions

Bits	Name	Description
0–11	Op_0	Op_0 contains two subfields, EU_SELECT and MODE_DATA. Figure 3 shows the subfield detail. EU_SELECT[0–3]—Programs the channel to select a primary EU of a given type. Table 2 lists the possible EU_SELECT values. MODE_DATA[4–11]—Programs the primary EU mode data. The mode data is specific to the chosen EU. This data is passed directly to bits 0–7 of the specified EU mode register.
12–23	Op_1	Op_1 contains two subfields, EU_SELECT and MODE_DATA. Figure 3 shows the subfield detail. EU_SELECT[12–15]—Programs the channel to select a secondary EU of a given type. Table 2 lists the possible EU_SELECT values. MODE_DATA[16–23]—Programs the secondary EU mode data. The mode data is specific to the chosen EU. This data is passed directly to bits 0–7 of the specified EU mode register. The MDEU is the only valid secondary EU. Values for Op_1 EU_SELECT other than 0011 (MDEU) or 0000 (No EU selected) will result in an unrecognized header error condition. Selecting 0011 (MDEU) for both primary and secondary EU will also create an error condition.
24–28	Desc_Type	Descriptor Type—Each type of descriptor determines the following attributes for the corresponding data length/pointer pairs: the direction of the data flow, which EU is associated with the data, and which internal EU address is used. Table 6 lists the valid types of descriptors.
29	—	Reserved

Table 1. Descriptor Header Field Descriptions (continued)

Bits	Name	Description
30	DIR	Direction: direction of overall data flow: 0 Outbound 1 Inbound This, along with the DESC_TYPE field, helps determine the sequence of actions to be performed by the channel and selected EUs.
31	DN	Done notification flag Setting this bit indicates whether to perform notification upon completion of this descriptor. The notification can take the form of an interrupt, modified header write back, or both depending upon the state of the INTERRUPT_ENABLE and WRITEBACK_ENABLE control bits in the crypto-channel configuration register. 0 Do not signal DONE upon completion of this descriptor (unless globally programmed to do so via the crypto-channel configuration register.) 1 Signal DONE upon completion of this descriptor **Note:** The SEC can be programmed to perform DONE notification upon completion of each descriptor, upon completion of any descriptor, or completion of the final descriptor in a chain. This bit provides for the second case. When the crypto-channel requests a write of the descriptor header back to system memory, the most significant byte (big endian) of the header is always read as set to 0xFF, and the remaining 24 bits are not changed.

Figure 3 shows the two sub fields of Op_*n*.

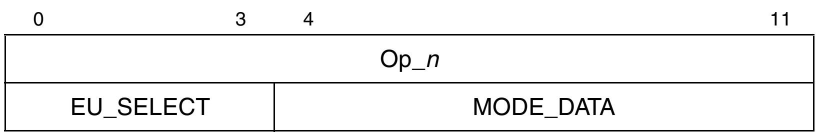

Figure 3. Op_*n* sub fields

The following rules govern the choices for these fields:

- EU_SEL0 values of “No EU selected” or “Reserved” result in an “Unrecognized Header Error” condition during processing of the descriptor header.
- The only valid choices for EU_SEL1 are “No EU selected” or MDEU. Any other choice results in an “Unrecognized Header” error condition.
- If EU_SEL1 is MDEU, then EU_SEL0 must be DEU, AESU, or AFEU. All other values of EU_SEL0 result in an “Unrecognized header” error condition.

The full range of permissible EU_Select values is shown in Table 2.

Table 2. EU_Select Values

Value	EU Select
0000	No EU selected
0001	AFEU
0010	DEU
0011	MDEU

Table 2. EU_Select Values (continued)

Value	EU Select
0100	RNG
0101	PKEU
0110	AESU
others	Reserved
1111	Reserved for header writeback

4 Execution Unit MODE_DATA

The SEC execution units are programmed via the descriptor header. The MODE_DATA portion of the Op_*n* field in the descriptor header is written to bits 56–63 of the mode register in the execution unit selected by the EU_SELECT field in Op_*n*. A complete explanation of the execution unit registers can be found in the applicable device reference manual, however, the mode register for each EU is provided in this section for convenience.

4.1 PKEU Mode Register

This register specifies the internal PKEU routine to be executed. For the root arithmetic routines, PKEU has the capability to perform arithmetic operations on subsegments of the entire memory. This is particularly useful for operations such as ECDH (elliptic curve Diffie-Hellman) key agreement computation, where the point multiplication results are converted from projective to affine by multiplying the projective result by the result of the external inverse computation. By using regAsel and regBsel, for example, parameter memory A subsegment 2 can be multiplied into parameter memory B subsegment 1. Figure 4 and Figure 5 detail two definitions.

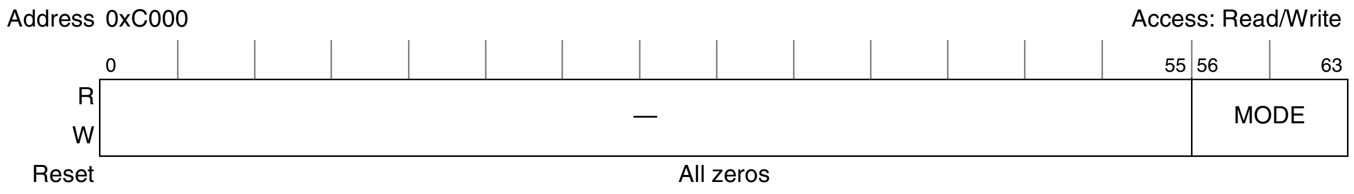

Figure 4. PKEU Mode Register: Definition 1

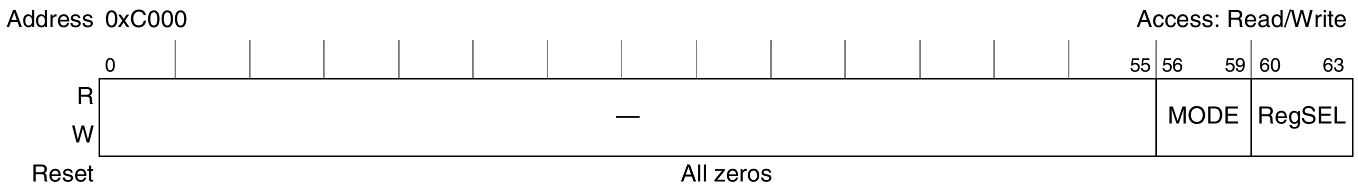

Figure 5. PKEU Mode Register: Definition 2

Table 3 lists mode register routine definitions. Parameter memories are referred to for the base address, as shown.

Table 3. Mode Register Routine Definitions

Routine	Mode [56-59]	Mode [60–61]	Mode [62–63]
Reserved	0000	00	00
Clear memory	0000	0	01
Modular exponentiation	0000	00	10
R^2 mod N	0000	00	11
R_nR_p mod N	0000	01	00
MULTKPTOQ	0000	01	01
MULTKPTOQf2M	0000	01	10
MULTKPTOQxyz	0000	01	11
MULTKPOTOQ	0000	10	00
FPADDPTOQ	0000	10	01
FPDOUBLEQ	0000	10	10
F2MADDPTOQ	0000	10	11
F2MDOUBLEQ	0000	11	00
F_2m R^2 CMD	0000	11	01
F_2m INV CMD	0000	11	10
MOD INV CMD	0000	11	11
Modular addition	0001	regAsel[1] 00 = A0 01 = A1 10 = A2 11 = A3	regBsel[1] 00 = B0 01 = B1 10 = B2 11 = B3
Modular subtraction	0010		
Modular multiplication with single reduction	0011		
Modular multiplication with double reduction	0100		
Polynomial addition	0101		
Polynomial multiplication with single reduction	0110		
Polynomial multiplication with double reduction	0111		
Single Step Exponentiation (combines R^2 mod N+ mod mult + mod exp)	1000	00	00

Note:

[1] regAsel and regBsel here refer to the specific segment of parameter memory A and B.

4.2 DEU Mode Register

Shown in Figure 6, the mode register is used to control operation of the DEU and contains 64 bits.

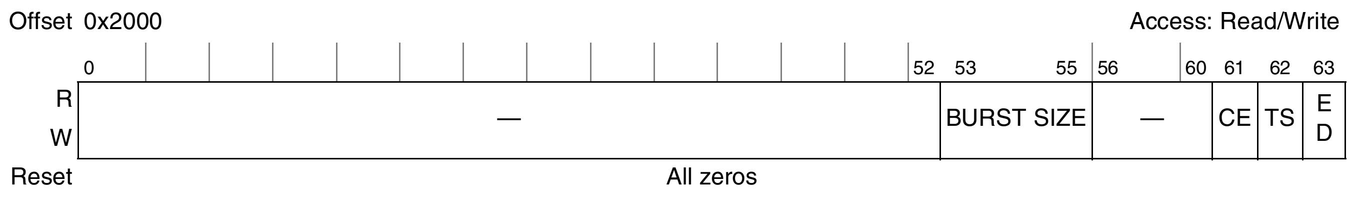

Figure 6. DEU Mode Register

Table 4 describes DEU mode register fields.

Table 4. DEU Mode Register Field Descriptions

Bits	Name	Description
0–52	—	Reserved
53–55	BURST SIZE	BURST SIZE implements flow control to allow larger than FIFO sized blocks of data to be processed with a single key/IV. The DEU signals to the channel that a “Burst Size” amount of data is available to be pushed to or pulled from the FIFO. The inclusion of this field in the DEU mode register is to avoid confusing a user who may read this register in debug mode. Burst size should not be written directly to the DEU.
56–60	—	Reserved
61	CE	CBC/ECB If set, DEU operates in cipher-block-chaining mode. If not set, DEU operates in electronic codebook mode. 0 ECB mode 1 CBC mode
62	TS	Triple/Single DES If set, DEU operates the Triple DES algorithm; if not set, DEU operates the single DES algorithm. 0 Single DES 1 Triple DES
63	ED	Encrypt/decrypt If set, DEU operates the encryption algorithm; if not set, DEU operates the decryption algorithm. 0 Perform decryption 1 Perform encryption

4.3 AFEU Mode Register

The mode register, shown in Figure 7, is used to control operation of AFEU and contains 3 bits.

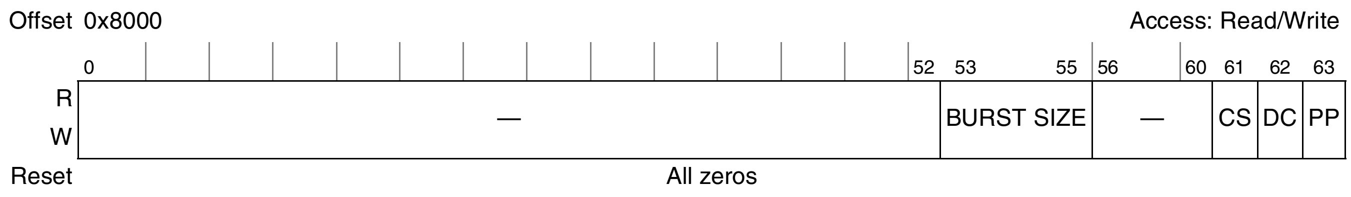

Figure 7. AFEU Mode Register

Table 5 describes AFEU mode register fields.

Table 5. AFEU Mode Register Field Descriptions

Bits	Name	Description
0–52	—	Reserved
53–55	BURST SIZE	BURST SIZE implements flow control to allow larger than FIFO sized blocks of data to be processed with a single key/context. The AFEU signals to the channel that a ‘burst size’ amount of data is available to be pushed to or pulled from the FIFO. The inclusion of this field in the AFEU Mode Register is to avoid confusing a user who may read this register in debug mode. Burst size should not be written directly to the AFEU.
56–60	—	Reserved
61	CS	Context Source If set, this causes the context to be moved from the input FIFO into the S-box prior to starting encryption/decryption. Otherwise, context should be directly written to the context registers. Context Source is only checked if the prevent permute bit is set. 0 Context not from FIFO 1 Context from input FIFO
62	DC	Dump Context If set, this causes the context to be moved from the S-box to the output FIFO following assertion AFEU’s done interrupt. 0 Do not dump context 1 After cipher, dump context
63	PP	Prevent Permute Normally, AFEU receives a key and uses that information to randomize the S-box. If reusing a context from a previous descriptor, this bit should be set to prevent AFEU from reperforming this permutation step. 0 Perform S-Box permutation 1 Do not permute

4.4 MDEU Mode Register

The MDEU mode register, displayed in Figure 8, is used to control operation of the MDEU and contains 6 bits.

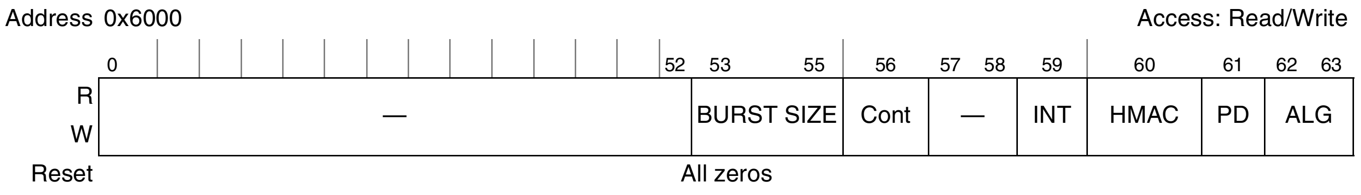

Figure 8. MDEU Mode Register

Table 6 describes MDEU mode register signals.

Table 6. MDEU Mode Register Field Descriptions

Bits	Signal	Description
0–52	—	Reserved
53–55	BURST SIZE	The implements flow control to allow larger than FIFO sized blocks of data to be processed with a single key/context. The MDEU signals to the channel that a “Burst Size” amount of data is available to be pushed to the FIFO. The inclusion of this field in the MDEU Mode Register is to avoid confusing a user who may read this register in debug mode. Burst size should not be written directly to the MDEU.
56	Cont	Continue Used during HMAC/HASH processing when the data to be hashed is spread across multiple descriptors 0 Do not continue—operate the MDEU in auto completion mode. 1 Preserve context to operate the MDEU in Continuation mode.
57–58	—	Reserved
59	INT	Initialization Bit Causes an algorithm-specific initialization of the digest registers. Most operations will require this bit to be set. Only operations that load context from a known intermediate hash value would not initialize the registers. 0 Do not initialize 1 Initialize the selected algorithm’s starting registers
60	HMAC	Identifies the hash operation to execute 0 Perform standard hash 1 Perform HMAC operation. This requires a key and key length information.
61	PD	If set, PD configures the MDEU to automatically pad partial message blocks. 0 Do not autopad 1 Perform automatic message padding whenever an incomplete message block is detected.
62–63	ALG	Message Digest algorithm selection 00 SHA-160 algorithm (full name for SHA-1) 01 SHA-256 algorithm 10 MD5 algorithm 11 Reserved

4.4.1 Recommended Settings for MDEU Mode Register

The most common task likely to be executed via the MDEU is HMAC generation. HMACs are used to provide message integrity within a number of security protocols, including IPSec, and SSL/TLS. When the HMAC is being generated by a single descriptor (the MDEU acting as sole or secondary EU), the following mode register bit settings should be used:

Table 7. Mode Register—HMAC Generated by Single Descriptor

Bits	Field	Value
0	Cont	0 (off)
3	Init	1(on)
4	HMAC	1 (on)
5	PD	1 (on)

When the HMAC is being generated for a message that is spread across a chain of descriptors, the following mode register bit settings should be used:

Table 8. Mode Register—HMAC Generated for a Message Across a Chain of Descriptors

Bits	Field	Value		
		First Descriptor	Middle Descriptor(s)	Final Descriptor
0	Cont	1 (on)	1 (on)	0 (off)
3	Init	1 (on)	0 (off)	0 (off)
4	HMAC	1 (on)	0 (off)	1 (on)
5	PD	0 (off)	0 (off)	1 (on)

All descriptors other than the final descriptor must output the intermediate message digest for the following descriptor to reload as MDEU context.

4.5 RNG Mode Register

The RNG mode register is used to control the RNG. One operational mode, randomizing, is defined. Writing any other value than 0 to 56:63 results in a data error interrupt that's reflected in the RNG Interrupt Status Register. The mode register also reflects the value of burst size, which is loaded by the crypto-channel during normal operation with the SEC as an initiator. Burst size is not relevant to target mode operations, where the CPU pushes and pulls data from the execution units.

The mode register is cleared when the RNG is reset or re-initialized and is shown in Figure 9.

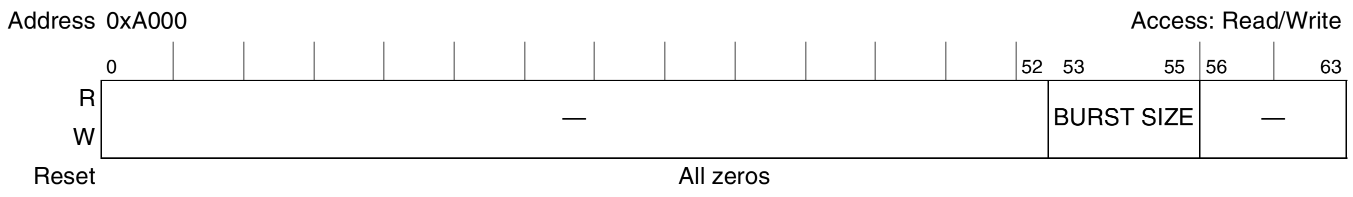

Figure 9. RNG Mode Register

4.5.1 AESU Mode Register

The AESU mode register, shown in Figure 10, contains 7 bits that are used to program the AESU. It also reflects the value of burst size, which is loaded by the crypto-channel during normal operation with the SEC as an initiator. Burst size is not relevant to target mode operations, where the CPU pushes and pulls data from the execution units.

The mode register is cleared when the AESU is reset or re-initialized. Setting a reserved mode bit will generate a data error. If the mode register is modified during processing, a context error will be generated.

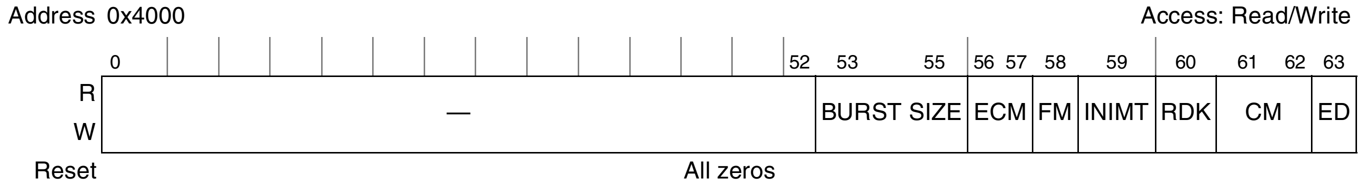

Figure 10. AESU Mode Register

Table 9 describes AESU mode register signals.

Table 9. AESU Mode Register Signals

Bits	Signal	Description
0–52	—	Reserved
53–55	BURST SIZE	The implements flow control to allow larger than FIFO sized blocks of data to be processed with a single key/context. The AESU signals to the channel that a “Burst Size” amount of data is available to be pushed to the FIFO. The inclusion of this field in the AESU Mode Register is to avoid confusing a user who may read this register in debug mode. Burst size should not be written directly to the AESU.
56–57	ECM	Extend cipher mode: Used in combination with CM to define the mode of AES operation. See Table 10 for mode bit combinations.
58	FM	Final MAC (FM) Processes final message block and generates final MAC tag at end of message processing (CCM mode only) 0 Do not generate final MAC tag 1 Generate final MAC tag after CCM processing is complete.
59	IM	Initialize MAC(IM) Initializes AESU for new message (CCM mode only) 0 Do not initialize (context will be loaded by CPU) 1 Initialize new message with nonce
60	RDK	Restore Decrypt Key (RDK) Specifies that key data write will contain pre-expanded key (decrypt mode only). See note on use of RDK bit. 0 Expand the user key prior to decrypting the first block 1 Do not expand the key. The expanded decryption key will be written following the context switch.
61–62	CM	Cipher mode Used in combination with ECM to define the mode of AESU operation. See Table 10 for mode bit combinations.
63	ED	Encrypt/Decrypt If set, AESU operates the encryption algorithm; if not set, AESU operates the decryption algorithm. This bit is ignored if CM is set to “11” - CTR Mode. 0 Perform decryption 1 Perform encryption

NOTE: Restore Decrypt Key (RDK)

In most networking applications, the decryption of an AES-protected packet will be performed as a single operation. However, if circumstances dictate that the decryption of a message should be split across multiple descriptors, the AESU allows the user to save the decrypt key, and the active AES context, to memory for later re-use. This saves the internal AESU processing overhead associated with regenerating the decryption key schedule (approximately 12 AESU clock cycles for the first block of data to be decrypted).

The use of RDK is completely optional, as the input time of the preserved decrypt key may exceed the 12 cycles required to restore the decrypt key for processing the first block.

To use RDK, the following procedure is recommended:

The descriptor type used in decryption of the first portion of the message is 0100 (aesu_key_expand_output). The AESU mode must be Decrypt. See the “SEC Lite Descriptors” chapter in the *MPC885 PowerQUICC Family Reference Manual* for more information. The descriptor will cause the SEC Lite to write the contents of the context registers and the key registers (containing the expanded decrypt key) to memory.

To process the remainder of the message, use a normal descriptor type (descriptor type selected based on need for simultaneous HMAC generation, and so forth) and set the restore decyrpt key mode bit. Load the context registers and the expanded decrypt key with the previously saved key and context data from the first message. The key size is written as before (16, 24, or 32 bytes).

Table 10. AES Cipher Modes

Mode	ECM	CM
ECB	00	00
CBC	00	01
Res	xx	10
CTR	00	11
SRT[1]	01	11
CCM	10	00

5 Selecting Descriptor Type—DESC_TYPE

The SEC 2.0 accepts 16 fixed-format descriptors. The Desc_Type field in the descriptor header advises the crypto-channel of the predetermined ordering of keys, context, and null fields. The ordering of inputs and outputs in the length/pointer pairs (as defined by Desc_Type) is shown in Table 14.

Table 11 shows the permissible values for the Desc_Type field in the descriptor header. Note: When compared to the SEC 1.0 descriptor types (SEC 1.0 is the integrated security core found in the Motorola MPC8272), fewer descriptor types are listed. Many of the permissible SEC 1.0 descriptor types aren't operationally useful, and exist for test and debug purposes. The SEC 2.0 eliminates many of the descriptor types which were not operationally useful, and adds several new types which offer optimized processing for specific security protocols. Descriptor types from the SEC 1.0, which have "0" in the last bit, are listed first, followed by new SEC 2.0 types, which have "1" in the last bit.

NOTE

SEC 1.0 descriptors do not run unmodified on the SEC 2.0, due to a formatting change. SEC 2.0 descriptors which have a "0" in the last bit use the same inputs are SEC 1.0 descriptors, and are easily mapped to SEC 2.0 descriptors. The SEC 2.0 descriptor format allows SEC 2.0 descriptors to support scatter/gather, and don't require the use of the "Next Descriptor Pointer" field to process descriptor chains. Chaining is automatic via the channel Fetch FIFOs.

Table 11. Descriptor Types

Value (Binary)	Descriptor Type	Notes
0000_0	aesu_ctr_nonsnoop	AESU CTR non-snooping [1]
0001_0	common_nonsnoop_no_afeu	Common, nonsnooping, non-PKEU, non-AFEU [1]
0010_0	hmac_snoop_no_afeu	Snooping, HMAC, non-AFEU [2]
0011_0	—	Reserved
0100_0	—	Reserved
0101_0	common_nonsnoop_afeu	Common, nonsnooping, AFEU
0110_ 0	—	Reserved
0000_0	aesu_ctr_nonsnoop	AESU CTR nonsnooping
0001_0	common_nonsnoop	Common, nonsnooping, non-PKEU, non-AFEU
0010_0	hmac_snoop_no_afeu	Snooping, HMAC, non-AFEU
0011_0	—	Reserved
0100_0	—	Reserved
0101_0	common_nonsnoop_afeu	Common, nonsnooping, AFEU
0110_ 0	—	Reserved
0111_0	—	Reserved
1000_0	pkeu_mm	PKEU-MM
1001_0	—	Reserved
1010_0	—	Reserved
1011_ 0	—	Reserved
1100_0	hmac_snoop_aesu_ctr	AESU CTR hmac snooping [2]

Table 11. Descriptor Types (continued)

Value (Binary)	Descriptor Type	Notes
1101_0	—	Reserved
1110_0	—	Reserved
1111_0	—	Reserved
0000_1	ipsec_esp	IPsec ESP mode encryption and hashing
0001_1	802.11i AES ccmp	CCMP encryption and hashing, suitable for 802.11i
0010_1	srtp	SRTP encryption and hashing
0011_1	pkeu_assemble	pkeu_assemble Elliptic Curve Cryptography
0100_1	pkeu_ptmul	pkeu_ptmul Elliptic Curve Cryptography
0101_1	pkeu_ptadd_dbl	pkeu_ptadd_dbl Elliptic Curve Cryptography
others	—	Reserved

Note:

1. Type 0000_0 is for AES-CTR operations. Type 0001_0 also supports AES-CTR, however to use AES-CTR with 0001_0, the user must prepend zeros to the AES-Ctx before loading the AES context registers.
2. Type 1100_0 is for AES-CTR operations with HMAC. Type 0010_0 also supports AES-CTR with HMAC, however to use AES-CTR with 0010_0, the user must prepend zeros to the AES-Ctx before loading the AES context registers.

6 Direction Bit

As mentioned in Table 1, bit 30 advises the channel of the ‘direction’ of communication applicable to a given descriptor operation. The path data must take between the symmetric EU and the hashing EU depends on whether the data is being prepared for transmission, or has just been received. The Direction Bit is used to control the type of “snooping” which must occur between the primary and secondary EU. The rationale for “in-snooping” vs. “out-snooping” is found in security protocols which perform both encryption and integrity checking, such IPSec. Upon transmission of an IPSec ESP packet, the “encapsulator” must encrypt the packet payload, then calculate an HMAC over the header plus encrypted payload. Because the MDEU cannot generate the HMAC without the output of the primary EU (the one performing encryption, typically the DEU or AESU), the MDEU must “out-snoop”.

Upon receiving an IPSec packet, the “decapsulator” must calculate the HMAC over the encrypted portion of the packet prior to decryption. This allows the MDEU to source its data from the input FIFO of the primary EU, without waiting for the primary EU to finish its task.

Note that slightly different portions of an IPSec packet would pass through the primary and secondary EUs, in both the “In-Snooping” and “Out-Snooping” cases. These off-sets are dealt with by providing different starting pointers and byte lengths to the channel in the body of the descriptor.

Figure 11 provides an overview of the snooping concept.

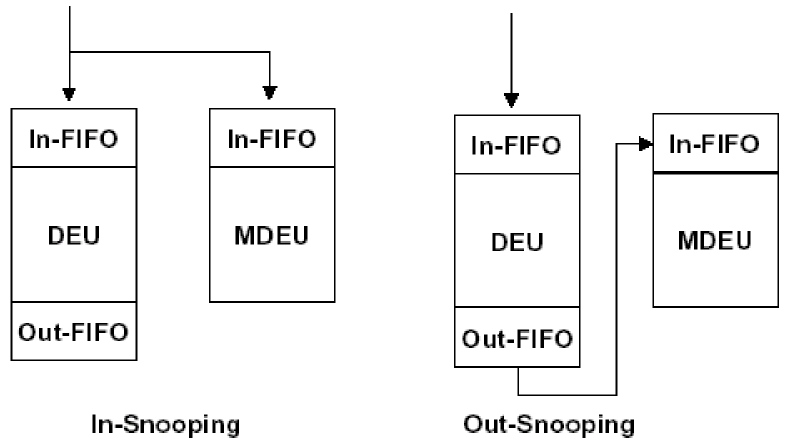

Figure 11. Snooping

7 Notification Bit

The Notification Bit in the SEC descriptor header can be set to cause the channel to signal DONE upon completion of any descriptor with the Notification Bit set. This provides an alternative to signalling DONE upon completion of each descriptor, as would occur if the Notification Type (bit 61 in the CCCR) is set to Global. The “DONE” notification can take the form of an interrupt or modified header write back or both depending upon the state of the INTERRUPT_ENABLE and WRITEBACK_ENABLE control bits in Crypto Channel Configuration Register.

When the channel signals DONE via header writeback, the most significant byte of the original header (at its original location in system memory) will always read as set to 0xFF, and the remaining 56 bits will not be modified.

8 Descriptor Format: Pointer Dwords

The descriptor contains seven “pointer dwords” which define where in memory the SEC should access its input and output data parcels. The channel determines how it will use each of the pointer dwords based on the “Descriptor Type” and “Direction” fields in the header. The channel accesses the first data parcel by starting at a location given by a POINTER value, and accessing a number of bytes given by a LENGTH or EXTENT value. EXTENT, which is short for ‘Extension’ is also a length field, and this field is used to as an offset to trigger changes in data processing when multiple operations are performed by the same descriptor. A typical use of EXTENT is to allow a single pointer to point to a range of data (data parcel) to be hashed and encrypted, with the LENGTH field providing the total data length, and the EXTENT length indicating how many bytes should be hashed without encryption, as is often required for headers in security protocol packets. EXTENT can also be used to output integrity check values, which are typically appended to the end of the encrypted and hashed data range. Although the EXTENT field exists in each

Pointer DWORD of the SEC descriptor, current usage is limited to Pointer DWORDs 4–6. Figure 12 shows the pointer Dword.

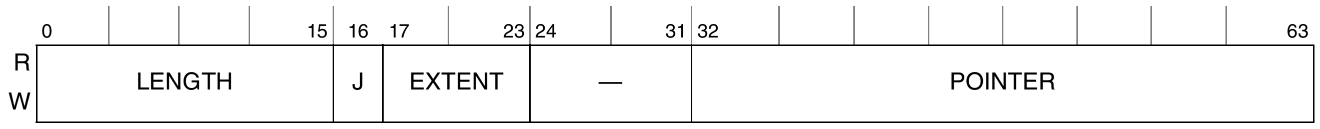

Figure 12. Pointer Dword

Table 12. Pointer DWORD Field Definitions

Bits	Name	Description
0–15	LENGTH	Length A number of bytes in the range 0 to 65535. The use of this field depends on the “Descriptor Type” and “Direction” in the header dword. A value of zero causes the channel to skip this dword.
16	J	Jump Determines whether to “jump” to a link table whenever the POINTER field in this same lword is used. 0 The POINTER field points to data. 1 The POINTER field points to a link table, and scatter/gather is enabled.
17–23	EXTENT	Extent A number of bytes in the range 0 to 127. The use of this field depends on the “Descriptor Type” and “Direction” in the header dword.
24–31	—	Reserved
32–63	POINTER	Pointer A memory address.

On occasion, a descriptor field may not be applicable to the requested service. With seven length/pointer pairs, it is possible that not all descriptor fields will be required to load the required keys, context, and text-data. (Some operations, for example, don’t require context.) Therefore, when processing descriptors, the SEC will skip entirely any pointer that has an associated LENGTH of zero.

If the channel procedure calls for reading a parcel using a nonzero LENGTH field, but the POINTER field is zero, the length value is written to the EU but no data parcel is fetched from the bus.

The J bit in each pointer dword is used to enable the scatter/gather feature. If a data parcel to be read or written by SEC is in one contiguous block of memory locations, then the scatter/gather feature is not needed. In this case the POINTER should be set to point directly at the first byte of the parcel, and the J bit should be 0. On the other hand, if the data parcel is stored in several separate segments of memory, then the scatter/gather capability is needed to assemble or distribute the complete parcel. In this case the POINTER should be set to point to a “link table”, and the J bit should be 1. For link table format, see Section 8.1, “Link Table Format.”

8.1 Link Table Format

Link tables implement scatter/gather capability. For “gather” operations, a link table specifies a list of “memory segments” that are to be concatenated in the process of assembling data parcels. For “scatter” operations, a link table specifies a list of memory segments into which the output data should be written.

Scatter or gather of a data parcel may be specified by a single link table or by a chain of link tables that are linked together with pointers (see Figure 14).

The link table or chain of link tables accessed through some descriptor POINTER must specify enough memory segments to hold *all* the data that will be accessed through that pointer. In most cases, only a single data parcel is accessed through a given POINTER, and the chain of link tables specifies just that parcel. In other cases, the descriptor POINTER is used multiple times to access a sequence of data parcels, and the chain of link tables must supply data for the entire sequence. In such cases, the end of each parcel must also be at the end of a memory segment. In other words, a single memory segment must not straddle two data parcels. An example of proper construction of link tables is illustrated in Figure 13.

A link table may contain any number of long word entries. There are two kinds of entries, “regular” entries and “next” entries. Each “regular” entry specifies a memory segment by means of a 36-bit starting address (SegAdr) and a 16-bit length (SegLen). A “next” entry is used at the end of a link table to specify that the list of memory segments is continued in another link table. In a “next” entry, the N bit is set and the SegAdr field gives the address of the next link table. A chain of link tables may contain any number of link tables.

Whether the list of memory segments is in a single link table or split into several link tables, the last entry in the last link table is a “regular” entry with the R (return) bit set. The R bit signifies the end of link table operations so that the channel returns to the descriptor for its next pointer (if any). Link tables are illustrated in Figure 13.

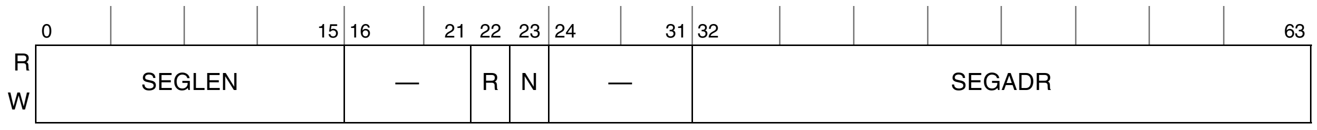

Figure 13. Link Table Entry Format

Table 13. Link Table Field Definitions

Name	Description
SEGLEN	Length: When N = 0, a number in the range 1 to 65535, specifying the number of bytes in the memory segment. pointed to by SEGADR. A value of 0 will cause an error bit to be set in the Channel Pointer Status Register—GER for a gather operation or SER for a scatter operation. When N = 1, ignored.
—	Reserved
R	Return: When N = 0: 0 No special action. 1 This is the last entry in the chain of link tables. If this entry does not specify the right number of bytes to complete the last data parcel, a GER or SER error will be set in the Channel Pointer Status Register. When N = 1, ignored.
N	Next: 0 No special action. 1 This is the last long word in the current link table. The SEGADR field is the address of the next link table in the chain.

Table 13. Link Table Field Definitions (continued)

Name	Description
—	Reserved
SEGADR	Segment address A memory address.

Figure 14 illustrates various ways that a descriptor may specify data parcels:

The first pointer dword in the descriptor specifies Parcel A using the simplest method—the parcel is specified directly through Pointer0 and Length0.

The next pointer dword uses a chain of link tables to specify Parcel B. Since J=1, Pointer1 is used as the address of a link table. The link table specifies several "regular" entries specifying data segments to be concatenated. The last word of the link table is a "next" entry indicating that the list continues in the next link table. The last entry in the last link table of the chain has the R bit set.

The last cases illustrate how one pointer in a descriptor can be used to specify multiple parcels. Pointer2 and Length2 specify Parcel C, then Parcel D follows immediately afterwards, with length specified by Extent2. Pointer3 is used for three data parcels (E, F, and G), this time using link tables.

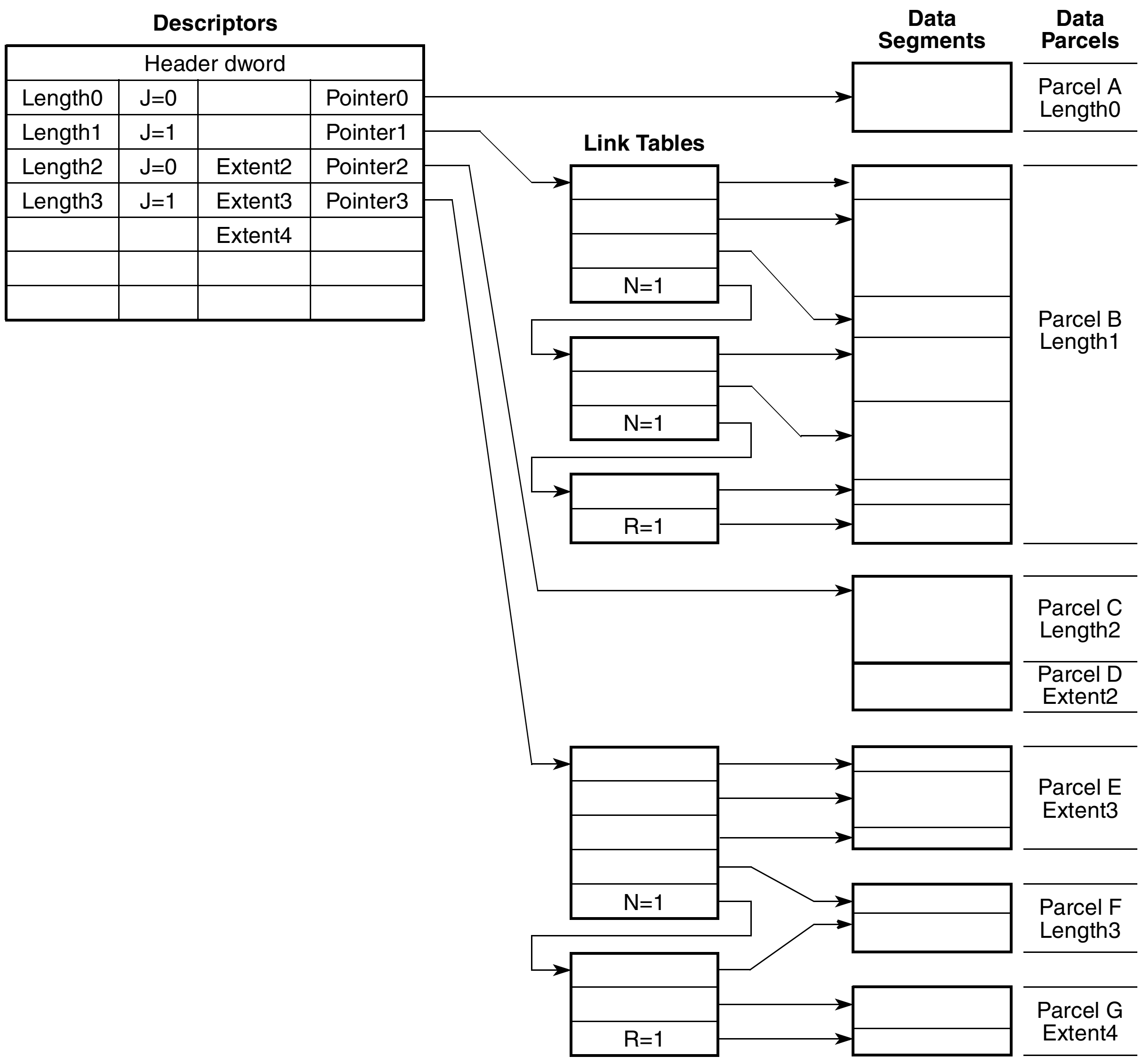

Figure 14. Descriptors, Link Tables, and Data Parcels

For any sequence of data parcels accessed by a link table or chain of link tables, the combined lengths of the parcels (the sum of their LENGTH and/or EXTENT fields) must equal the combined lengths of the link table memory segments (SEGLEN fields). Otherwise the channel sets the appropriate error bit in the Channel Pointer Status Register—GER for gather error or SER for scatter error (see Section 1.6.1.2 of the SEC 2.0 Chapter in the MPC85xx User's Manual).

Example (from Figure 14): To demonstrate use of a link table, assume that the current descriptor type calls for the channel to access a data parcel using Pointer3 and Extent3 fields, and assume that J3=1. Due to the J3 value, Pointer3 is not used as a data address but instead used as the address of a link table. The channel begins by reading the first four long words starting at Pointer3 into an internal "link table buffer".

Using the first entry of the link table, the channel starts accessing the data parcel by reading SEGLEN bytes beginning at SEGADR. If the required data parcel size (Extent3) is greater than this first SegLen, the channel moves on to the next entry of the link table, and reads SEGLEN bytes starting at SEGADR. While there are more bytes to be read in the data parcel, this process continues. If the channel's link table buffer is exhausted, the channel reads the next four long words of the link table into its link table buffer. If a link table entry is encountered in which the N bit is set, the channel uses the SEGADR field in that word to find the next link table in the chain. The last byte of the required parcel size (Extent3) must coincide with the last byte of a memory segment, or unpredictable results may occur.

Now assume that the channel accesses its next data parcel using Pointer3 again, this time with length given by Length3. In this case the channel continues to the next line of the link table, and begins reading the memory segment specified there. As before, the channel concatenates memory segments from as many link table entries as necessary to obtain the required number of bytes (Length3).

Similarly, the next data parcel is obtained by using Pointer3 yet again, this time with length given by Extent4.

Assume that for the current descriptor type, the Extent4 data parcel is the last one to be accessed through Pointer3. Then the link table entry that supplies the last memory segment for Extent4 has the R bit set, signifying that this is the last entry in the chain of link tables.

8.2 Pointer DWORD Format by Descriptor Type

Table 14 shows how the length/pointer pairs should be used with the various descriptor types to load keys, context, and data into the execution units, and how the required outputs should be unloaded. Note: Some outputs are optional.

Table 14. Descriptor Length/Pointer Mapping

Descriptor Type	Field Type	Pointer Dword0	Pointer Dword1	Pointer Dword2	Pointer Dword3	Pointer Dword4	Pointer Dword5	Pointer Dword6
0000_0 aesu_ctr_ nosnoop	**Length**	Reserved	Primary EU Context In	Confidentiality Key	Data In	Data Out	Primary EU Context Out	Reserved
	Extent	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
0001_0 common_ nosnoop for Confidentiality algorithms, AES-CCM, and RNG	**Length**	Reserved	Primary EU Context In	Confidentiality Key	Data In	Data Out	Primary EU Context Out	Reserved
	Extent	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
0001_0 common_ nosnoop for Hash and HMAC	**Length**	Reserved	Primary EU Context In	Integrity Key	Data In	Reserved	CICV (or Hash) Out	Reserved
	Extent	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved

Table 14. Descriptor Length/Pointer Mapping (continued)

Descriptor Type	Field Type	Pointer Dword0	Pointer Dword1	Pointer Dword2	Pointer Dword3	Pointer Dword4	Pointer Dword5	Pointer Dword6
0001_0 common_ nosnoop for KEU f9, STEU f9, AES-XCBC, AES-CMAC	**Length**	Reserved	Primary EU Context In	Integrity Key	Data In	Reserved	Primary EU Context Out	CICV Out
	Extent	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
0010_0 hmac_snoop _no_afeu	**Length**	Integrity Key	Integrity Only Data	Confidentiality Key	Primary EU Context In	Data In	Data Out	CICV Out
	Extent	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
0101_0 common_ nosnoop_ afeu	**Length**	Reserved	Context In (via In FIFO)	Confidentiality Key	Data In	Data Out	Context Out (via Out FIFO)	Reserved
	Extent	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
1000_0 pkeu_mm	**Length**	“N” In	“B” In	“A” In	“E” In	“B” Out	Reserved	Reserved
	Extent	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
1100_0 hmac_snoop _ aesu_ctr	**Length**	Integrity Key	Integrity Only Data	Confidentiality Key	Primary EU Context In	Data In	Data Out	CICV Out
	Extent	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
0000_1 ipsec_esp	**Length**	Integrity Key	Integrity Only Data	Primary EU Context In	Confidentialit y Key	Data In	Data Out	Primary EU Context Out
	Extent	Reserved	Reserved	Reserved	Reserved	Reserved	CICV Out	Reserved
0001_1 802.11i AES ccmp	**Length**	Reserved	Primary EU Context In	Confidentiality Key	Integrity Only Data	Data In	Data Out	Primary EU Context Out
	Extent	Reserved	Reserved	Reserved	Reserved	Reserved	CICV Out	Reserved
0010_1 srtp	**Length**	Integrity Key	Primary EU Context In	Confidentiality Key	Data In	Data Out	CICV Out	Primary EU Context Out
	Extent	Reserved	Reserved	Reserved	Integrity Only Data	Integrity Only Trailer	Reserved	Reserved
0011_1 pkeu_build	**Length**	“A0” In	“A1” In	“A2” In	“A3” In	“B0” In	“B1” In	“Build” Out
	Extent	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
0100_1 pkeu_ptmul	**Length**	“N” In	“E” In	“Build” In	“B1” Out	“B2” Out	“B3” Out	Reserved
	Extent	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved

Table 14. Descriptor Length/Pointer Mapping (continued)

Descriptor Type	Field Type	Pointer Dword0	Pointer Dword1	Pointer Dword2	Pointer Dword3	Pointer Dword4	Pointer Dword5	Pointer Dword6
0101_1	**Length**	“N” In	“Build” In	“B2” In	“B3” In	“B1” Out	“B2” Out	“B3” Out
pkeu_ptadd_db	**Extent**	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
Others		Reserved						

8.2.1 Null Fields

On occasion, a descriptor field may not be applicable to the requested service. With seven length/pointer pairs, it is possible that not all descriptor fields will be required to load the required keys, context, and data. (Some operations do not require context, others may only need to fetch a small, contiguous block of data.) Therefore, when processing data packet descriptors, the SEC will skip entirely any pointer that has an associated length of zero.

9 Use of Specific Descriptor Types

The remainder of this document describes in greater detail how specific descriptor types should be used to accelerate various common cryptographic operations and security protocols. The data associated with any given descriptor could be located in non-contiguous memory, a situation which could be managed via the scatter/gather capabilities of the SEC 2.0 descriptors. The purpose of the examples below are to illustrate the use of specific descriptor header and the definition of the associated inputs and outputs, and to reduce the complexity of the explanation, scatter/gather is never invoked.

9.1 Descriptor Type 0001_0

Descriptor Type 0001_0 is used for a wide variety of functions, most of which do not require all the Pointer DWORDS to be used. A few “non-obvious” uses of this descriptor type are highlighted in Table 15.

Table 15. Descriptor Type 0001_0 Length/Pointer Mapping

Descriptor Type	Field Type	Pointer Dword0	Pointer Dword1	Pointer Dword2	Pointer Dword3	Pointer Dword4	Pointer Dword5	Pointer Dword6	Use
0001_0 common_ nosnoop	Length	nil	nil	nil	nil	Data Out	nil	nil	RNG Only
	Extent	undefined	undefined	undefined	nil	nil	nil	undefined	
0001_0 common_ nosnoop	Length	nil	MDEU Ctx-In (opt)	nil	Data In	nil	Hash out	nil	Hash Only
	Extent	undefined	undefined	undefined	nil	nil	nil	undefined	
0001_0 common_ nosnoop	Length	nil	MDEU Ctx-In (opt)	HMAC Key	Data In	nil	HMAC out	nil	HMAC Only
	Extent	undefined	undefined	undefined	nil	nil	nil	undefined	

Table 15. Descriptor Type 0001_0 Length/Pointer Mapping (continued)

Descriptor Type	Field Type	Pointer Dword0	Pointer Dword1	Pointer Dword2	Pointer Dword3	Pointer Dword4	Pointer Dword5	Pointer Dword6	Use
0001_0 common_ nosnoop	Length	Reserved	Primary EU Context In	Confidentiality Key	Data In	Data Out	Primary EU Context Out	Reserved	Self Integrity Checking operations
	Extent	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	

For RNG operations, there is no key, context, or data to send into the SEC, so only a single length/pointer pair is needed to cause the random data to be written out.

For hash only operations, the length and location of the data to be hashed is designated with Pointer DWORD 4, with Pointer DWORD 6 used to designate the length and location of the hash value to be output. There is no requirement for the SEC 2.0 to write out the full length hash as defined by the hash algorithm. Pointer DWORD 2 is used if the requested hash operation is a continuation of a hash operation from a previous descriptor. The MDEU Context In in this case would be the intermediate hash, which would have been written out using Pointer DWORD 6 in a previous hash only descriptor.

HMAC only operations are similar to hash only operations, however Pointer DWORD 3 is used to load the HMAC key. The HMAC itself is written out via Pointer DWORD 6. If an HMAC calculation is spread across multiple descriptors, all descriptors after the first would need to load the MDEU Context registers via Pointer DWORD 2. This requires the first descriptor to output the MDEU context, or Message Digest, rather than an HMAC, with Pointer DWORD 6.

Certain protocols don't rely on the HMAC function provided by the MDEU to generate MACs, or message integrity check values. The only supported case of this is the SEC 2.0 is AES-CCM mode, in which the AESU can both encrypt and integrity check data in a single pass, without snooping data to another EU. Table 17 provides an example of this method.

9.2 Descriptor Type 0001_0 Examples

The descriptor shown in Table 16 uses Descriptor Type 0001_0 to completely sets up the DEU for an encryption operation; loads the keys, context, and data; writes the permuted data back to memory; and (optionally) writes the altered context (IV) back to memory. (This may be necessary when DES is operating in CBC mode.) Upon completion of the descriptor, the DEU is automatically cleared and released. A test descriptor for this operation is provided in Section A.1, "3DES_CBC_ENC," of the Appendix.

Table 16. Representative Descriptor DPD_Type 0001_0_3DES_CBC_Encrypt

Field	Value/Type	Description
Header	0x2070_0010	DPD_Type 0001_0_3DES_CBC_Encrypt
Reserved	0x0000_0000	—
Length 0	Length	Nill

Table 16. Representative Descriptor DPD_Type 0001_0_3DES_CBC_Encrypt (continued)

Field	Value/Type	Description
Pointer 0	Pointer	Nill
Length 1	Length	Number of Bytes of IV to be written to DEU IV register (always 8)
Pointer 1	Pointer	Address of IV
Length 2	Length	Number of Bytes of Key to be written to DEU Key register (must be 16 or 24)
Pointer 2	Pointer	Address of Key
Length 3	Length	Number of bytes to be ciphered
Pointer 3	Pointer	Address of data to be ciphered
Length 4	Length	Bytes to be written (should be equal to Length of Data-in)
Pointer 4	Pointer	Address where ciphered data is to be written
Length 5	Length	(Optional) Number of Bytes of IV to be written to memory space (always 8)
Pointer 5	Pointer	(Optional) Address where IV is to be written
Length 6	Length	Nill
Pointer 6	Pointer	Nill

9.2.1 Descriptor Type 0001_0 Additional Examples

The descriptor shown in Table 17 uses Descriptor Type 0001_0 to completely sets up the AESU for an encryption operation in CCM mode; loads the keys, context, and data; writes the permuted data back to memory; and writes the altered context (IV) back to memory. This altered context include the MIC, which software can append to the 802.11i frame, or use to verify the received MIC. This is the original SEC 1.0 method for performing AES-CCM operations. Existing SEC 1.0 descriptors of type 0001 can be converted in a straightforward manner to SEC 2.0 descriptors of type 0001_0. A test descriptor for this operation is provided in Section A.2 of the Appendix.

Table 17. Representative Descriptor DPD_Type 0001_0_AES-CCM_Encrypt

Field	Value/Type	Description
Header	0x6B10_0010	DPD_Type 0001_0_AES_CCM_Encrypt
Reserved	0x0000_0000	—
Length 0	Length	Nill
Pointer 0	Pointer	Nill
Length 1	Length	Number of Bytes of AES Context to be written to AESU Context registers (always 56)
Pointer 1	Pointer	Address of Context
Length 2	Length	Number of Bytes of Key to be written to AESU Key register (must be 16)
Pointer 2	Pointer	Address of Key
Length 3	Length	Number of bytes to be ciphered
Pointer 3	Pointer	Address of data to be ciphered

Table 17. Representative Descriptor DPD_Type 0001_0_AES-CCM_Encrypt (continued)

Field	Value/Type	Description
Length 4	Length	Bytes to be written back to memory (should be equal to Length of Data-in)
Pointer 4	Pointer	Address where ciphered data is to be written
Length 5	Length	Number of Bytes of AES Context to be written to memory (always 56)
Pointer 5	Pointer	Address where AES Context is to be written
Length 6	Length	Nill
Pointer 6	Pointer	Nill

9.2.2 HMAC-MD-5 (In-Bound/Out-Bound IPSec AH)

The descriptor shown in Table 18 uses Descriptor Type 0001_0 to completely sets up the MDEU for an HMAC operation using the MD-5 algorithm. The descriptor loads the HMAC keys, and data to be HMAC'd, then writes the calculated HMAC value back to memory. This is the original SEC 1.0 method for performing HMAC only operations. Existing SEC 1.0 descriptors of type 0001 can be converted in a straightforward manner to SEC 2.0 descriptors of type 0001_0. A test descriptor for this operation is provided in Section A.3 of the Appendix.

Table 18. Representative Descriptor DPD_Type 0001_0_HMAC-MD-5

Field	Value/Type	Description
Header	0x31E0_0010	DPD_Type 0001_0_HMAC_MD-5
Reserved	0x0000_0000	Pointer to Next Descriptor
Length 0	Length	Nill
Pointer 0	Pointer	Nill
Length 1	Length	Nill
Pointer 1	Pointer	Nill
Length 2	Length	Number of bytes of HMAC key to be written to MDEU Key register
Pointer 2	Pointer	Address of HMAC key
Length 3	Length	Number of bytes of data to be written to MDEU Input FIFO
Pointer 3	Pointer	Address of data
Length 4	Length	Nill
Pointer 4	Pointer	Nill
Length 5	Length	Number of bytes of HMAC to be written out to memory (always 16 MD-5)
Pointer 5	Pointer	Address where HMAC is to be written
Length 6	Length	Nill
Pointer 6	Pointer	Nill

The descriptor header encodes the information required to select the MDEU for Op_0, and no EU for Op_1. The Op_0 Mode Data configured the MDEU to operate in HMAC-MD-5 mode. Because all the data necessary to calculate the HMAC in a single descriptor is available, Initialize, and Autopad are set, while Continue is off.

The descriptor header also encodes the descriptor type 0001, which defines the input and output ordering for “common_nonsnoop_no_afeu”. This is the descriptor type used for most operations which don’t require a secondary EU. Following some null pointers, the HMAC key is loaded, followed by the length and pointer to the data over which the HMAC will be calculated.

The data is brought into the MDEU input FIFO, and when the final byte of data to be HMAC’d has been processed through the MDEU, the descriptor will cause the MDEU to write the HMAC to the indicated area in memory. The SEC will write the entire 16 bytes HMAC-MD-5 to memory, and depending on whether the packet is in-bound or out-bound, the CPU will either insert the most significant 12 bytes of the HMAC generated by the SEC into the packet header (out-bound) or compare the HMAC generated by the SEC with the HMAC which was received with the in-bound packet (obviously in-bound). If the HMACs match, the packet integrity check passes.

9.3 Descriptor Type 0010_0 Example

The descriptor shown in Table 19 uses Descriptor Type 0010_0 to completely sets up the DEU and MDEU for a decryption and HMAC verification operation equivalent to IPSec ESP mode. The descriptor loads the keys, context, and data; writes the permuted data back to memory; as well as the calculated HMAC for comparison by the CPU. This is the original SEC 1.0 method for performing IPSec operations. Existing SEC 1.0 descriptors of type 0010 can be converted in a straightforward manner to SEC 2.0 descriptors of type 0010_0. A test descriptor for this operation is provided in Section A.4, “3DES_HMAC_SHA_1.”

Table 19. Representative Descriptor DPD_Type 0010_0_3DES-HMAC-SHA-1 Decrypt

Field	Value/Type	Description
Header	0x2063_1C22	DPD_Type 0010_0_3DES_CBC_HMAC_SHA-1 Decrypt
Reserved	0x0000_0000	—
Length 0	Length	Number of bytes of HMAC Key to be written to MDEU Key register
Pointer 0	Pointer	Address of HMAC Key
Length 1	Length	Number of bytes to be HMAC’d but not ciphered
Pointer 1	Pointer	Address of data to be HMAC’d
Length 2	Length	Number of bytes of key to be written to DEU Key register (must be 16 or 24)
Pointer 2	Pointer	Address of key
Length 3	Length	Number of bytes of IV to be written to DEU IV register (always 8)
Pointer 3	Pointer	Address of IV
Length 4	Length	Number of bytes of ciphertext to be decrypted **Note:** For this descriptor type, the MDEU will also process this data, so that the total data processed through the HMAC function will be Length 2 + Length 5

Table 19. Representative Descriptor DPD_Type 0010_0_3DES-HMAC-SHA-1 Decrypt (continued)

Field	Value/Type	Description
Pointer 4	Pointer	Address of ciphertext to be decrypted **Note:** This address must be the first address after Pointer 2 + Length 2, so that data being ciphered is contiguous to the data being HMAC'd only.
Length 5	Length	Number of bytes of plaintext to be written out to memory (should be equal to Length 5)
Pointer 5	Pointer	Address where plaintext is to be written
Length 6	Length	Number of bytes of HMAC to be written to memory **Note:** Cannot be greater than the number of bytes produced by the HMAC algorithm (16 for MD-5, 20 for SHA-1). May be smaller than algorithm size, ie, 12 bytes to match length of IPSec Authentication Data
Pointer 6	Pointer	Address where HMAC is to be written

The descriptor header encodes the information required to select the DEU for Op_0, and the MDEU for Op_1. The Op_0 Mode Data configured the DEU to operate in 3DES, CBC, Decrypt mode. The Op_1 Mode Data configured the MDEU to operate in HMAC-SHA-1 mode. Because all the data necessary to calculate the HMAC in a single dynamic descriptor is available, Initialize, and Autopad are set, while Continue is off.

The descriptor header also encodes the descriptor type 0010_0, which defines the input and output ordering for "hmac_snoop_no_afeu". The HMAC key is loaded first, followed by the length and pointer to the data over which the HMAC will be calculated. The 3DES key is loaded next, followed by the 3DES IV. The data to be decrypted and HMAC'd is only brought into the SEC a single time, with the DEU and MDEU selectively reading the portions of the data stream corresponding to their data of interest.

Ciphertext is brought into the DEU input FIFO, with the MDEU "in-snooping" the portion of the data it has been told to process. As the decryption continues, the plaintext fills the DEU output FIFO, and this data is written back to system memory as needed. When the final byte of data to be HMAC'd has been processed through the MDEU, the descriptor will cause the MDEU to write the HMAC to the indicated area in memory. The SEC will write the requested number of HMAC bytes to memory, and the CPU will compare the most significant 12 bytes of the HMAC generated by the SEC with the HMAC which was received with the in-bound packet. If the HMACs match, the packet integrity check passes.

9.4 Descriptor Type 1000_0 Example

The descriptor shown in Table 20 uses Descriptor Type 1000_0 to completely sets up the PKEU for an RSA sign or verify operation, a basic function in IKE, SSL, and certificate signature operations. The descriptor loads A -the data (message to be encrypted or decrypted), N -modulus, and E -public or private exponent; and writes B-Out -the encrypted/decrypted message back to memory. This descriptor is similar to the original SEC 1.0 method for performing IPSec operations, however ordering of the inputs (N, B, A, and E) is different. A test descriptor for this operation is provided in Section A.5, “RSA_SSTEP.”

Table 20. Representative Descriptor DPD_Type 1000_0_PK_MM_Encrypt

Field	Value/Type	Description
Header	0x5800_0080	DPD_Type 1000_0 PK_MM using RSA_SSTEP to Encrypt out-bound message using Public Key
Reserved	0x0000_0000	—
Length 0	Length	Number of significant bytes in N -Modulus (leading zeros not significant)
Pointer 0	Pointer	Address of N-Modulus
Length 1	Length	Null
Pointer 1	Pointer	Null
Length 2	Length	Number of bytes of A -data/message to be ciphered
Pointer 2	Pointer	Address of A- data/message
Length 3	Length	Number of bytes of E - public key (public exponent)
Pointer 3	Pointer	Address of E- public key
Length 4	Length	Number of significant bytes in B-Out -Encrypted Message out. Should be the same as Length 1
Pointer 4	Pointer	Address where B-Out -Encrypted Message is to be written
Length 5	Length	Null
Pointer 5	Pointer	Null
Length 6	Length	Null
Pointer 6	Pointer	Null

9.5 SEC 2.0 Specific Descriptors

The SEC 2.0 has several descriptor types which aren’t straightforward mappings of the SEC 1.0 descriptors. These descriptors were defined to offer improved support for current and emerging security protocols, such as IPSec, 802.11i, and SRTP. An example of each, with comparison of the SEC 1.0 method, is provided below.

9.6 Descriptor Type 0000_1: IPsec_ESP

The IPsec_ESP descriptor type is designed to efficiently process IPsec ESP packets with separate encryption and integrity algorithms.

Table 21 summarizes the IPsec ESP descriptor format.

Table 21. IPsec ESP Descriptor Format Summary

Descriptor Type	Field Type	Pointer Dword0	Pointer Dword1	Pointer Dword2	Pointer Dword3	Pointer Dword4	Pointer Dword5	Pointer Dword6	Usage
0000_1 **ipsec_esp**	**Length**	Integrity Key	Integrity Only Data	Primary EU Context In	Confidentiality Key	Data In	Data Out	Primary EU Context Out	IPsec_ ESP
	Extent	Reserved	Reserved	Reserved	Reserved	Reserved	CICV Out	Reserved	

9.6.1 IPsec-ESP Outbound

Figure 15 shows the diagram for the IPsec ESP outbound packet pointer.

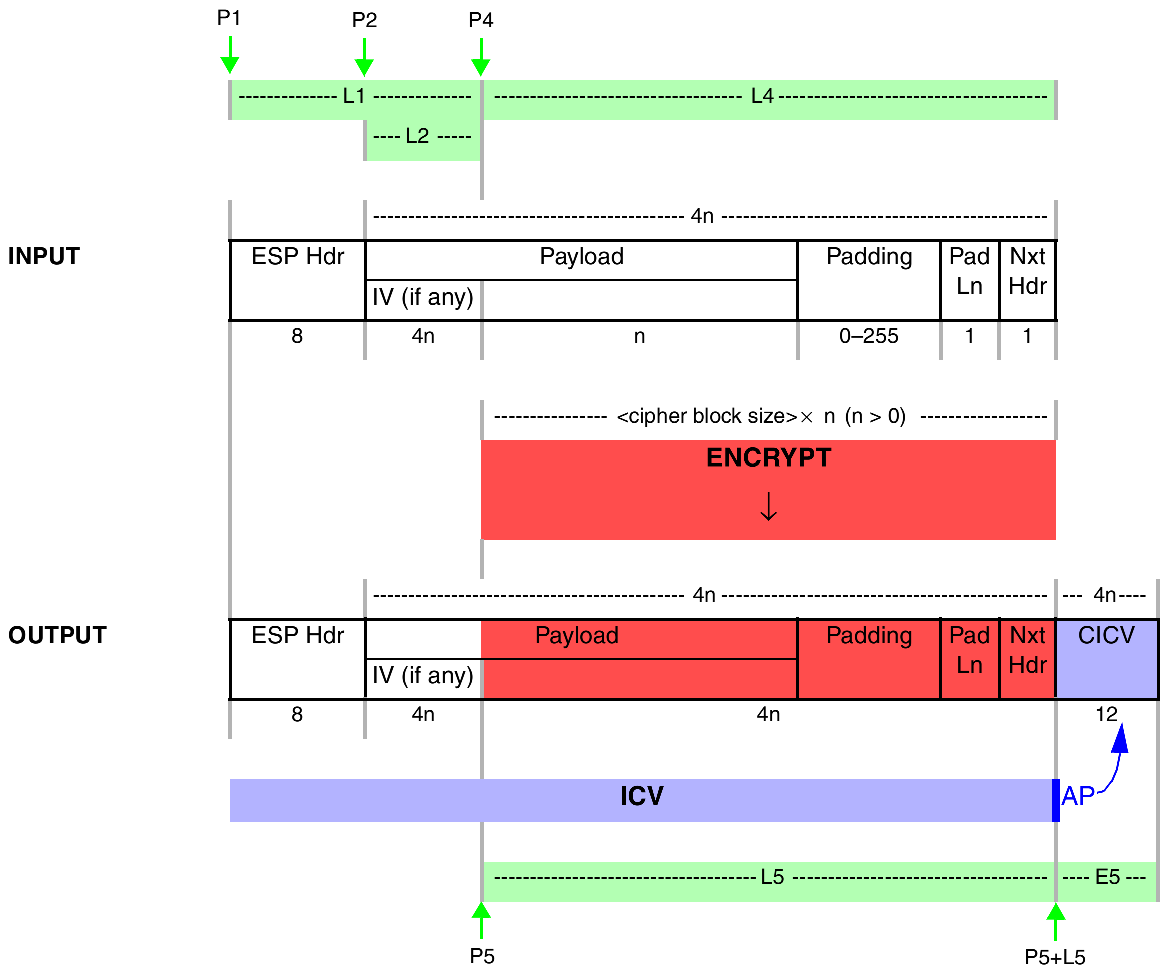

Figure 15. IPsec ESP Outbound Packet Pointer Diagram

Note: If the IV is in the packet (explicit IV), then the IV is the same as the last part of the ICV-only header. In this case the L1 and L2 regions may overlap in memory, as shown above.

The data processing steps (as managed by the channel) are as follows:

1. Starting at address P0, fetch L0 bytes of the integrity algorithm key (not shown).
2. Starting at address P1, fetch L1 bytes of authentication only data (the ESP header). The SEC Channel computes the overall authentication datasize from L1 + L4.
3. Starting at address P2, fetch L2 bytes of encryption IV. User SW is expected to place the IV in the proper location within the packet prior to the start of SEC ESP processing.
4. Starting at address P3, fetch L3 bytes of the encryption algorithm key (not shown)
5. Starting at P4, SEC fetches L4 bytes of plaintext and feeds them to the selected cipher EU input FIFO.
6. SEC write L5 bytes of ciphertext to P5.
7. After the cipher EU has finished encryption, the MDEU finishes computation of the CICV (HMAC). The SEC writes E5 bytes of CICV to the end of the encrypted data.

9.6.2 IPsec-ESP Inbound

Figure 16 shows the diagram for the IPsec ESP inbound packet pointer.

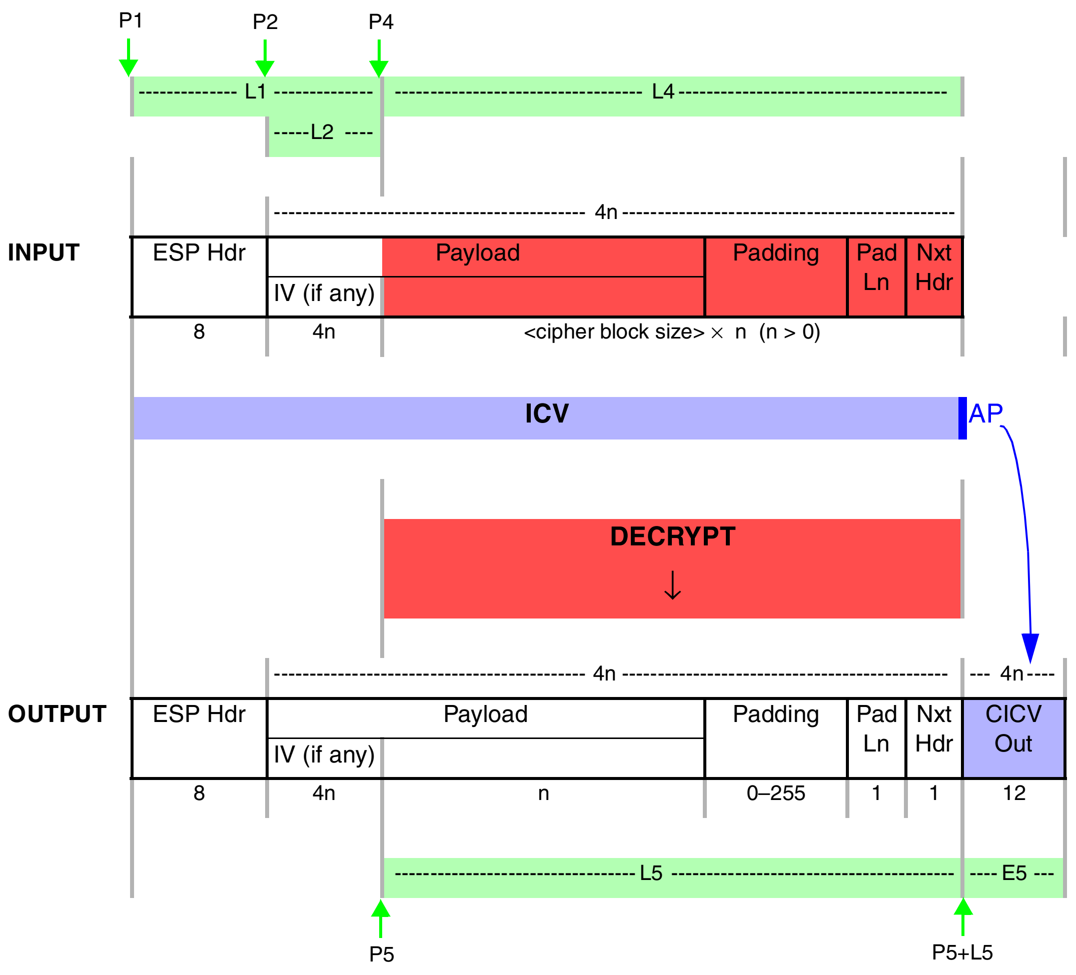

Figure 16. IPsec ESP Inbound Packet Pointer Diagram

Note:

1 If the IV is in the packet (explicit IV), then the IV is the same as the last part of the ICV-only header. In this case the L1 and L2 regions may overlap in memory, as shown above.

The data processing steps (as managed by the channel) are as follows:

1. Starting at address P0, fetch L0 bytes of the integrity algorithm key (not shown).
2. Starting at address P1, fetch L1 bytes of authentication only data (the ESP header). The SEC Channel computes the overall authentication datasize from L1 + L4.
3. Starting at address P2, fetch L2 bytes of encryption IV. The location of the IV in IPsec packet is defined by the standard.
4. Starting at address P3, fetch L3 bytes of the decryption algorithm key (not shown)
5. Starting at P4, SEC fetches L4 bytes of ciphertext and feeds them to the selected cipher EU input FIFO.

6. SEC write L5 bytes of plaint text to P5.
7. The MDEU recalculates the CICV (HMAC). The SEC writes E5 bytes of CICV to the end of the encrypted data.
   Note that E5 is written directly to the end of the decrypted data (P5 + L5). If a packet is being decrypted in place (for example, if the same buffer is being used for packet data in and data out, P5 = P4), then E5 would overwrite the original ICV (located at P4 + L4), which would not be desirable, since the host needs to compare the received ICV and CICVs. The solutions are to either use a different buffer for output (P5 != P4), or for the host to use the SEC's “scatter” capability to cause the CICV to be written to some more convenient place.

An example of a Type 0000_1 descriptor can be found in Section A.6, “IPSec_ESP.”

9.6.3 Descriptor Type 0001_1 for AES-CCM

Descriptor type 0001_1 is the preferred descriptor type for AES-CCM operations. Descriptor type 0001_0 can also be used, but 0001_1 has some advantages in dealing with Integrity Only Data, and in treatment of the MIC (ICV).

Table 22 shows the AES-CCM context register values. The encrypt and decrypt outputs are only shown for reference, as the register contents change from the start of the operation till its completion. There is no need to output all the context registers with descriptor pointer 6 unless the packet is being processed with continuing descriptors. If continuing descriptors are used, all 56 bytes of AES context needs to be output in order to be reloaded by the subsequent descriptor, otherwise only the first 24B need to ne output, as described in subsequent sections.

Table 22. AES Context Registers for CCM

Context Register (Byte Address)	Encrypt Input	Encrypt Output	Decrypt Input	Decrypt Output
1 (0x34100)	B0	MAC	B0	Computed MAC
2 (0x34108)		0		Typically 0
3 (0x34110)	0	Encrypted MIC	Received MIC	Decrypted MIC
4 (0x34118)		Typically 0		Typically 0
5 (0x34120)	Counter*	Running Counter	Counter*	Running Counter
6 (0x34128)				
7 (0x34130)	Counter Modulus Exponent*	Counter Modulus Exponent	Counter Modulus Exponent*	Counter Modulus Exponent

Note: The counter modulus for CCM cipher mode is currently defined as 2^{128} making the exponent 128. This value has been made programmable to support the various values used for counter modulus exponent in AES-CTR and AES-CCM modes. Because this is a programmable field, it must included in the IV each time the IV is loaded.

For 802.11i (WiFi) the Context includes the following:

B0 = 0x59 || 13-byte Nonce || 2-byte Payload Length

Initial Counter = 0x01 || 13-byte Nonce || 0x0000

For both B0 and the Counter, the Nonce = 1-byte PRI || 6-byte Address 2 || 6-byte Packet Number

For 802.11i (WiMax) the Context includes the following:

B0 = 0x19 || 13-byte Nonce || 2-byte Payload Length

Counter = 0x01 || 13-byte Nonce || 0x0000

For both B0 and the Counter, the Nonce = Bits 0:39 of GMH (GMH less Header Checksum) || 0x0000_0000 || 4-byte Packet Number, where Packet Number is 0x01_00_00_00 for PN=1

9.6.3.1 Type 0001_1 for IEEE Std 802.11i™_aes_ccmp

This descriptor type is suitable for IEEE 802.11i (WiFi) security. Table 23 summarizes the 802.11i AES-CCMP descriptor format.

Table 23. Descriptor Format Summary for IEEE 802.11i_aes_ccmp

Descriptor Type	Field Type	Pointer Dword0	Pointer Dword1	Pointer Dword2	Pointer Dword3	Pointer Dword4	Pointer Dword5	Pointer Dword6	Usage
0001_1 **802.11i AES ccmp**	**Length**	Reserved	Primary EU Context In	Confidentiality Key	Integrity Only Data	Data In	Data Out	Primary EU Context Out	IEEE 802.11i
	Extent	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	

Table 24 shows the descriptor header values for AES-CCM encrypt and decrypt.

Table 24. Descriptor Header Values for WiMax with AES-CCM

Operation	Descriptor Header
AES-CCM Encrypt	0x6b10_0019
AES-CCM Decrypt	0x6b00_001B

9.6.3.2 IEEE 802.11i Outbound

From the SEC's perspective, 802.11i processing is the same as other types of protocol processing; the SEC fetches and executes a descriptor. 802.11i requires more complex data preparation on the part of software, including the construction of the Authentication Only Data, the CCM Header, and the MAC Header.

Unlike other protocols in which the Authentication Only Data is the same as a transmitted header, the 802.11i AAD is a constructed value which adds fields not found in the MAC Header, removes a field that is in the MAC Header, and applies a mask to two other MAC Header fields.

Figure 17 shows the construction of the 802.11i (WiFi) AAD used during datagram authentication.

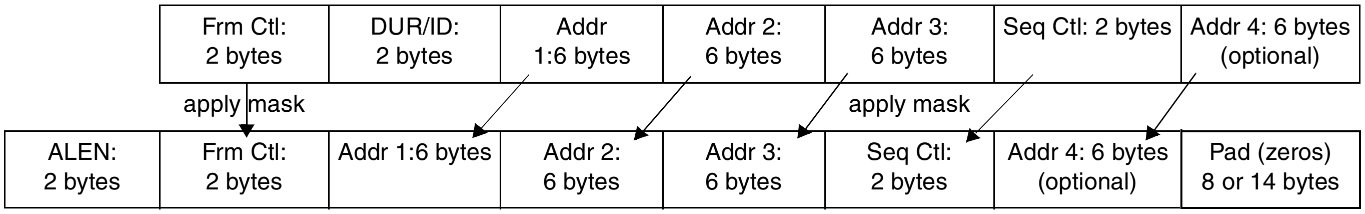

Figure 17. Construction of AAD from MAC Header

Notes on AAD construction:

1. ALEN field—This is the length of the AAD field, less the ALEN field itself and the PAD required to make the whole AAD an integral number of 16B blocks. Realistically, there are 2 possible values for ALEN and PAD:
   a) Optional Address 4 not used. ALEN = 22 bytes, Padding = 8 bytes
   b) Optional Address 4 used. ALEN = 28 bytes, Padding = 14 bytes
2. An explanation of Frame Control and Sequence Control masking is beyond the scope of this document.

The SEC's data processing steps for IEEE 802.11i (AES-CCM) outbound packet processing are as follows. All references to P, L, and E are as shown in Figure 18.

1. With P1 and L1, load the 56-byte AES-CCM Context, which is constructed as shown in Table 22.
2. With P2 and L2, load the 16-byte Confidentiality Key (not shown). This is actually a combined Confidentiality/Integrity Key, as it is also used to generate the MIC (CICV).
3. Starting at P3, read L3 bytes of the constructed AAD. The AESU automatically uses the "Alen" field to determine if any additional data (beyond the constructed AAD value) is also included in the authentication only data for MIC generation.
4. With P4 and L4, read the plaintext payload data. The SEC encrypts the data while simultaneously generating the MIC, which is itself encrypted.
5. With P5 and L5, write the ciphertext payload data. Setting P5 = P4 causes ciphertext to overwrite the plaintext. If this is not desired, set P5 to the address of a new buffer.
6. With P6 and L6, write out the first 24 bytes of the AES context register. Bytes 17–24 are the MIC, which must be copied by software to the end of the encrypted payload. Once the MIC has been copied, the 24 bytes of context can be discarded.

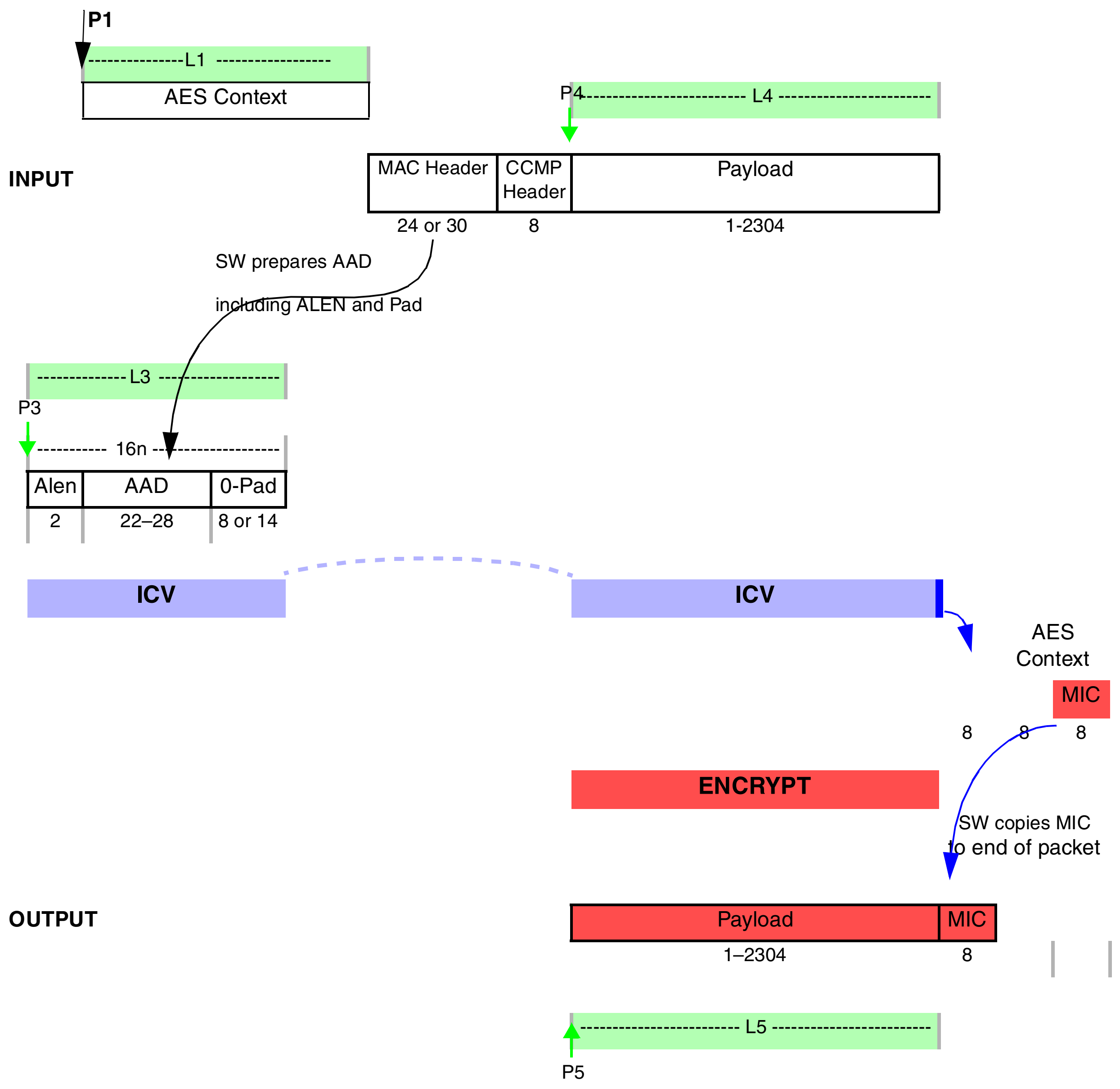

Figure 18. IEEE 802.11i (AES-CCM) Outbound Packet Pointer Diagram

9.6.3.3 IEEE 802.11i Inbound

Figure 19 shows the diagram for the IEEE 802.11i (AES-CCM) inbound packet pointer.

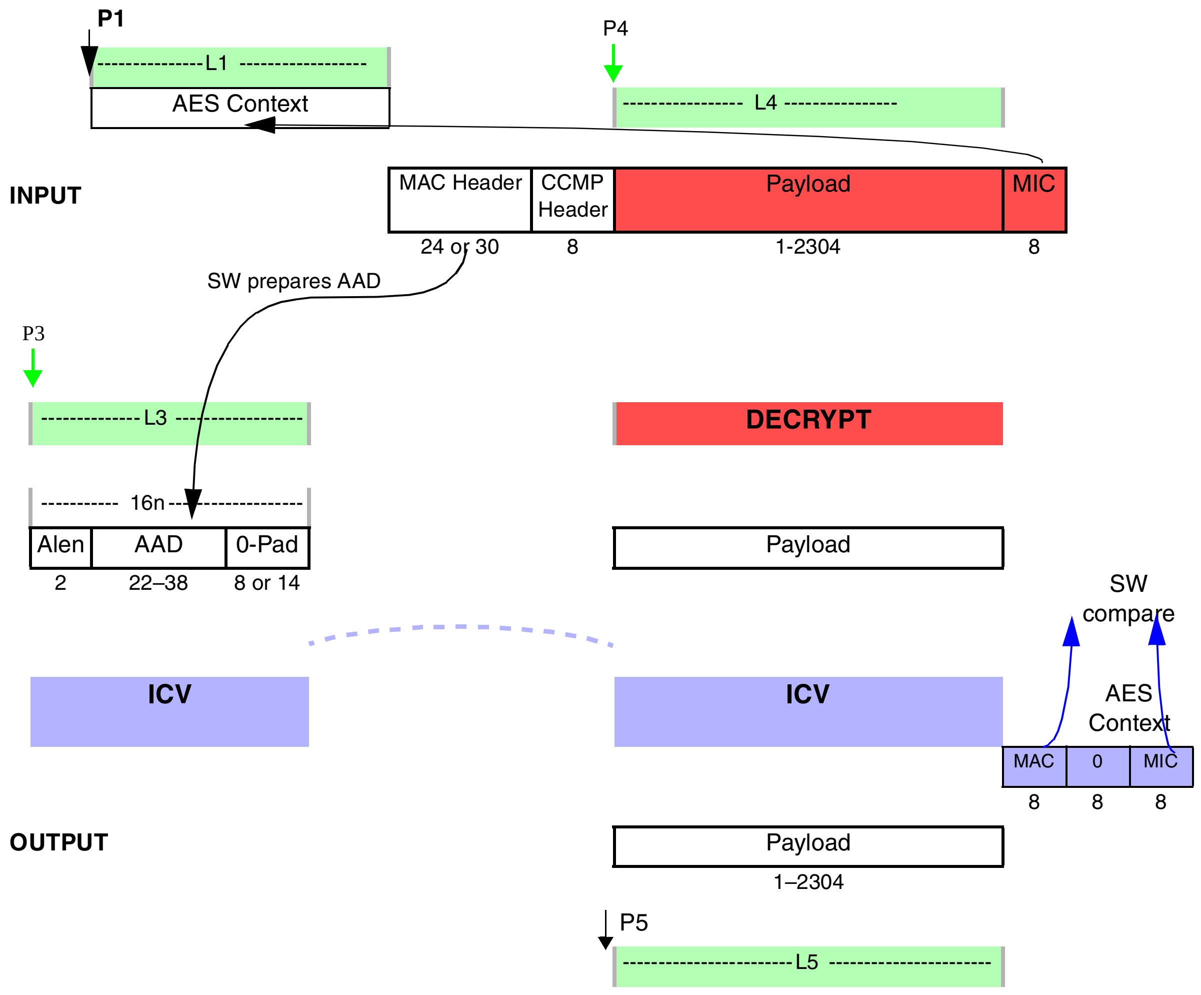

Figure 19. IEEE 802.11i (AES-CCM) Inbound Packet Pointer Diagram

The data processing steps (as managed by the channel) are as follows:

The SEC's data processing steps for IEEE 802.11i (AES-CCM) inbound packet processing are as follows. All references to P, L, and E are as shown in Figure 19.

1. With P1 and L1, load the 56-byte AES-CCM Context, which is constructed as shown in Table 22. As shown in Figure 19, the received MIC needs to be copied to the AES Context. The received MIC is typically 8B, however if larger, the received MIC can overflow into DWORD 4 of the AES Context Registers.
2. With P2 and L2, load the 16-byte Confidentiality Key (not shown). This is actually a combined Confidentiality/Integrity Key, as it is also used to generate the MIC (CICV).

3. Starting at P3, read L3 bytes of the constructed AAD. The AESU automatically uses the “Alen” field to determine if any additional data (beyond the constructed AAD value) is also included in the authentication only data for MIC generation.
4. With P4 and L4, read the ciphertext payload data. The SEC decrypts the data, while simultaneously regenerating the MIC for comparison.
5. With P5 and L5, write the plaintext payload data. L5 = L4.
6. With P6 and L6, write out the first 24 bytes of the AES context register. Bytes 0-7 are the calculated MAC, bytes 17–24 are the decrypted MIC. Software must compare these two values, and if they match, the received packet passes the integrity check. Once the MIC comparison has been completed, the 24 bytes of context can be discarded.

An example of a Type 0001_1 descriptor can be found in Section A.7, “AES_CCM.”

9.6.3.4 Descriptor Type 0001_1: AES-CCM for WiMax

AES-CCM, as required for WiMax security, is also performed using type 0001_1. Unlike 802.11i and IPsec with AES-CCM, WiMax does not treat the PDU header as Integrity Only Data. Header fields are included in the Nonce which becomes part of the IV, so any changes to the header during transmission would be cause the packet to fail to decrypt and fail the integrity check due to a bad MIC.

Table 25 summarizes the descriptor type 0001_1 descriptor format as it applies to AES-CCM for WiMax.

Table 25. Descriptor Format Summary for AES-CCM for WiMax

Descriptor Type	Field Type	Pointer Dword0	Pointer Dword1	Pointer Dword2	Pointer Dword3	Pointer Dword4	Pointer Dword5	Pointer Dword6	Usage
0001_1 **AES-CCM for WiMax**	**Length**	Reserved	Primary EU Context In	Confidentiality Key	Integrity Only Data (Reserved for WiMax)	Data In	Data Out	Primary EU Context Out	WiMax
	Extent	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved	

Table 25 shows the descriptor header values for AES-CCM encrypt and decrypt.

Table 26. Descriptor Header values for WiMax with AES-CCM

Operation	Descriptor Header
AES-CCM Encrypt	0x6b10_0019
AES-CCM Decrypt	0x6b00_001B

9.6.3.5 AES-CCM for WiMax encrypt

Figure 20 shows the diagram for AES-CCM MIC generation and encryption.

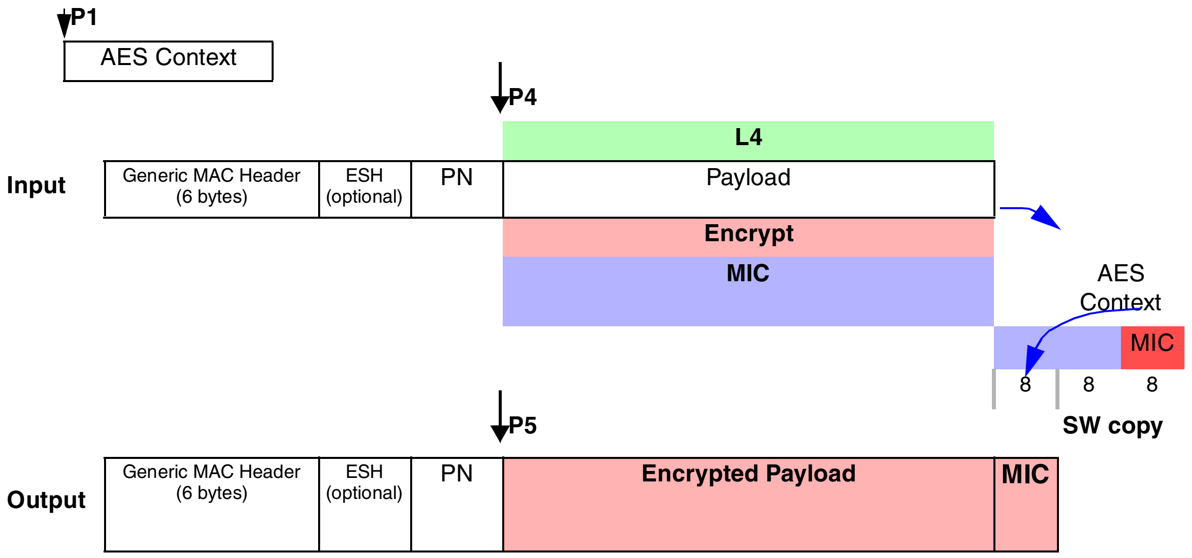

Figure 20. AES-CCM Encrypt

The data processing steps for AES-CCM encryption and MIC generation (as managed by the channel) are as follows:

1. With P1 and L1, load the 56-byte AES-CCM Context, which is constructed as shown in Table 22.
2. With P2 and L2, load the 16-byte Confidentiality Key (not shown). This is actually a combined Confidentiality/Integrity Key, as it is also used to generate the MIC.
3. For WiMax, there is no Integrity Only Data, so P3 and L3 are 0.
4. With P4 and L4, read the plaintext payload data. The SEC encrypts the data while simultaneously generating the MIC, which is itself encrypted.
5. With P5 and L5, write the ciphertext payload data. L5 = L. P5 can be used to overwrite the plaintext, or the data can be written to a new buffer.
6. With P6 and L6, write out the first 24 bytes of the AES context register. Bytes 17–24 are the computed MIC, which software must append to the end of the payload. Once the MIC has been appended, the 24 bytes of context can be discarded.

9.6.3.6 AES-CCM for WiMax decrypt

Figure 21 shows the diagram for AES-CCM MIC generation and encryption.

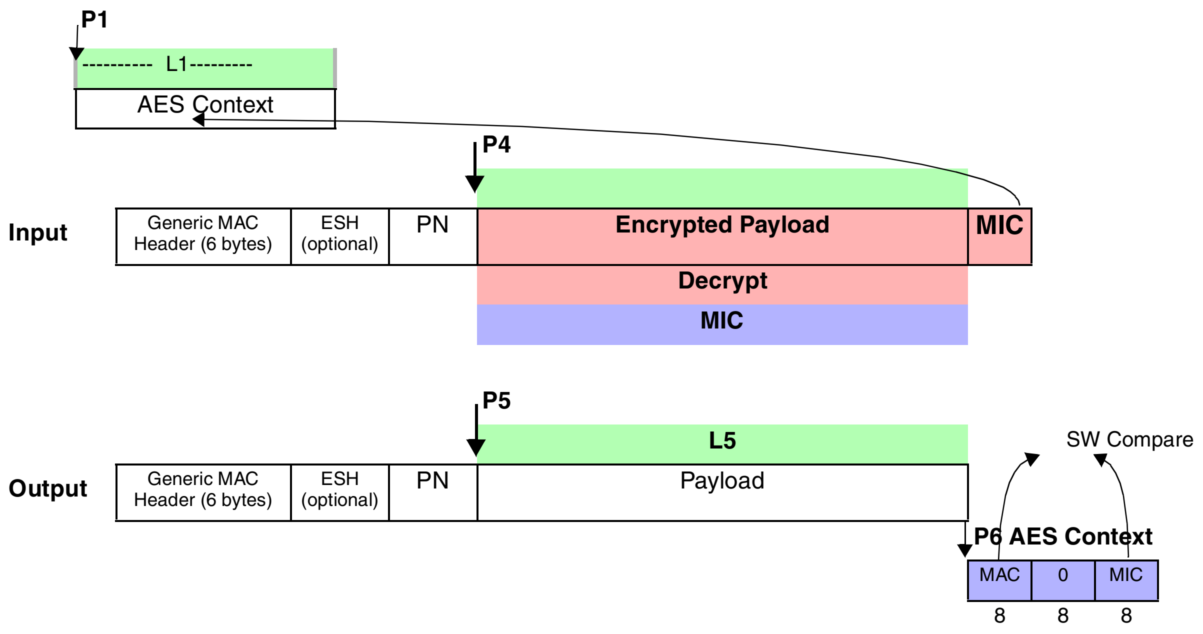

Figure 21. AES-CCM Decrypt

The data processing steps for AES-CCM decryption with hardware MIC as managed by the channel are as follows:

1. With P1 and L1, load the 56-byte AES-CCM Context, which is constructed as shown in Table 22. As shown in Figure 21, the received MIC needs to be copied to the AES Context. The received MIC is typically 8B, however it is larger, the received MIC can overflow into DWORD 4 of the AES Context Registers.
2. With P2 and L2, load the 16-byte Confidentiality Key. This is actually a combined Confidentiality/Integrity Key, as it is also used to generate the MIC.
3. For WiMax, there is no Integrity Only Data, so P3 and L3 are 0.
4. With P4 and L4, read the ciphertext payload and encrypted MIC data. The SEC decrypts the data and MIC while simultaneously recalculating the MIC.
5. With P5 and L5, write the ciphertext payload data. L5 = L4.
6. With P6 and L6, write out the first 24 bytes of the AES context register. Bytes 0-7 are the calculated MAC, bytes 17–24 are the decrypted MIC. Software must compare these two values, and if they match, the received packet passes the integrity check. Once the MIC comparison has been completed, the 24 bytes of context can be discarded.

9.6.4 Descriptor Type 0010_1: SRTP

Although other common or snooping descriptor types can be used to perform SRTP, this is the preferred descriptor type as it offers the ability to calculate the HMAC over both initial Integrity Only Data (the SRTP header) and trailing integrity only data (roll over counter).

Table 27 summarizes the SRTP descriptor format.

Table 27. SRTP Descriptor Format Summary

Descriptor Type	Field Type	Pointer Dword0	Pointer Dword1	Pointer Dword2	Pointer Dword3	Pointer Dword4	Pointer Dword5	Pointer Dword6	Usage
0010_1 **srtp**	**Length**	Integrity Key	Primary EU Context In	Confidentiality Key	Data In	Data Out	CICV Out	Primary EU Context Out	SRTP
	Extent	Reserved	Reserved	Reserved	Integrity Only Data	Integrity Only Trailer	Reserved	Reserved	

The Primary EU Context in Pointer DWORD 1 is the 24-byte AES context shown in Table 28.

Table 28. AES Context Registers for SRT versus CTR Mode

Context Register (Byte Address)	0010_1 with CTR
1 (0x34100)	Counter
2 (0x34108)	
3 (0x34110)	Counter modulus exponent
4 (0x34118)	—
5 (0x34120)	
6 (0x34128)	
7 (0x34130)	

9.6.4.1 Descriptor Type 0010_1 SRTP Outbound

Figure 22 shows the diagram SRTP outbound packet pointer.

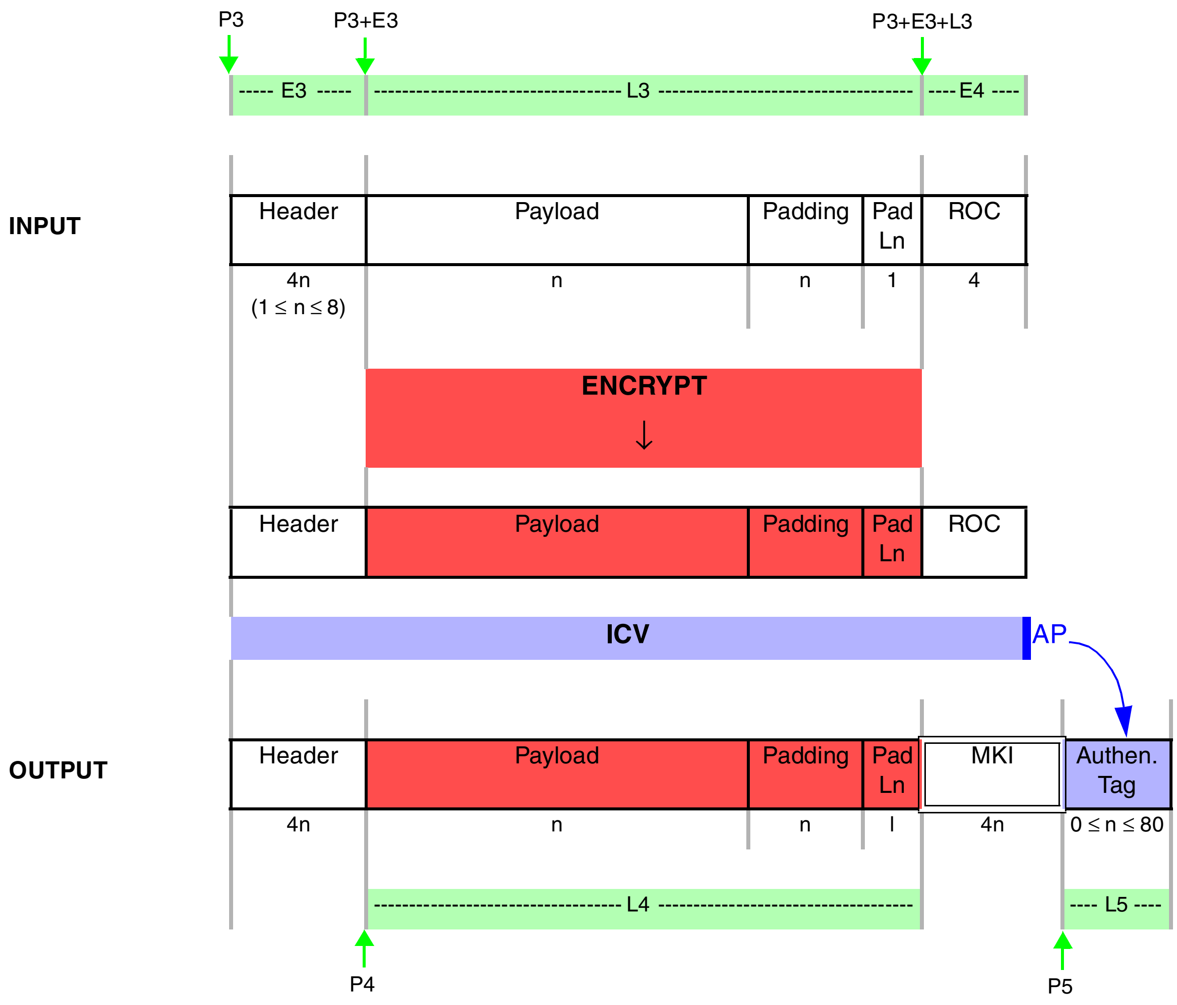

Figure 22. SRTP Outbound Packet Pointer Diagram

The data processing steps (as managed by the channel) are as follows:

1. Starting at address P0, fetch L0 bytes of the integrity algorithm key (not shown).
2. Starting at address P1, fetch L1 bytes of AES Context (not shown, should always be 24B).
3. Starting at address P2, fetch L2 bytes of encryption algorithm key (not shown).
4. Starting at P3, SEC fetches E3 bytes of plaintext packet data, which are treated as authentication only data. The SEC continues fetching L4 bytes of plaintext which are fed to both the encryption and integrity algorithms.
5. SEC write L4 bytes of ciphertext to P4. When it reaches the end of the ciphertext output, the SEC reads E4 bytes of ROC (rollover counter) and feeds it to the integrity algorithm. The ROC must be located at P3+L3+L4, contiguous with the Pad Ln field of the packet.
6. Authentication Tag output: Finish computation of the Authentication Tag in the MDEU. Obtain *L5* bytes of Authentication Tag from the MDEU and write them to *P5*.

Figure 23 shows the SRTP outbound descriptor format.

	0 — 15	16	17 — 23	24 — 27	28 — 31	32 — 63
Header Dword	Descriptor Control					Descriptor Feedback
Pointer Dword 0	ICV Key Length	J0	—	—	Eptr0	P0: Address of ICV Key
Pointer Dword 1	AES Context Length	J1	—	—	Eptr1	P1: Address of Cipher Context In
Pointer Dword 2	AES Key Length	J2	—	—	Eptr2	P2: Address of Cipher Key
Pointer Dword 3	Length of Plaintext	J3	Hdr Len	—	Eptr3	P3: Header • Plaintext • Trailer
Pointer Dword 4	Ciphertext Length	J4	Trlr Len	—	Eptr4	P4: Address of Ciphertext
Pointer Dword 5	AuthTag Length	J5	—	—	Eptr5	P5: Address of Authen. Tag Out
Pointer Dword 6	Cipher Ctxt Length	J6	—	—	Eptr6	P6: Cipher Context Out
	Length	**J**	**Extent**	**—**	**Eptr**	**Pointer**

Figure 23. SRTP Outbound Descriptor Format

9.6.4.2 Descriptor Type 0010_1 SRTP Inbound

Figure 24 shows the diagram for the SRTP inbound.

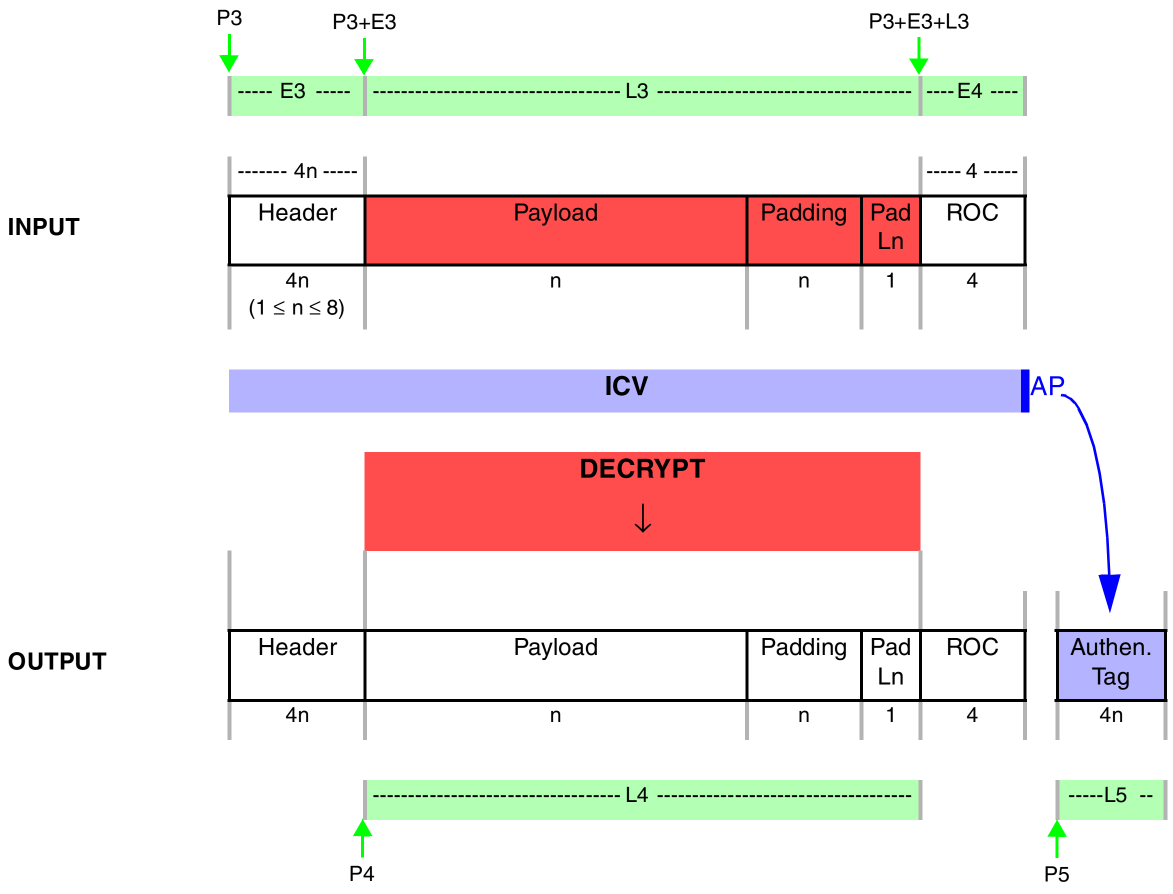

Figure 24. SRTP Inbound Packet Pointer Diagram

The data processing steps (as managed by the channel) are as follows:

1. Starting at address P0, fetch L0 bytes of the integrity algorithm key (not shown).
2. Starting at address P1, fetch L1 bytes of AES Context (not shown, should always be 24B).
3. Starting at address P2, fetch L2 bytes of decryption algorithm key (not shown).
4. Starting at P3, SEC fetches E3 bytes of ciphertext packet data, which are treated as authentication only data. The SEC continues fetching L4 bytes of ciphertext which are fed to both the decryption and integrity algorithms.
5. SEC write L4 bytes of plaintext to P4. When it reaches the end of the ciphertext output, the SEC reads E4 bytes of ROC (rollover counter) and feeds it to the integrity algorithm. The ROC must be located at P3+L3+L4, contiguous with the Pad Ln field of the packet.
6. Authentication Tag output: Finish computation of the Authentication Tag in the MDEU. Obtain *L5* bytes of Authentication Tag from the MDEU and write them to *P5*. P5 should be chosen so as not to overwrite the received Authentication Tag, otherwise SW won't be able to perform the comparison.

Figure 25 shows the SRTP inbound descriptor format.

	0 15	16	17 23	24 27	28 31	32 63
Header Dword	Descriptor Control					Descriptor Feedback
Pointer Dword 0	ICV Key Length	J0	—	—	Eptr0	P0: Address of ICV Key
Pointer Dword 1	AES Context Length	J1	—	—	Eptr1	P1: Address of Cipher Context In
Pointer Dword 2	AES Key Length	J2	—	—	Eptr2	P2: Address of Cipher Key
Pointer Dword 3	Length of Ciphertext	J3	Hdr Len	—	Eptr3	P3: Header • Plaintext • Trailer
Pointer Dword 4	Plaintext Length	J4	Trlr Len	—	Eptr4	P4: Address of Plaintext
Pointer Dword 5	AuthTag Length	J5	—	—	Eptr5	P5: Address of Authen. Tag Out
Pointer Dword 6	Cipher Ctxt Length	J6	—	—	Eptr6	P6: Cipher Context Out
	Length	**J**	**Extent**	**—**	**Eptr**	**Pointer**

Figure 25. SRTP Inbound Descriptor Format

An example of a Type 0010_1 descriptor can be found in Appendix A, "Protocol Examples," including Section A.8, "SRTP."

9.7 SSLv3.1/TLS1.0 Processing

The SEC is capable of assisting in SSL record layer processing, however for SSL v3.0 and earlier, this support is limited to acceleration of the encryption only. The MDEU does not calculate the version of HMAC required by early versions of SSL. SSLv3.1 and TLSv1.0 use the same HMAC version as IPSec (specified in RFC2104), which the SEC MDEU supports, allowing it to off-load both bulk encryption and authentication from the CPU.

SSLv3.1 and TLSv1.0 (henceforth referred to as TLS) record layer encryption/decryption is more complicated for hardware than IPSec, due to the order of operations mandated in the protocol. TLS performs the HMAC function first, then attaches the HMAC (which is variable size) to the end of the payload data. The payload data, HMAC, and any padding added after the HMAC are then encrypted. Parallel encryption and authentication of TLS “records” cannot be performed using the SEC snooping mechanisms which work for IPSec.

Performing TLS record layer encryption and authentication with the SEC requires two descriptors. For out-bound records, one descriptor is used to calculate the HMAC, and a second is used to encrypt the record, HMAC, and padding. For inbound records, the first descriptor decrypts the record, while the second descriptor is used to recalculate the HMAC for validation by the CPU.

The following examples and explanations cover TLS outbound and inbound processing using dynamic assignment.

9.7.1 Out-Bound TLS Descriptor 1

The first descriptor performs the HMAC of the record header and the record payload. In the example shown, the HMAC is generated using the MD-5 algorithm.

Table 29. Out-Bound TLS Descriptor 1

Field	Value / Type	Description
Header	0x31E0_0010	DPD_Type 0001_0 HMAC_MD-5
Reserved	0x0000_0000	Nill
Length 0	Length	Nill
Pointer 0	Pointer	Nill
Length 1	Length	Nill
Pointer 1	Pointer	Nill
Length 2	Length	Number of bytes of HMAC key to be written to MDEU Key register
Pointer 2	Pointer	Address of HMAC key
Length 3	Length	Number of bytes of data to be written to MDEU Input FIFO
Pointer 3	Pointer	Address of data
Length 4	Length	Nill
Pointer 4	Pointer	Nill
Length 5	Length	Number of bytes of HMAC to be written out to memory (always 16 bytes for MD-5)
Pointer 5	Pointer	Address where HMAC is to be written
Length 6	Length	Nill
Pointer 6	Pointer	Nill

The primary EU is the MDEU, with its mode bits set to cause the MDEU to initialize its context registers, perform auto-padding if the data size is not evenly divisible by 512 bits, and calculate an HMAC-MD-5.

At the conclusion of Out-bound TLS Descriptor 1, the crypto-channel has calculated the HMAC, placed it in memory, and has reset and released the MDEU.

9.7.2 Out-Bound TLS Descriptor 2

The second descriptor performs the encryption of the record, HMAC, pad length, and any padding generated to disguise the size of the TLS record. A test descriptor for this operation is provided in Section A.9, “ARC4.”

Table 30. Out-Bound TLS Descriptor 2

Field	Value/ Type	Description
Header	0x1000_0050	AFEU, new key, don’t dump context, perform permute
Reserved	0x0000_0000	Nill
Length 0	Length	Nill
Pointer 0	Pointer	Nill
Length 1	Length	Nill
Pointer 1	Pointer	Nill
Length 2	Length	Length of ARC-4 key
Pointer 2	Pointer	Pointer to ARC-4 Key
Length 3	Length	Length of data to be read and permuted
Pointer 3	Pointer	Pointer to data in memory
Length 4	Length	Length of data to be written after permutation
Pointer 4	Pointer	Pointer to memory buffer for write back
Length 5	Length	Nill
Pointer 5	Pointer	Nill
Length 6	Length	Nill
Pointer 6	Pointer	Nill

Not surprisingly, in-bound TLS processing reverses the order of operations of out-bound processing.

9.7.3 In-Bound TLS Descriptor 1

The first descriptor performs the decryption of the record, HMAC, pad length, and any padding generated to disguise the size of the TLS record.

Table 31. In-Bound TLS Descriptor 1

Field	Value/Type	Description
Header	0x1000_0050	AFEU, new key, don’t dump context, perform permute
Reserved	0x0000_0000	Nill
Length 0	Length	Nill
Pointer 0	Pointer	Nill
Length 1	Length	Nill

Table 31. In-Bound TLS Descriptor 1 (continued)

Field	Value/Type	Description
Pointer 1	Pointer	Nill
Length 2	Length	Length of ARC-4 key
Pointer 2	Pointer	Pointer to ARC-4 Key
Length 3	Length	Length of data to be read and permuted
Pointer 3	Pointer	Pointer to data in memory
Length 4	Length	Length of data to be written after permutation
Pointer 4	Pointer	Pointer to memory buffer for write back
Length 5	Length	Nill
Pointer 5	Pointer	Nill
Length 6	Length	Nill
Pointer 6	Pointer	Nill

NOTE

ARC-4 does not have a concept of Encrypt vs. Decrypt. As a stream cipher, ARC-4 generates a key stream which is XOR'd with the input data. If the input data is plaintext, the output is ciphertext. If the input data is ciphertext (which was previously XOR'd with the same key), the result is plaintext.

The primary EU is the AFEU, with its mode bits set to cause the AFEU to load the key and initialize the AFEU S-box for data permutation.

At the conclusion of In-bound TLS Descriptor 1, the AFEU has decrypted the TLS record so that the payload and HMAC are readable. The negotiation of the TLS session should provide the receiver with enough information about the session parameters (hash algorithm for HMAC, whether padding is in use) to create In-Bound Descriptors 2 and dispatch it to the same channel's Fetch FIFO prior to completion of In-Bound Descriptor 1.

Alternatively, the SEC could signal DONE at the conclusion of In-Bound Descriptor 1 to allow the CPU to inspect the decrypted record, and generate the descriptor necessary to validate the HMAC. If this is the case, In-Bound Descriptor 2 does not need to be linked to In-Bound Descriptor 1, and could even be processed by a different crypto-channel.

9.7.4 In-Bound TLS Descriptor 2

The second descriptor performs the HMAC of the record header and the record payload. In the example shown, the HMAC is generated using the MD-5 algorithm.

Table 32. In-Bound TLS Descriptor 2

Field	Value/ Type	Description
Header	0x31E0_0010	MDEU, HMAC, MD-5, Autopad
Reserved	0x0000_0000	Nill
Length 0	Length	Nill
Pointer 0	Pointer	Nill
Length 1	Length	Nill
Pointer 1	Pointer	Nill
Length 2	Length	Length of MD-5 key
Pointer 2	Pointer	Pointer to MD-5 Key
Length 3	Length	Length of data to be read and permuted
Pointer 3	Pointer	Pointer to data in memory
Length 4	Length	Nill
Pointer 4	Pointer	Nill
Length 5	Length	Length of HMAC to be written to memory (16 bytes for MD-5)
Pointer 5	Pointer	Pointer to memory location for HMAC write
Length 6	Length	Nill
Pointer 6	Pointer	Nill

The primary EU is the MDEU, with its mode bits set to cause the MDEU to initialize its context registers, perform auto-padding if the data size is not evenly divisible by 512 bits, and calculate an HMAC-MD-5.

The descriptor header doesn't designate a secondary EU, so the setting of the Snoop Type bit is ignored.

At the conclusion of In-bound TLS Descriptor 2, the crypto-channel has calculated the HMAC, placed it in memory, and has reset and released the MDEU. The CPU can compare the HMAC generated by In-Bound TLS Descriptor 2 with the HMAC that came as part of the record. If the HMACs match, the record is known to have arrived unmodified, and can be passed to the Application Layer.

10 Conclusion

The SEC 2.0 is capable of accelerating a wide range of common cryptographic operations. Users can take advantage of the SEC 2.0 Reference Device Driver provided by Freescale, or with the understanding of SEC descriptor construction provided by this application note, users can create their own device driver or optimized cryptographic macro routines.

11 Revision History

Table 33 provides a revision history for this application note.

Table 33. Document Revision History

Rev. Number	Date	Substantive Change(s)
1	04/2010	• Updated terminology. • Updated Section 9.6, “Descriptor Type 0000_1: IPsec_ESP.” • Updated Section 9.6.3, “Descriptor Type 0001_1 for AES-CCM.” • Updated Section 9.6.4, “Descriptor Type 0010_1: SRTP.” • Updated Section A.7, “AES_CCM.” • Updated Section A.8, “SRTP.”
0.2	02/08/05	• Updated Figure 14, “Descriptors, Link Tables, and Data Parcels.” • Added Appendix A, “Protocol Examples.”
0.1	01/21/05	General content updates
0	01/17/05	Initial public release

Appendix A Protocol Examples

The following are real examples of the descriptors shown in several of the tables. Not all tables have a matching example descriptor in the case of redundancy.

A.1 3DES_CBC_ENC

```
encrypt_type : desa


begin_descriptor:
                        //   descriptor type = common_nonsnoop_no_afeu
                        //   cipher function = triple-des-cbc
                        //   direction = outbound
                        //   done notification = off
                        //   primary cha = desa

    20700010            // Header word 1
           0            // Header word 2

           0            // 0 Length (unused)
           0            //   Extent (unused)
           0            //   Nil pointer 0

           8            // 1 Length of cipher context in = 8
           0            //   Extent (unused)
         @p1            //   Pointer to cipher context in

          18            // 2 Length of cipher key = 24
           0            //   Extent (unused)
         @p2            //   Pointer to key

          68            // 3 Length of cipher data = 104
           0            //   Extent (unused)
         @p3            //   Pointer to cipher data

          68            // 4 Length of cipher output = 104
           0            //   Extent (unused)
         @q4            //   Pointer to cipher out

           8            // 5 Length of cipher context out = 8
           0            //   Extent (unused)
         @q5            //   Pointer to auth/context out

           0            // Nil length 6
           0            //   Extent (unused)
           0            //   Nil pointer 6
end_descriptor

begin_memory p1:                // cipher IV in
        9163BD902F531D50
end_memory

begin_memory p2:                // cipher key
        83638CA559AC9C3F
```

A.2 AES_CCM_ENC

```
encrypt_type : aesa

begin_descriptor:
                        //   descriptor type = common_nonsnoop_no_afeu
                        //   cipher function = aes-ccm
                        //   direction = outbound
                        //   done notification = off
                        //   primary cha = aesa

    6b100010            // Header word 1
           0            // Header word 2

           0            // 0 Length (unused)
           0            //   Extent (unused)
           0            //   Nil pointer 0
```

```
          38            // 1 Length of cipher context in = 56
           0            //   Extent (unused)
         @p1            //   Pointer to cipher context in

          18            // 2 Length of cipher key = 24
           0            //   Extent (unused)
         @p2            //   Pointer to key

         310            // 3 Length of auth only data + cipher data = 784
           0            //   Extent (unused)
         @p3            //   Pointer to cipher data

          a0            // 4 Length of cipher output = 160
           0            //   Extent (unused)
         @q4            //   Pointer to cipher out

          20            // 5 Length of cipher context out = 32
           0            //   Extent (unused)
         @q5            //   Pointer to auth/context out

           0            // Nil length 6
           0            //   Extent (unused)
           0            //   Nil pointer 6
end_descriptor

begin_memory p1:                // CCMP IV
        56EC035771A136DA
        4E000000000000A0
                                //
        0000000000000000
        0000000000000000
                                // CCMP counter
```

A.3 HMAC_MD_5 Out

```
encrypt_type : mdha
```

```
begin_descriptor:
                        //   descriptor type = common_nonsnoop_no_afeu
                        //   authentication function = hmac-md5
                        //   direction = outbound
                        //   done notification = off
                        //   primary cha = mdha

    31e00010            // Header word 1
           0            // Header word 2

           0            // 0 Length (unused)
           0            //   Extent (unused)
           0            //   Nil pointer 0

           0            // 1 Length of cipher context in = 0
           0            //   Extent (unused)
           0            //   Nil pointer 1

           4            // 2 Length of auth key = 4
           0            //   Extent (unused)
         @p2            //   Pointer to key

         308            // 3 Length of cipher data = 776
           0            //   Extent (unused)
         @p3            //   Pointer to cipher data

           0            // Nil length 4
           0            //   Extent (unused)
           0            //  Nil pointer 4

          10            // 5 Length of auth out = 16
           0            //   Extent (unused)
         @q5            //   Pointer to auth/context out
```

```
              0            // Nil length 6
              0            //   Extent (unused)
              0            //   Nil pointer 6
end_descriptor

begin_memory p2:                // auth key
        543203C500000000
end_memory

begin_memory p3:                // cipher data in
        7CD566AA543D541D
```

A.4 3DES_HMAC_SHA_1

```
encrypt_type : desa

begin_descriptor:
                           //   descriptor type = hmac_snoop_no_afeu
                           //   cipher function = triple-des-cbc
                           //   authentication function = hmac-sha-1
                           //   direction = inbound
                           //   done notification = off
                           //   primary cha = desa
                           //   secondary cha = mdha

    20631c22               // Header word 1
           0               // Header word 2

          31               // 0 Length of authenticate key = 49
           0               //   Extent (unused)
         @p0               //   Pointer to auth key

          45               // 1 Length of auth_only_data = 69
           0               //   Extent (unused)
         @p1               //   Pointer to auth_only_data in
```

```
          18             // 2 Length of cipher key = 24
           0             //   Extent (unused)
         @p2             //   Pointer to cipher key

           8             // 3 Length of cipher context in = 8
           0             //   Extent (unused)
         @p3             //   Pointer to cipher context in

          38             // 4 Length of cipher in = 56
           0             //   Extent (unused)
         @p4             //   Pointer to cipher in

          38             // 5 Length of cipher out = 56
           0             //   Extent (unused)
         @q5             //   Pointer to cipher out

          14             // 6 Length of hmac out = 20
           0             //   Extent (unused)
         @q6             //   Pointer to hmac out
end_descriptor

begin_memory p0:                 // authentication key
        3408BD82AF03C5E9
        01C59BB6CE71DB35
        F9C14175ACAF79F7
        3C8104299C25F002
        9369AD7EA8BAF85D
        B158E38B1BE67A05
        BD00000000000000
end_memory

begin_memory p1:                 // auth only data
        746640CA17491D24
        4B573EED816D5CDD
        48ADC44F86272616
        6EBF00594E07C4F2
        14F24ACFD4A90D4A
        A9B1D35C6C85CC0F
        5E17515D881B2B35
        A53C62660421ABD8
        A1227AEEC2000000
end_memory

begin_memory p2:                // cipher key
        82CD41B2076632A6
        E7EB3A23ADCC655F
        E1C3173ECB5B3E07
end_memory

begin_memory p3:                // cipher IV in
        FA9AB7D9CFA25F33
end_memory

begin_memory p4:                // cipher data in
        ADB6EA9B95BFE619
        AEC195DFB32940BE
        3EC54FC35B72C830
        30B0310FFD6633AC
        506A79E759B55D18
        C8B23A44CC1B454F
        8BF4F9911DE29C68
end_memory

begin_memory exp_q5:                // cipher data out
        2B81FEA2A88AC074
        B80CE859EAF198F2
        10B8C77E0EBE1EF1
        C118AD63E63EBC86
        67626F335A02D445
        61ED4E103F4ED30D
        359D3E686C827329
end_memory

begin_memory exp_q6:                    // authentication code out
        5B27A503EDFF1DA3
        16CB111110271E19
        5852F3A400000000
end_memory
```

A.5 RSA_SSTEP

```
encrypt_type : pkha

begin_descriptor:
                        //   descriptor type = pkha_mm
                        //   pkha function = rsa_sstep
                        //   done notification = off
                        //   primary cha = pkha

    58000080            // Header word 1
           0            // Header word 2

          10            // 0 Length of N = 16
           0            //   Extent (unused)
         @p0            //   Pointer to reg N

           0            // Nil length 1
           0            //   Extent (unused)
           0            //   Nil pointer 1

          10            // 2 Length of A = 16
           0            //   Extent (unused)
         @p2            //   Pointer to reg A

           d            // 3 Length of E = 13
           0            //   Extent (unused)
         @p3            //   Pointer to reg E

          10            // 4 Length of output = 16
           0            //   Extent (unused)
         @q4            //   Pointer to output

           0            // Nil length 5
           0            //   Extent (unused)
           0            //   Nil pointer 5

           0            // Nil length 6
           0            //   Extent (unused)
           0            //   Nil pointer 6
end_descriptor

begin_memory p0:                // reg N
        8F9ABC2052DDBBD6
        899F1817033239CB
end_memory

begin_memory p2:                // reg A
        2E41675F09B3986D
        7FE9206FEA76372D
end_memory

begin_memory p3:                // reg E
        0F9E2DBBA7C57CFA
        9C52ACA0DB000000
end_memory

begin_memory exp_q4:                // expected output data
        89e015aeccd25569
        7049a6466982543c
end_memory
```

A.6 IPSec_ESP

```
encrypt_type : desa


begin_descriptor:
                        //   descriptor type = ipsec_esp
                        //   cipher function = triple-des-cbc
                        //   authentication function = hmac-sha-1
                        //   direction = inbound
                        //   done notification = off
                        //   primary cha = desa
                        //   secondary cha = mdha

    20631c0a            // Header word 1
           0            // Header word 2

          14            // 0 Length of authenticate key = 20
           0            //   Extent (unused)
         @p0            //   Pointer to auth key

          10            // 1 Length of auth-only data = 16
           0            //   Extent (unused)
         @p1            //   Pointer to auth-only data

           8            // 2 Length of cipher context in = 8
           0            //   Extent (unused)
         @p2            //   Pointer to cipher context in

          10            // 3 Length of cipher key = 16
           0            //   Extent (unused)
         @p3            //   Pointer to cipher key

         270            // 4 Length of cipher data in = 624
           0            //   Extent (unused)
         @p4            //   Pointer to cipher data in
```

```
          270            // 5 Length of cipher data out = 624
            c            //   Extent of auth result = 12
          @q5            //   Pointer to cipher data out

            8            // 6 Length of cipher context out = 8
            0            //   Extent (unused)
          @q6            //   Pointer to cipher context out
end_descriptor

begin_memory p0:                    // authentication key
        0EC76B5E9ECF6AA2
        3ED269F912EA9BB8
        0314872500000000
end_memory

begin_memory p1:                    // auth-only data in
        B9436C94BA6EFD59
        6308FB5C14FB0690
end_memory

begin_memory p2:                    // cipher context in
        6308FB5C14FB0690
end_memory

begin_memory p3:                    // cipher key
        595389E02EDE4F18
        18460AF0130C1BE0
end_memory

begin_memory p4:                    // cipher data in
        397C32968F978389
        9916BB00BA7B6C1B
        ED1DF1A10C51491C
        655FFA17F530AAE4
        19741D8A9A1ACCC2
```

```
687F00415BF2C3B9

6078D508F412F262

6AAA9B9CA10D79EB

690E80E63ECE8432

9BB3923FD88D3943

B3059AF8EE7B7A8C

FE2A21BA4BE5B406

D6736BBC2C052ABD

94D827AAC82C3794

DE5D8A0E61B31880

D89A74CB2235357A

6253B814E34714AF

554219A21AE9EE77

D51CA0AB6B039829

2CD3000515F0AE44

7F7E69B5D8BEC9C3

DEB4CA653390DF67

46C4E70FF3FB17A3

2A507E6A36116C1B

8B225886A87DF911

04B630665E612FF2

46A5DD62B649040A

35D49E01AEF93FF5

FF3B8DF53C7D64EA

29613CB2E62DFD39

25F6553ED5692F7C

317D2B5CF1F73759

344C3244A178AAA3

BBDD95C2AAA705B6

B18CA89C17EC447E

F01DC5608D7C42F8

3D62F5EB675E1649

EF6172168D069E53

1054DDD542C8C059

24B4F44B173254D3
```

```
CC0D7365DE895C6E

9F6F3E530AB982AB

C0942F8DB13797C7

3A8877AA037066C4

8CD47091FCD6C669

5D0C8DFF9398F668

79DDD39386168155

A08F0F4425FB6D07

2A968D969A777B1D

209C528BE24E8A5B

32B5500A0CB79419

C85217A960B69138

791C96C93532190E

A7D891D951808923

1B609F020D83CA91

373FE8066748D47C

42ADC02D1D0A5F00

6EA5CEEBD0400FFE

DBB34EA59B82F9A8

F27805B79D34AAA1

1FA8C1BC5ED4A1D6

D7FC2041159C5274

B254032BB2293BD3

D2B6E98C41B63541

358311C28E6CD926

E4FBB0E18A0C4235

F1524C4BA8992C3E

1902A0A9F4A6536C

149BC24177B133B2

4D76F09D4A54AA1A

F1D7764D0412A3EF

D0B6254B4DB9439B

1FAA63CD2E6EF8D8

F2AE6092B0FB043B

53ECD76EF86EE05E
```

```
        63B6865DEAC16F4E
        1DA15F8E0AB6DE28
        A3DDB4AACF89F5D5
end_memory

begin_memory exp_q5:                 // cipher data out
        897D2C0277633F99
        E059FAEA7B08C520
        C75037E88271FFA6
        18B4527709832D23
        73FABFAF3C75B4D1
        16302B6B8C5CBC9B
        4649B1675B8D7207
        3E1AABDABAF7CB3E
        4A7D636A42AD9E7E
        F7A154A1F2B58D6F
        D13CED20DAAE9A6D
        EB5B6F999FF81F8B
        49A87C2586D6B51F
        A78960C890AF6C1F
        855757C307FB0A01
        B9EAEF6B113DF435
        760DC4373D210FA3
        B82DFA37CE930BD3
        87543D2C477BABEA
        BD6C5FDDC6ADDF66
        EFFDACF12D2F1F24
        39D2D1E8B9D243E5
        14124BEFCE9D704C
        2046E690B83FF228
        8935F7A68688AC35
        5F2C16C836DDDAA3
        35B993209AADE39E
        81F101B8F934449C
        C2204FEB951D7D6D
```

```
A32044BC646C37FE
2E36ED455075F312
D5B16E303B3A23C0
40922488C09FDA18
D25A3224E877ED9C
8D82B09F4732BEF8
79FA6097408B2637
714639D522CDA3FC
43EC8A93D69BC293
F3316AF8555226EA
3E5865FF114F731D
D9F84CBC583F36B8
C59A1E7BF2893C75
12BA2A545515245D
C7A5C70F25981A1A
DF87490A121FBFDA
43839617AF4FCE4F
9AC7D6FCAC3A3637
F01EFAFBB7E0A96C
150B97182EC9D06A
BA556664EA329B03
58F40316F21B8643
74D7A5C4FE9E4A76
78A8B22EB7CBBF3C
4AFB595F90A1CEBE
65ACCABF8C536C94
EE87FC61AA34B561
A83617CF7008362D
C2AAAFBD54C389CA
C2B4A7C11096E190
E4134B0636A93ECB
142040E3FFD79207
3AE70FA41C46CBE5
35978156502CAF61
584E5EE11F50F8EB
```

```
        AF29AE5F28EEED3F
        D2B080DA5B099719
        7CB7BB560C3E7C14
        F95DFB3D82EFA5D9
        94C5D0C14BB87498
        CDB0A9E43E19448E
        CD0125786B8D53A7
        3BA9FCE3C93F00EF
        F2E2EDB7995A96C1
        FBEC001E4E2C4D51
        A1080768F9131072
        EF0FB4ED1A50918E
        430034C3304F98D3
        9CCFF6F06820D6D7
// Get New Link
                                // authentication code out
        3CF990619CD2038C
        2ECE1C4600000000
end_memory

begin_memory exp_q6:                // cipher context out
        A3DDB4AACF89F5D5
end_memory
```

A.7 AES_CCM

```
// Source: IEEE P802.11i / D3.2 Preliminary Draft:
//      Annex F.4 CCMP Test Vectors, CCMP test mpdu 1

encrypt_type : aesa

begin_descriptor:
                        //          Pri  Pri Mode  Sec  Sec Mode   Type
    6B100018            //Header 0110 1011-0001 0000 0000-0000  0001-1 000
                        //       AESA |                          CCMP    |
                        //            CCM                                Outbound
```

```
                    //              Final MAC                        No-
                    //               Initialize                      notify
                    //                 Expand key
                    //                  CCM
                    //                    Encrypt
                    //
0                   // Writeback value / Reserved

0                   // 0 Length UNUSED
0                   //   Extent
0                   //   Pointer

38                  // 1 Length of IV in = 56
0                   //   Extent
@p1                 //   Pointer to IV in

10                  // 2 Length of key = 16 (TK)
0                   //   Extent
@p2                 //   Pointer to key

20                  // 3 Length of hash-only data = 32
0                   //   Extent
@p3                 //   Pointer to hash-only data

14                  // 4 Length of cleartext data in = 20
0                   //   Extent
@p4                 //   Pointer to cleartext data in

14                  // 5 Length of encrypted data out = 20
0                   //   Extent
@q5                 //   Pointer to encrypted data out

20                  // 6 Length of IV out = 8 (MIC)
0                   //   Extent
@q6                 //   Pointer to IV out
```

```
end_descriptor



begin_memory p1:            // IV in:


                            // Registers 1-2
                            //   Code      Nonce
                            //   (binary)  (hex)
        59005030f1844408    //   01011001  005030f1844408b5039776e70c  0014
        b5039776e70c0014    //    |                                    |
                            //    Header is hash-only                  Message
                            //     MAC size = 8                        length=20
                            //         Nonce size = 13

        0000000000000000    // Registers 3-4
        0000000000000000


                            // Registers 5-6
                            //   Code      Nonce
                            //   (binary)  (hex)
        01005030f1844408    //   00000001  005030f1844408b5039776e70c  0000
        b5039776e70c0000    //         |                               |
                            //         Nonce size = 13                 Initial
                            //                                         Counter=0

                            // Register 7
        0000000000000080    //   Counter modulus = 2**128
end_memory

begin_memory p2:            // Key in
        c97c1f67ce371185
        514a8a19f2bdd52f
end_memory

begin_memory p3:            // Hash-only data
```

```
        001608400fd2e128
        a57c5030f1844408
        abaea5b8fcba0000               //Should not have 0s at the end.
                                       //Should make a descriptor in which
                                       //padding does not start at word boundary
        0000000000000000    // Padding
end_memory

begin_memory p4:            // Cleartext data in
        f8ba1a55d02f85ae
        967bb62fb6cda8eb
        7e78a050
end_memory

begin_memory exp-q5:        // Encrypted data out
        f3d0a2fe9a3dbf23
        42a643e43246e80c
        3c04d019
end_memory

begin_memory exp_q6:        // IV out:

        fcd14abd11309b04    // Registers 1-2: MAC out
        0000000000000000

        7845ce0b16f97623    // Registers 3-4: MIC out
        0000000000000000
end_memory
```

A.8 SRTP

```
encrypt_type : aesa

begin_descriptor:
                        //   descriptor type = srtp
```

```
                      //   cipher function = aes-ctr
                      //   authentication function = hmac-sha-1
                      //   direction = inbound
                      //   done notification = off
                      //   primary cha = aesa
                      //   secondary cha = mdha

64631c2a              // Header word 1
       0              // Header word 2

      14              // 0 Length of authenticate key = 20
       0              //   Extent (unused)
     @p0              //   Pointer to auth key

      18              // 1 Length of cipher context in = 24
       0              //   Extent (unused)
     @p1              //   Pointer to cipher context in

      10              // 2 Length of cipher key = 16
       0              //   Extent (unused)
     @p2              //   Pointer to cipher key

      90              // 3 Length of cipher data = 144
      10              //   ExtentA header length = 16
     @p3              //   Pointer to cipher data

      90              // 4 Length of cipher output = 144
       4              //   ExtentB ROC length = 4
     @q4              //   Pointer to cipher output

       8              // 5 Length of auth result = 8
       0              //   Extent (unused)
     @q5              //   Pointer to auth result

      10              // 6 Length of cipher context out = 16
```

```
           0            //   Extent (unused)
         @q6            //   Pointer to cipher context out
end_descriptor

begin_memory p0:                // authentication key
        FEEBD056EA52493E
        7A78265E9751EA35
        0BF9398200000000
end_memory

begin_memory p1:                // initial counter
        404C8CF1F36DD21F
        31DEC14880B20000
                                // ctr modulus
        0000000000000050
end_memory

begin_memory p2:                // cipher key
        317F037A003BC706
        A58A1BB461DC6F94
end_memory

begin_memory p3:                // auth-only header in
        EC5D54CC55BFCC2E
        B781A5FD605645E1
// Get New Link
                                // cipher data in
        4B613B5DC90890E1
        78BE5B08E0457563
        ECA9935959A0D5A6
        D602CA3BA8B11F97
        DE07EE07355A9667
        CF735CBB0F4E9EDA
        5483ED8AE8733F8B
        4681FB0B72082836
        28DEB4B8A510548B
        28BD7F45190804CF
        63D161742A1792F9
        C3C44E79DA90543B
        296736A77C1B2E85
        A33DCF86D98CA868
        2C444A0D5363A48E
        37B8AAB386383C39
        AF9C48E06A6C1FC1
        626D8E07E8805F75
// Get New Link
                                // ROC data in
        1487611200000000
end_memory

begin_memory exp_q4:                // cipher data out
        C3FEC9C6B72646F0
        03CB06FCF161A6AE
        1681CBEA1D894903
        310F8137A6ED5911
        B8CFEF9289B3FF73
        E0F49CC3B7F1CCDA
        AEF937BF1A84BFBA
        100BDDAB81256F9A
        8079361727EAC472
        99FBAE056270F9C3
        09E16515632B959E
        2B10D9ED544D4A00
        AC4EE556C4E419B3
        3B8D73C4EA6A72DA
        22C59150C721A2D1
        675CDB3D33CECD6D
        80BB7FF2BEF52036
        BAA19B06A5CE98EC
end_memory
```

```
begin_memory exp_q5:                // authentication code out
        92CA9003AC92CBEE
end_memory

begin_memory exp_q6:                // counter out
        404C8CF1F36DD21F
        31DEC14880B20009
end_memory
```

A.9 ARC4

```
encrypt_type : afha

begin_descriptor:
                        //   descriptor type = common_nonsnoop_afha
                        //   cipher function = rc4
                        //   direction = outbound
                        //   done notification = off
                        //   primary cha = afha

    10000050            // Header word 1
           0            // Header word 2

           0            // Nil length 0
           0            //   Extent (unused)
           0            //   Nil pointer 0

           0            // Nil length 1
           0            //   Extent (unused)
           0            //   Nil pointer 1

           a            // 2 Length of cipher key = 10
           0            //   Extent (unused)
         @p2            //   Pointer to cipher key
```

```
          89            // 3 Length of cipher data = 137
           0            //   Extent (unused)
         @p3            //   Pointer to cipher data

          89            // 4 Length of cipher ouput = 137
           0            //   Extent (unused)
         @q4            //   Pointer to cipher output

           0            // Nil length 5
           0            //   Extent (unused)
           0            //   Nil pointer 5

           0            // Nil length 6
           0            //   Extent (unused)
           0            //   Nil pointer 6
end_descriptor

begin_memory p2:                // cipher key
        3E044EA17F0D1FFC
        D4E8000000000000
end_memory

begin_memory p3:                // cipher data in
        5C88BD0ECD17D165
        C6775CA30B66EEB0
        D1E66E4ACBF44DC7
        4E3D072CC994FA0A
        66A6862E0091DCB7
        B710128B0A304AF8
        6E778076D7566FFF
        9E46A679EA99BA36
        F4EAEB34D3DF9B2D
        3667F2E0663237C9
        E8C218D1477643C0
        6C33D5645114B230
        96DA8F8003C80E1E
        F1F73A30BC9C7EBB
        371EE7BBA878D4DA
        2551F856EEE011E2
        D77AE94A347A9DCB
        E900000000000000
end_memory

begin_memory exp_q4:                    // cipher data out
        1C851D657AB80F5F
        48AFE0F1D4BF1665
        E33F566AF0BB179D
        1D00DB6D6C351B11
        F5BA3E068BB129AF
        68D14F235D0C5F9F
        64C2E87700ED2733
        EA6E32BFE974E981
        A0173ED7C283B007
        C2988A19A65BD941
        4D665BA7CDBFF8B1
        2D94731F4AD52BDF
        3B7A15488A55CABE
        1CB642C4845ED5CF
        58A4EC9709D3FD9E
        755EA685EB6D212E
        33AA4763A998CD1B
        6200000000000000
end_memory
```

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

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)
www.freescale.com/support

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

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: AN2755
Rev. 1
04/2010