Table 7-79 RSC Context (Continued)

Size	Name	Description
37 bytes		Total.
RSC Header[2]		
2 bytes	RSCIPLEN	Total *Length* field in the IP header defines the size of the IP datagram (IP header and IP payload) in bytes. Dynamic parameter updated by each received packet.
5 bits	IPOFF	The word offset of the IP header within the packet that is transferred to the DMA unit.
1 bit	RSCTS	TCP time stamp header presence indication.
1 bit	RSCACK	ACK bit in the TCP header is a dynamic parameter taken from the last coalesced packet.
1 bit	RSCACKTYPE	ACK packet type indication (ACK bit is set while packet does not has TCP payload).
2 bits	CE, ECT	ECN bits in the IP.TOS header: *CE* and *ECT.*
4 bytes	RSCSEQ	Non-RSCACKTYPE case: Expected sequence number in the TCP header of the next packet. RSCACKTYPE case: The ACK sequence number in the last good packet. Dynamic parameter updated by each received packet.
8 bytes		Total.
DMA Parameters		
7 bits	RXQUEUE	Receive queue index. This parameter is set by the first packet in the RSC and expected to be the same for all packets in the RSC.
4 bits	RSCDESC	Remaining descriptors of this context. The device initialized RSCDESC by the *MAXDESC* field in the RSCCTL register of the associated receive queue.
4 bits	RSCCNT	Count the number of packets that are started in the current descriptor. The counter starts at 0x1 for each new descriptor. RSCCNT stops incrementing when it reaches 0xF.
8 bytes	HPTR	Header buffer pointer defines the address in host memory of the large receive header (see Section 7.11.5.3).
2 bytes	DATDESC	Data descriptor is the active descriptor index. Initialized by the first packet in the RSC to the first descriptor. It is updated to the active descriptor at a packet DMA completion.
2 bytes	DATOFF	Offset within the data buffer. The data of the first packet in a large receive is the same as the legacy (non-coalescing) definition. Following a DMA completion, it points to the beginning of the data portion of the next packet.
13 bytes		Total.

1. These parameters are extracted from the first packet that opens (activate) the context.
2. All parameters are set by the first packet that opens the context while some are dynamic.

7.11.3 Processing New RSC

Defining the RSC context parameters activates a new large receive. If a received packet does not match any active RSC context, the packet starts (opens) a new one. If there is no free context, the oldest active large receive is closed and its evicted context is used for the new large receive.

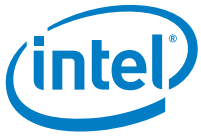
7.11.3.1 RSC Context Setting

The 82599 extracts the flow identification and RSC header parameters from the packet that opens the context (the first packet in a large receive that activates an RSC context). The context parameters can be divided into categories: flow identification; RSC header and DMA parameters.

7.11.4 Processing Active RSC

Received packets that belong to an active RSC can be added to the large receive if all the following conditions are met:

- The L2 header size equals the size of previous packets in the RSC as recorded in the internal IPOFF parameter in the RSC context table.
- The packet header length as reported in the HDR_LEN field is assumed to be the same as the first packet in the RSC (not checked by hardware).
- The ACK bit in the TCP header is 1b or equals to the RSCACK bit in the RSC context (an active RSCACK context and inactive received ACK bit is defined as no match).
- The packet type remains the same as indicated by the RSCACKTYPE bit in the RSC context. Packet type can be either ACK packet (with no TCP payload) or other.
- For non-RSCACKTYPE (packet with TCP payload): The sequence number in the TCP header matches the expected value in the RSC context (RSCSEQ).
- For RSCACKTYPE: The ACK sequence number in the TCP header is greater than the RSCSEQ number in the RSC context. Note that the 82599 does not coalesce duplicated ACK nor ACK packets that only updates the TCP window.
- ECN handling: The value of the *CE* and *ECT* bits in the IP.TOS field remains the same as the RSC context and different than 11b.
- The target receive queue matches the RXQUEUE in the RSC context.
- The packet does not include a TCP time stamp header unless it was included on the first packet that started the large receive (indicated by the RSCTS). Note that if the packet includes other option headers than time stamp, NOP or End of option header, the packet is not processed by RSC flow at all.
- The packet fits within the RSC buffer(s).
- If the received packet does not meet any of the above conditions, the matched active large receive is closed. Then hardware opens a new large receive by that packet. Note that since the 82599 closes the old large receive it is guaranteed that there is at least one free context.



If the received packet meets all the above conditions, the 82599 appends this packet to the active large receive and updates the context as follows. The packet is then DMA'ed to the RSC buffers (as described in Section 7.11.5).

- Update the TCP ACK: The RSCACK in the large receive context gets the value of the ACK bit in the TCP header in the received packet.
- Update the expected sequence number for non-RSCACKTYPE: The RSCSEQ in the large receive context is incremented by the value of the TCP payload size of the received packet.
- Update the expected sequence number for RSCACKTYPE: The RSCSEQ in the large receive context is updated to the value of the ACK sequence number field in the received packet.
- Update the total length: The RSCIPLEN in the large receive context is incremented by the value of the TCP payload size of the received packet. The value of the *Total Length* field in the IP header in the received packet gets the updated RSCIPLEN. Note that in RSCACKTYPE packets the received payload size is zero.
- Note that LinkSec encapsulation (if it exists) is stripped first by hardware. In this case, hardware also strips the Ethernet padding (if it exists).
- IP header checksum is modified to reflect the changes in the *Total Length* field as follows (note that there is no special process for RSCACKTYPE packets):

1's {(RSCIPLEN — Packet total length) + 1's (Packet IP header checksum)} while...

  - — Packet total length is the total length value in the received packet.
  - — Packet IP header checksum stands for the IP header checksum field in the received packet.
  - — 1's operation defines a ones complement.
  - — Plus (+) operation is a cyclic plus while the carry out is fed as a carry in.

- TCP header checksum is left as is in the first packet in the RSC and is set to zero on any succeeding packets.
- Update the DMA parameters.
  - — The RSCCNT is initialized to 0x1 on each new descriptor. It is then incremented by one on each packet that starts on the same descriptor as long as it does not exceed a value of 0xF. When the RSCCNT is set to 0xF (14 packets) the RSC completes.
  - — Decrement by one the Remaining Descriptors (RSCDESC) for each new descriptor.
  - — Update the receive descriptor index (DATDESC) for each new descriptor.
  - — Update the offset within the data buffer (DATOFF) at the end of the DMA to its valid value for the next packet.
- All other fields are kept as defined by the first packet in the large receive.

7.11.5 Packet DMA and Descriptor Write Back

The Figure 7-44 shows a top view of the RSC buffers using advanced receive descriptors and header split descriptors.

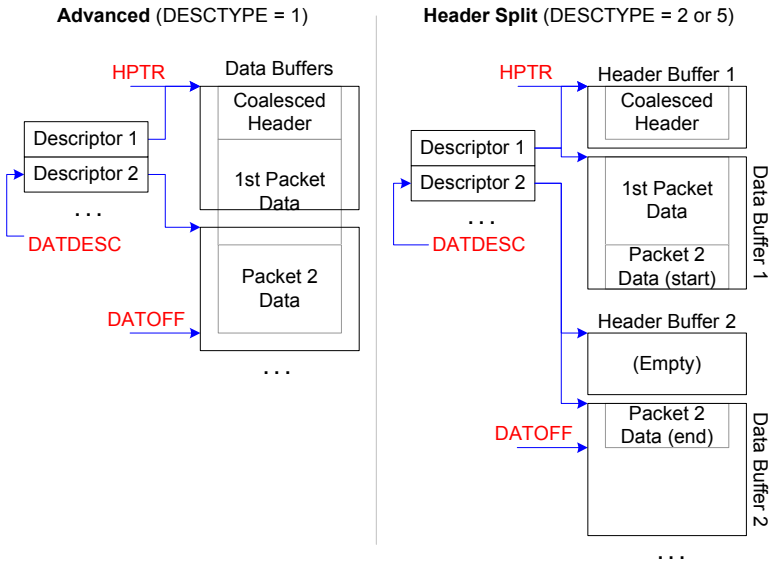
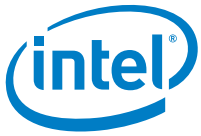


Figure 7-44 RSC — Header and Data Buffers

7.11.5.1 RSC Descriptor Indication (Write Back)

Following reception of each packet, the 82599 posts the packet data to the data buffers and updates the coalesced header in its buffer. Any completed descriptor is indicated (write back) by setting the fields listed in the following table. A descriptor is defined as the last one when an RSC completes. Section 7.11.5.1 summarizes all the causes for RSC completion. Any other descriptor in the middle of the RSC is indicated (write back) when the hardware requires the next descriptor so it can report the NEXTP field explained as follows.

Fields on the Last Descriptors of Large Receive	Fields on All Descriptors Except for the Last One
EOP: End of packet, and all other fields that are reported together with the EOP.	**NEXTP**: Points to the next descriptor of the same large receive.
DD: indicates that this descriptor is completed by the hardware and can be processed by the software.	
RSCCNT: indicates the number of coalesced packets in this descriptor.	



7.11.5.2 Received Data DMA

On the first packet of a large receive, the entire packet is posted to its buffers in host memory. On any other packet, the packet's header and data are posted to host memory as detailed in Section 7.11.5.3 and Section 7.11.5.4.

7.11.5.3 RSC Header

The RSC header is stored at the beginning of the first buffer when using advanced receive descriptors, or at the header buffer of the first descriptor when using header split descriptors. (It is defined by the internal HPTR parameter in the RSC context - see Figure 7-44).

The packet's header is posted to host memory after it is updated by the RSC context as follow:

Packets with payload coalescing (RSCACKTYPE=0) - The TCP sequence number is taken from the TCP context (it is taken from the first packet). The Total Length field in the IP header is taken from the RSC context (it represent the length of all coalesced packets). The IP checksum is re-calculated. The TCP checksum is set to zero.

ACK no payload coalescing (RSCACKTYPE=0) - The received packet header is posted as is to host memory. Note that if the received packet includes padding bytes, these bytes are discarded.

7.11.5.4 Large Receive Data

The data of a coalesced packet is posted to its buffer by the DMA engine as follows.

Ethernet CRC.

- When RSC is enabled on any queue, the global CRC strip must be set (HLREG0.RXCRCSTRP =1b).

Packet data spans on a single buffer.

- The data of the received packet spans on a single buffer if buffer has the required space.
- The DMA engine posts the packet data to its buffer pointed to by DATDESC descriptor at an offset indicated by the DATOFF.

Packet data spans on multiple buffers.

- The data of the received packet spans across multiple buffers when it is larger than a single buffer or larger than the residual size of the current buffer.
- When a new buffer is required (new descriptor) the DMA engine writes back to the completed descriptor linking it to the new one (Section 7.11.5.1 details the indicated descriptor fields).
- Decrement the RSCDESC parameter by one and update the DATDESC for each new opened descriptor.

DMA completion.

- Following DMA completion, set the DATOFF to the byte offset of the next packet.

Not enough descriptors in the receive ring buffer.

- If the SRRCTL[n].Drop_En bit on the relevant queue is set, The large receive completes and the new packet is discarded.
- Otherwise (the Drop_En bit is cleared), the packet waits inside the internal packet buffer until new descriptors are added (indicated by the relevant Tail register).

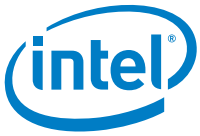
Not enough descriptors due to RSCDESC exhaust.

- If the received packet requires more descriptors than indicated by the internal RSCDESC parameter, then the 82599 completes the current large receive while the new packet starts a new large receive.

7.11.6 RSC Completion and Aging

This section summarizes all causes of large receive completion (the first three cases repeat previous sections).

- A packet of a new flow is received while there are no free RSC contexts. the 82599 completes (closes) the oldest large receive (opened first). The new packet starts a new large receive using the evicted context.
- The received packet cannot be added to the active large receive due to one of the following cases (indicated also in Section 7.11.4). In these cases the existing RSC completes and the received packet opens a new large receive.
  - The sequence number does not meet expected value.
  - The receive packet includes a time stamp TCP option header while there was no time stamp TCP option header in the first packet in the RSC.
  - There is not enough space in the RSC buffer(s) for the packet data. Meaning, the received packet requires a new buffer while the RSC already exhausted all permitted buffers defined by the RSCCTL[n].MAXDESC.
  - The received packet requires a new buffer while its descriptor wraps around the descriptor ring.
- When a packets is received while there are no more descriptors in the receive queue and the SRRCTL.Drop_En bit is set, the large receive completes and the new packet is discarded.
- EITR expiration while interrupt is enabled — RSC completion is synchronized with interrupt assertion to the host. It enables software to process the received frames since the last interrupt. See more details and EITR setting in Section 7.3.2.1.1.
- EITR expiration while interrupt is disabled — The ITR counter continues to count even when its interrupt is disabled. Every time the timer expires it triggers RSC completion on the associated Rx queues.
- LLI packet reception — All active RSC's on the same Rx queue complete and then the interrupt is asserted. Hardware then triggers RSC completion on all other queues associated with this interrupt.



- Low number of available descriptors — Whenever crossing the number of free Rx descriptors, the receive descriptor minimum threshold size defined in the SRRCTL[n] registers an LLI event is generated that affects RSC completion as well.
- Interrupt assertion by setting the EICS register has the same impact on packet reception as described in Section 7.3.1.2.1.
- Auto RSC Disable — When the interrupt logic triggers RSC completion it might also auto-disable further coalescing by clearing the RSCINT.RSCEN bit. Auto RSC disablement is controlled by the RSCINT.AUTORSC bit. In this mode, hardware also re-enables RSC (by setting the RSCINT.RSCEN bit back to 1b) when the interrupt is re-enabled by the EIMS (either by software or hardware as described in Section 7.3.1.3 and Section 7.3.1.5).

Note: In some cases packets that do not meet coalescing conditions might have active RSC of the same flow. As an example: received packets with ECE or CWR TCP flags. Such packets bypass completely the RSC logic (posted as single packets), and do not cause a completion of the active RSC. The active RSC would eventually be closed by either reception of a legitimate packet that is processed by the RSC logic but would not have the expected TCP sequence number. Or, an interrupt event closes all RSC's in its Rx queue. When software processes the packets, it gets them in order even though the RSC completes after the previous packet(s) that bypassed the RSC logic.

Any interrupt closes all RSC's on the associated receive queues. Therefore, when ITR is not enabled any receive packet causes an immediate interrupt and receive coalescing should not be enabled on the associated Rx queues.

7.12 IPsec Support

7.12.1 Overview

This section defines the hardware requirements for the IPsec offload ability included in the 82599. IPsec offload is the ability to handle (in hardware) a certain amount of the total number of IPsec flows, while the remaining are still handled by the operating system. It is the operating system's responsibility to submit to hardware the most loaded flows, in order to take maximum benefits of the IPsec offload in terms of CPU utilization savings. Establishing IPsec Security Associations between peers is outside the scope of this document, since it is handled by the operating system. In general, the requirements on the driver or on the operating system for enabling IPsec offload are not detailed here.

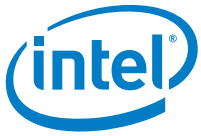
When an IPsec flow is handled in software, since the packet might be encrypted and the integrity check field already valid, and as IPv4 options might be present in the packet together with IPsec headers, the 82599 processes it like it does for any other unsupported Layer4 protocol, and without performing on it any layer4 offload.

Refer to section Section 4.6.12 for security offload enablement.

7.12.2 Hardware Features List

7.12.2.1 Main Features

- Offload IPsec for up to 1024 Security Associations (SA) for each of Tx and Rx.
  - On-chip storage for both Tx and Rx SA tables
  - Tx SA index is conveyed to hardware via Tx context descriptor
  - Deterministic Rx SA lookup according to a search key made of SPI, destination IP address, and IP version type (IPv6 or IPv4)
- IPsec protocols:
  - IP Authentication Header (AH) protocol for authentication
  - IP Encapsulating Security Payload (ESP) for authentication only
  - IP ESP for both authentication and encryption, only if using the same key for both
- Crypto engines:
  - For AH or ESP authentication only: AES-128-GMAC (128-bit key)
  - For ESP encryption and authentication: AES-128-GCM (128-bit key)
- IPsec encapsulation mode: transport mode, with tunnel mode only in receive

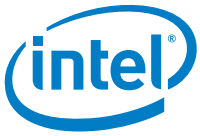


  - In Tx, packets are provided by software already encapsulated with a valid IPsec header, and
    - for AH with blank ICV inside
    - for ESP single send, with a valid ESP trailer and ESP ICV (blank ICV)
    - for ESP large send, without ESP trailer and without ESP ICV
  - In Rx, packets are provided to software encapsulated with their IPsec header and for ESP with the ESP trailer and ESP ICV,
    - where up to 255 bytes of incoming ESP padding is supported, for peers that prefer hiding the packet length
- IP versions:
  - IPv4 packets that do not include any IP option
  - IPv6 packets that do not include any extension header (other than AH/ESP extension header)
- Rx status reported to software via Rx descriptor:
  - Packet type: AH/ESP (in the SECSTAT field)
  - IPsec offload done (SA match), in the *IPSSA* field
- One Rx error reported to software via Rx descriptor in the following precedence order: no error, invalid IPsec protocol, packet length error, authentication failed (*SECERR* field)

7.12.2.2 Cross Features

- When IPSec offload is enabled, Ethernet CRC must be enabled as well by setting both TXCRCEN and RXCRCSTRP bits in the HLREG0 register
- Segmentation: full coexistence (TCP/UDP packets only)
  - increment IPsec Sequence Number (SN) and Initialization Vector (IV) on each additional segment
- Checksum offload: full coexistence (Tx and Rx)
  - IP header checksum
  - TCP/UDP checksum
- IP fragmentation: no IPsec offload done on IP fragments
- RSS: full coexistence, hash on the same fields used without IPsec (either 4-tuples or 2-tuples)
- LinkSec offload:
  - A device interface is operated in either LinkSec offload or IPsec offload mode, but not both of them altogether
  - If both IPsec and LinkSec encapsulations are required on the same packets, the 82599's interface is operated in LinkSec offload mode, while IPsec is performed by the operating system

- Virtualization:
  - Full coexistence in VMDq mode
  - in IOV mode, all IPsec registers are owned by the VMM/PF. For example, IPsec can be used for VMotion traffic.
  - No coexistence with VM-to-VM switch, IPsec packets handled in hardware are not looped back by the 82599 to another VM. Tx IPsec packets destined to a local VM must be handled in software and looped back via the software switch. However, an anti-spoofing check is performed on any IPsec packet.
- DCB: full coexistence
  - Priority flow control, with special care to respect timing considerations
  - Bandwidth allocation scheme enforced on IPsec packets since 802.1p field is always sent in clear text
  - CM-tagging takes place at Layer2 and then does not interfere with IPsec
- FCoE: no interaction as FCoE packets are not IP packets
- RSC: no coexistence
- Jumbo frames: When the SECTXCTRL.STORE_FORWARD bit is set (as required for IPSec offload), the maximum supported jumbo packet size is 9.5 KB (9728 bytes). This limitation is valid for all packets regardless if they are offloaded by hardware or carry IPSec encapsulation altogether.
- 802.1x: no interaction
- Teaming: no interaction
- TimeSync:
  - TimeSync IEEE 1588v1 UDP packets must not be encapsulated as IPsec packets
  - No interaction with TimeSync 1588v2 Layer2 packets
- Layer2 encapsulation modes:
  - IPsec offload is not supported for flows with SNAP header
  - IPsec offload coexists with double VLAN encapsulations
- Tunneled IPsec packets in receive: IPsec offload supported, but no other Layer4 offload performed
- NFS and any other Layer5 Rx filter: NFS or Layer5 packets encapsulated over ESP (whether IPsec is offloaded in hardware or not) and over a Layer4 protocol other than TCP are not parsed, nor recognized
- SCTP Rx offload: partial coexistence with SCTP CRC32 offload for IPSec-AH packets only.
- SCTP Tx offload: full coexistence with SCTP CRC32 offload for both IPSec-AH and IPSec-ESP packets.
- Manageability traffic: IPsec offload ability is controlled exclusively by the host, and thus manageability traffic could use IPsec offload only if it is coordinated/configured with/by the host. For IPsec flows handled by software:
  - If manageability and host entities share some IP address(es), then manageability should coordinate any use of IPsec protocol with the host. Note it should be true for previous devices that do not offer IPsec offload.



  - If manageability and host entities have totally separate IP addresses, then manageability can use IPsec protocol (as long as it is handled by the manageability controller software)
- Header split:
  - Supported for SAs handled in hardware, IP boundary split includes the IPsec header
  - For SAs handled in software, no header split done

7.12.3 Software/Hardware Demarcation

The followings items are not supported by hardware but might be supported by operating system/driver:

- Multicast SAs
- IPsec protocols:
  - Both AH and ESP protocols on the same SA or packet
  - ESP for encryption only
  - ESP for both authentication and encryption using different keys and/or different crypto engines
- Crypto engines:
  - AES-256, SHA-1, AES-128-CBC, or any other crypto algorithm
- Tx IPsec packets encapsulated in tunnel mode
- Extended Sequence Number (ESN)
- IP versions:
  - IPv4 packets that include IP option
  - IPv6 packets that include extension headers other than the AH/ESP extension headers
- Anti-replay check and discard of incoming replayed packets
- Discard of incoming dummy ESP packets (packets with protocol value 59)
- IPsec packets that are IP fragments
- ESP padding content check
- IPsec statistics
- IPsec for flows with SNAP header

Note: For SCTP and other Layer4 header types, or for tunneled packets, hardware does not care what is there when doing Rx IPsec processing. Everything after the IP/IPsec headers can be opaque to hardware (consider as IP payload). IPsec processing can be done on any packet that has a matching SA and appropriate IP options/extension headers. There is no expectation that hardware determines what is in the packet beyond the IP/IPsec headers before decrypting/authenticating the packet. The most important point is that hardware should not corrupt or drop incoming IPsec packets —

in any situation. When hardware decides and starts performing IPsec offload on a packet, it should pursue the offload until the packet's end — at the price of eventually not doing other Layer3/4 offloads on it. It is always acceptable for hardware not to start doing the IPsec offload on a matched SA, if it knows it is an unsupported encapsulation. For example, one of the three cases: IPv4 option, IPv6 extensions, or SNAP.

7.12.4 IPsec Formats Exchanged Between Hardware and Software

This section describes the IPsec packet encapsulation formats used between software and hardware by an IPsec packet concerned with the offload in either Tx or Rx direction.

In Rx direction, the IPsec packets are delivered by hardware to software encapsulated as they were received from the line, whether IPsec offload was done or not, and when it was done, whether authentication/decrypting has succeeded or failed.

7.12.4.1 Single Send

In Tx direction, single-send IPsec packets are delivered by software to hardware already encapsulated and formatted with their valid IPsec header and trailer contents, as they should be run over the wire — except for the *ICV* field that is filled with zeros, and the ESP payload destined to be encrypted that is provided in clear text before IPsec encryption.

7.12.4.2 Single Send with TCP/UDP Checksum Offload

For single-send ESP packets with TCP/UDP checksum offload, the checksum computing includes the TCP/UDP header and payload before hardware encryption occurred and without the ESP trailer and ESP ICV provided by software. Software provides the length of the ESP trailer plus ESP ICV in a dedicated field of the Tx context descriptor (IPS_ESP_LEN field) to signal hardware when to stop TCP/UDP checksum computing.

Software calculates a full checksum for the IP pseudo-header as in the usual case. The protocol value used in the IP pseudo-header must be the TCP/UDP protocol value and not the AH/ESP protocol value that appears in the IP header. This full checksum of the pseudo-header is placed in the packet data buffer at the appropriate offset for the checksum calculation.

The byte offset from the start of the DMA'ed data to the first byte to be included in the TCP/UDP checksum (the start of the TCP header) is computed as in the usual case: MACLEN+IPLEN. It assumes that IPLEN provided by software in the Tx context descriptor is the sum of the IP header length with the IPsec header length.

Note: For the IPv4 header checksum offload, hardware cannot rely on the *IPLEN* field provided by software in the Tx context descriptor, but should rely on the fact that no IPv4 options are present in the packet. Consequently, for IPsec offload packets, hardware always computes IP header checksum over a fixed amount of 20 bytes.

7.12.4.3 TSO TCP/UDP

In Tx direction, TSO IPsec packets are delivered by software to the 82599 already encapsulated and formatted with only their valid IPsec header contents — except for the *ICV* field included in AH packets headers that is filled with zeros, and to the ESP payload destined to be encrypted that is provided in clear text before any encryption. No ESP trailer or ESP ICV are appended to TSO by software. It means that hardware has to append the ESP trailer and ESP ICV on each segment by itself, and to update IP total length / IP payload length accordingly.

The next header of the ESP trailer to be appended by hardware is taken from TUCMD.L4T field of the Tx context descriptor.

By definition TSO requires on each segment that the IP total length / IP payload length be updated, and the IP header checksum and TCP/UDP checksum be re-computed. But for the TSO of IPsec packets, the *SN* and the *IV* fields must be increased by one in hardware on each new segment (after the first one) as well.

Software calculates a partial checksum for the IP pseudo-header as in the usual case. The protocol value used in the IP pseudo-header must be the TCP/UDP protocol value and not the AH/ESP protocol value that appears in the IP header. This partial checksum of the pseudo header is placed in the packet data buffer at the appropriate offset for the checksum calculation.

The byte offset from the start of the DMA'ed data to the first byte to be included in the TCP checksum (the start of the TCP/UDP header) is computed as in the usual case: MACLEN+IPLEN. It assumes that IPLEN provided by software in the Tx context descriptor is the sum of the IP header length with the IPsec header length.

For TSO ESP packets, the TCP/UDP checksum computing includes the TCP/UDP header and payload before hardware encryption occurred and without the ESP trailer and ESP ICV appended by hardware. The 82599 thus stops TCP/UDP checksum computing after the amount of bytes given by L4LEN + MSS. It is assumed that the MSS value placed by software in the Tx context descriptor specifies the maximum TCP/UDP payload segment sent per frame, not including any IPsec header or trailer — and not including the TCP/UDP header.

Note: For IPv4 header checksum computing, refer to the note in section Section 7.12.4.2.

Shaded fields in the tables that follow correspond to fields that need to be updated per each segment.

Table 7-80 IPv4 TSO ESP Packet Provided by Software

	0 3	4 7	8 15	16 19	23 24 31
1	Ver	Hlen	TOS	IP Total Length	
2	Identification			Flags	Fragment Offset
3	TTL		Protocol = ESP	Header Checksum	
4	Source IPv4 Address				
5	Destination IPv4 Address				
1	Security Parameter Index (SPI)				
2	Sequence Number (SN)				
3	Initialization Vector (IV)				
4					
1	TCP/UDP Header				
	TCP/UDP Payload				

Table 7-81 IPv6 TSO ESP Packet Provided by Software

	0 3	4 7	8 15	16 23	24 31
1	Ver	Priority	Flow Label		
2	IP Payload Length			Next Header = ESP	Hop Limit
3	Source IPv6 Address				
4					
5					
6					
7	Destination IPv6 Address				
8					
9					
10					
1	Security Parameter Index (SPI)				
2	Sequence Number (SN)				
3	Initialization Vector (IV)				
4					
1	TCP/UDP Header				
	TCP/UDP Payload				

7.12.5 TX SA Table

IPsec offload is supported only via advanced transmit descriptors. See Section 7.2.3.2.4 for details.

7.12.5.1 Tx SA Table Structure

The Tx SA table contains additional information required by the AES-128 crypto engine to authenticate and encrypt data. This information is not run over the wire together with the IPsec packets, but is exchanged between the IPsec peers' operating system during the SA establishment process. When the IKE software does a key computation it computes four extra bytes using a pseudo-random function (it generates 20 bytes instead of 16 bytes that it needs to use as a key) and the last four bytes are used as a salt value.

The SA table in Tx is a 1024 x 20-byte table loaded by software. Each line in the table contains the following fields:

Table 7-82 TX SA Table

AES-128 KEY	AES-128 SALT
16 bytes	4 bytes

Refer to Section 7.12.7 for a description of the way these fields are used by the AES-128 crypto engine.

Each time an unrecoverable memory error occurs when accessing the Tx SA tables, an interrupt is generated and the transmit path is stopped until the host resets the 82599. Packets that have already started to be transmitted on the wire are sent with a wrong CRC.

Upon reset, the 82599 clears the contents of the Tx SA table. Note that access to Tx SA table is not guaranteed for 10 µs after the reset command.

7.12.5.2 Access to Tx SA Table

7.12.5.2.1 Write Access

1. Software writes the IPSTXKEY 0...3 and/or IPSTXSALT register(s).
2. Software writes the IPSTXIDX register with the SA_IDX field carrying the index of the SA entry to be written, and with the *Write* bit set (*Read* bit cleared).
3. Hardware issues a Write command into the SA table, copying the IPSTXKEY (16 bytes) and the IPSTXSALT (4 bytes) registers into the table entry pointed by the SA_IDX field configured in IPSTXIDX register. It then clears the *Write* bit in IPSTXIDX register.

4. Software starts/resumes sending IPsec offload packets with the *IPsec SA IDX* field in the Tx context descriptor pointing to valid/invalid SA entries. A valid SA entry contains updated key and salt fields currently in use by the IPsec peers.

7.12.5.2.2 Read Access

1. Software writes the IPSTXIDX register with the index of the SA entry to be read, and with the *Read* bit set (*Write* bit cleared).
2. Hardware issues a Read command from the SA table, copying into the registers the IPSTXKEY (16 bytes) and the IPSTXSALT (4 bytes) values from the table entry pointed by the SA_IDX field configured in the IPSTXIDX register. It then clears the *Read* bit in IPSTXIDX register.
3. Software reads the IPSTXKEY 0...3 and/or IPSTXSALT register(s).

7.12.6 TX Hardware Flow

7.12.6.1 Single Send without TCP/UDP Checksum Offload

1. Extract IPsec offload request from the IPSEC bit of the POPTS field in the advanced Tx transmit data descriptor.
2. If IPsec offload is required for the packet (IPSEC bit was set), then extract the *SA_IDX*, *Encryption*, and *IPSEC_TYPE* fields from the Tx context descriptor associated to that flow.
3. Fetch the AES-128 KEY and Salt from the Tx SA entry indexed by SA_IDX and according to the *Encryption* and *IPSEC_TYPE* bits to determine which IPsec offload to perform.
4. For AH, zero the mutable fields.
5. Compute ICV and encryption data (if required for ESP) over the appropriate fields, according to the operating rules described in Section 7.12.7, and making use of the AES-128 KEY and Salt fields fetched in step 3.
6. Insert ICV at its appropriate location and replace the plaintext with the ciphertext (if required for ESP).

7.12.6.2 Single Send with TCP/UDP Checksum Offload

1. Extract the IPsec offload command from the *IPSEC* bit of the *POPTS* field in the advanced Tx transmit data descriptor.
2. If IPsec offload is required for the packet (*IPSEC* bit was set), then extract the *SA_IDX*, *Encryption*, *IPSEC_TYPE*, and *IPS_ESP_LEN* fields from the Tx context descriptor associated to that flow.

3. Fetch the AES-128 KEY and Salt from the Tx SA entry indexed by SA_IDX, and according to the *Encryption* and *IPSEC_TYPE* bits to determine which IPsec offload to perform.
4. Compute the byte offset from the start of the DMA'ed data to the first byte to be included in the checksum (the start of the TCP header) as specified in Section 7.12.4.2.
5. Compute TCP/UDP checksum until either the last byte of the DMA data or for ESP packets, up to IPS_ESP_LEN bytes before it. As in the usual case, *implicitly* pad out the data by one zeroed byte if its length is an odd number.
6. Sum the full checksum of the IP pseudo header placed by software at its appropriate location with the TCP/UDP checksum computed in step 5. Overwrite the checksum location with the 1's complement of the sum.
7. For AH, zero the mutable fields.
8. Compute ICV and encrypt data (if required for ESP) over the appropriate fields, according to the operating rules described in Section 7.12.7, and making use of the AES-128 KEY and Salt fields fetched in step 3.
9. Insert ICV at its appropriate location and replace the plaintext with the ciphertext (if required for ESP).

7.12.6.3 TSO TCP/UDP

1. Extract the IPsec offload command from the *IPSEC* bit of the *POPTS* field in the advanced Tx transmit data descriptor.
2. If IPsec offload is required for the packet (*IPSEC* bit was set), then extract the *SA_IDX*, *Encryption*, and *IPSEC_TYPE* fields from the Tx context descriptor associated to that flow.
3. Fetch the AES-128 KEY and Salt from the Tx SA entry indexed by SA_IDX, and according to the *Encryption* and *IPSEC_TYPE* bits to determine which IPsec offload to perform.
4. Fetch the packet header from system memory, up to IPLEN+L4LEN bytes from the start of the DMA'ed data.
5. Overwrite the TCP SN with the stored accumulated TCP SN (if it is not the first segment).
6. Fetch (next) MSS bytes (or the remaining bytes up to PAYLEN for the last segment) from system memory and from the segment formed by packet header and data bytes, while storing the accumulated TCP SN.
7. Compute the byte offset from the start of the DMA'ed data to the first byte to be included in the checksum (the start of the TCP header) as specified in Section 7.12.4.3.
8. Compute TCP/UDP checksum until the last byte of the DMA data. As in the usual case, *implicitly* pad out the data by one zeroed byte if its length is an odd number.

9. For both IPv4 and IPv6, hardware needs to factor in the TCP/UDP length (typically L4LEN+MSS) to the software-supplied pseudo header partial checksum. It then sums to obtain a full checksum of the IP pseudo header with the TCP/UDP checksum computed in step 7. Overwrite the TCP/UDP checksum location with the 1's complement of the sum.
10. Increment by one the *AH/ESP SN* and *IV* fields on every segment (excepted to the first segment), and store the updated *SN* and *IV* fields with other temporary statuses stored for that TSO (one
TSO set of statuses per Tx queue).
11. For ESP, append the ESP trailer: 0-3 padding bytes, padding length, and next header = TCP/UDP protocol value, in a way to get the 4-byte alignment as described in Section 7.12.4.3.
12. Compute the IP total length / IP payload length and compute IPv4 header checksum as described in the note of Section 7.12.4.1. Place the results in their appropriate location.
13. For AH, zero the mutable fields.
14. Compute ICV and encryption data (if required for ESP) over the appropriate fields, according to the operating rules described in Section 7.12.7, and making use of the AES-128 KEY and Salt field fetched in step 3.
15. Insert ICV at its appropriate location and replace the plaintext with the ciphertext (if required for ESP).
16. Go back to step 4 to process the next segment (if necessary).

7.12.7 AES-128 Operation in Tx

The AES-128-GCM crypto engine used for IPsec is the same AES-128-GCM crypto engine used for LinkSec. It is referred throughout the document as an AES-128 black box, with 4-bit string inputs and 2-bit string outputs, as shown in Figure 7-45. Refer to the GCM specification for the internal details of the engine. The difference between IPsec and LinkSec, and between the different IPsec modes reside in the set of inputs presented to the box.

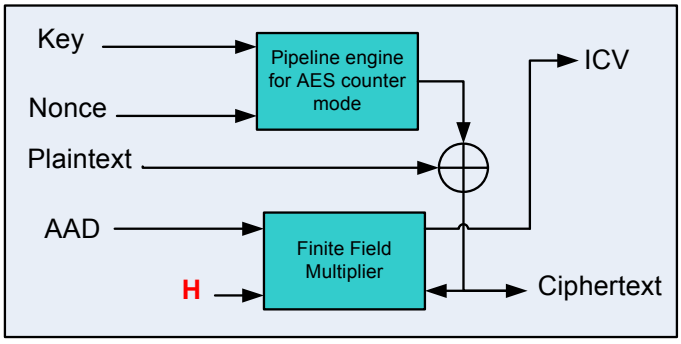


Figure 7-45 AES-128 Crypto Engine Box

- Key — 128-bits AES-128 KEY field (secret key) stored for that IPsec flow in the Tx SA table:

  ```
  Key = AES-128 KEY
  ```

- Nonce — 96-bits initialization vector used by the AES-128 engine, which is distinct for each invocation of the encryption operation for a fixed key. It is formed by the AES-128 Salt field stored for that IPsec flow in the Tx SA table, appended with the Initialization Vector (IV) field included in the IPsec packet:

  ```
  Nonce = [AES-128 SALT, IV]
  ```

  The nonce, also confusingly referred as IV in the GCM specification, is broken into two pieces — a fixed random part salt and increasing counter part IV, so the salt value goes with the packet as the fixed part. The purpose behind using the salt value is to prevent offline dictionary-type attacks in hashing case, to prevent predictable patterns in the hash.

- AAD — Additional Authentication Data input, which is authenticated data that must be left un-encrypted.
- Plaintext — Data to be both authenticated and encrypted.
- Ciphertext — Encrypted data, whose length is exactly that of the plaintext.
- ICV — 128-bit Integrity Check Value (referred also as authentication tag).
- H — is internally derived from the key.

Note: The square brackets in the formulas is used as a notation for concatenated fields.

7.12.7.1 AES-128-GCM for ESP — Both Authenticate and Encryption

```
AAD = [SPI, SN]
Plaintext = [TCP/UDP header, TCP/UDP payload, ESP trailer]
```

Note: Unlike other IPsec modes, in this mode, the *IV* field is used only in the nonce, and it is not included in either the plaintext or the AAD inputs.

ESP trailer does not include the *ICV* field.

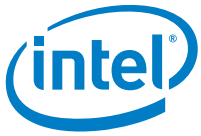
7.12.7.2 AES-128-GMAC for ESP — Authenticate Only

```
AAD = [SPI, SN, IV, TCP/UDP header, TCP/UDP payload, ESP trailer]
Plaintext = [] = empty string, no plaintext input in this mode
```

Note: ESP trailer does not include the *ICV* field.

7.12.7.3 AES-128-GMAC for AH — Authenticate Only

```
AAD = [IP header, AH header, TCP/UDP header, TCP/UDP payload]
Plaintext = []= empty string, no plaintext input in this mode
```



Note: Both IP header and AH header contain mutable fields that must be zeroed prior to be entered into the engine. Among other fields, the AH header includes *SPI*, *SN*, and *IV* fields.

7.12.8 RX Descriptors

IPsec offload is supported only via advanced receive descriptors. See Section 7.1.6 for details.

7.12.9 Rx SA Tables

7.12.9.1 Rx SA Tables Structure

The Rx SA tables contain additional information required by the AES-128 crypto engine to authenticate and decrypt the data. This information is not run over the wire together with the IPsec packets, but is exchanged between the IPsec peers' operating system during the SA establishment process. When the IKE software does a key computation it computes four extra bytes using a pseudo-random function (it generates 20 bytes instead of 16 bytes that it needs to use as a key) and the last four bytes are used as a salt value.

SPI is allocated by the receiving operating system in a unique manner. However, in a virtualized context, guest operating systems can allocate SPI values that collide with the SPI values allocated by the VMM/PF. Consequently, the SPI search must be completed by comparing the destination IP address with the IP addresses of the VMM/PF, which are stored in a separate table. Guest operating systems could thus use the proposed IPsec offload as long as their SAs are configured via the VMM/PF. It is assumed that refreshing the SAs would be done once every several minutes, and would thus not overload the VMM/PF.

There are three Rx SA tables in the 82599:

- IP address table — 128 entries
- SPI table — 1K entries
- KEY table — 1K entries

They are loaded by software via indirectly addressed CSRs, as described in Section 7.12.9.2.

Table 7-83 IP Address Table

IP Address
16 bytes

Table 7-84 SPI Table

SPI	IP Index (points to IP address table)
4 bytes	1 bytes

Table 7-85 KEY Table

IPsec Mode	AES-128 KEY	AES-128 SALT
1 byte	16 bytes	4 bytes

The *IPsec Mode* field contains the following bits:

- VALID
- IPv6
- PROTO
- DECRYPT

It is assumed that the SPI and IP address tables are implemented internally in CAM cells, while the KEY table uses RAM cells. When an incoming IPsec packet (which does not includes option in IPv4 or another extension header in IPv6) is detected, hardware first looks up for the destination IP address to match one of the IP addresses stored in the IP address table. If there is a match, the index of that IP Addr. match is used together with the *SPI* field extracted from the packet for a second lookup into the SPI table. If there is again a match, then the index of that SPI+IP Index match is used to retrieve the SA parameters from the KEY table. The packet is finally considered to get an SA match only after inspecting the corresponding entry in the KEY table, as long as all the following conditions are met:

- *Valid* bit is set
- *IPv6* bit match with the IP version (IPv6/IPv4) of the incoming IPsec packet
- *Proto* bit match with the AH/ESP type of the incoming IPsec packet

Each time an unrecoverable memory ECC error occurs when accessing one of the Rx SA tables, an interrupt is generated and the receive path is stopped until the host resets the 82599.

Upon reset, the 82599 clears the contents of the Rx KEY table and software is required to invalidate the entire IP address and SPI CAM tables by clearing their contents. Access to Rx SA tables is not guaranteed for 10 μs after the Reset command.

7.12.9.2 Access to Rx SA Tables

7.12.9.2.1 Write Access

1. Software writes the IP address table via the IPSRXIPADDR 0...3 registers
2. Software writes the IPSRXIDX register with the following:
   a. *Table* bits combination corresponding to the Rx SA table to be written (such as 01b for IP address table)
   b. TB_IDX field pointing to the index to be written within the table
   c. *Write* bit set (*Read* bit cleared)
3. Hardware issues a Write command into the Rx SA table pointed by the Table bits combination, copying the concerned register(s) into the entry pointed by the *TB_IDX* field configured in the IPSRXIDX register. It then clears the *Write* bit in IPSRXIDX register.
4. Software performs steps 1 to 3 twice, first for writing the SPI table via IPSRXSPI, IPSRXIPIDX registers, and second for writing the KEY table via IPSRXKEY 0...3, IPSRXSALT, and IPSRXMOD registers.

Each time an entry in the IP address or SPI table is not valid/in-use anymore, software is required to invalidate its content by clearing it. For the IP table, an entry must be invalidated by software each time there is no more SPI entry that points to it; while for the SPI table, software must invalidate any entry as soon as it is not valid/not used anymore.

7.12.9.2.2 Read Access

1. Software writes the IPSRXIDX register with the *Table* and *TB_IDX* fields corresponding to the Rx SA table and entry to be read, and with the *Read* bit set (*Write* bit cleared).
2. Hardware issues a Read command from the Rx SA table and entry pointed by *Table* bits combination and *TB_IDX* field, copying each field into its corresponding register. It then clears the *Read* bit in IPSRXIDX register.
3. Software reads the corresponding register(s).

Caution: There is an internal limitation in that only one single Rx SA table can be read accessed by software at a time. Hence, it is recommended that the entire read process, from steps 1 to 3, be repeated successively for each Rx SA table separately.

7.12.10 RX Hardware Flow without TCP/UDP Checksum Offload

1. Detect an IPsec header not encapsulated over a SNAP header is present without any IPv4 option or other IPv6 extension header encapsulated before it, and determine its type AH/ESP.
2. If such an IPsec header is present (as announced by the IP protocol field for IPv4 or by the next header for IPv6), then extract the SPI, destination IP address, and IP version (IPv4 or IPv6), and use these fields for the lookups into the Rx SA tables as described in Section 7.12.9.1. Also report the IPsec protocol found in the *Security* bits of the *Extended Status* field in the advanced Rx descriptor.
3. If there is a SA match for that packet, fetch the IPsec Rx mode from the SA entry, and according to the *Proto* and *Decrypt* bits determine which IPsec offload to perform. Also, set the *IPSSA* bit of the *Extended Status* field in the advanced Rx descriptor. If there was no SA match, then clear the *IPSSA* bit, report no error in *Security* error bits of the *Extended Errors* field in the advanced Rx descriptor, and stop processing the packet for IPsec.
4. If the *Proto* field recorded in the Rx SA table does not match the *IP Protocol* field (next header for IPv6) seen in the packet, then report it via the *Security* error bits of the *Extended Errors* field in the advanced Rx descriptor, and stop processing the packet for IPsec.
5. Fetch the AES-128 KEY and Salt from the matched Rx SA entry.
6. For AH, zero the mutable fields.
7. Make sure the AH/ESP header is not truncated, and for ESP make sure whether or not the packet is 4-bytes aligned. If not, report it via the *Security* error bits of the *Extended Errors* field in the advanced Rx descriptor, but processing of the packet for IPsec might be completed (if it has already started). A truncated IPsec packet is a valid Ethernet frame (at least 64 bytes) shorter than:
   a. ESP – at least 40 bytes following the IP header (16 [ESP header] + 4 [min. padding, pad_len, NH] + 16 [ICV] + 4 [CRC])
   b. AH over IPv4 – at least 40 bytes following the IP header (20 [AH header] + 16 [ICV] + 4 [CRC])
   c. AH over IPv6 – at least 44 bytes following the IP header (20 [AH header] + 4 [ICV padding] + 16 [ICV] + 4 [CRC])
8. Compute ICV and decrypt data (if required for ESP) over the appropriate fields, according to the operating rules described in Section 7.12.12, and making use of the AES-128 KEY and SALT fields fetched in step 5.
9. Compare the computed ICV with the *ICV* field included in the packet at its appropriate location, and report the comparison status match/fail via the *Security* error bits of the Extended Errors field in the advanced Rx descriptor.

7.12.11 RX Hardware Flow with TCP/UDP Checksum Offload

Perform the RX hardware flow described in Section 7.12.10 and add the following steps:

10. Start computing the checksum from the TCP/UDP header's beginning — found according to the Rx parser logic updated for IPsec formats. Do not perform Layer4 offloads if unsupported IPsec encapsulation is detected. For example, tunneled IPsec, IPv4 options or IPv6 extensions after the IPsec header.
11. For ESP, stop checksum computing before the beginning of the ESP trailer — found from the end of packet according to the padding length field content. As in the usual case, *implicitly* pad out the data by one zeroed byte if its length is an odd number.
12. Store the next header extracted from the AH header/ESP trailer into the *Packet Type* field of the advanced Rx descriptor, but use the TCP/UDP protocol value in the IP pseudo header used for the TCP/UDP checksum. Also compute the TCP/UDP packet length to be inserted in the IP pseudo header (excluding any IPsec header or trailer).
13. Compare the computed checksum value with the TCP/UDP checksum included in the packet. Report the comparison status in the Extended Errors field of the advanced Rx descriptor.

7.12.12 AES-128 Operation in Rx

The AES-128 operation in Rx is similar to the operation in Tx, while for decryption, the encrypted payload is fed into the plaintext input, and the resulted ciphertext stands for the decrypted payload. Refer to Section 7.12.7 for the proper inputs to use in every IPsec mode.

7.12.12.1 Handling IPsec Packets in Rx

The following table lists how IPsec packets are handled according to some of their characteristics.

Table 7-86 Summary of IPsec Packets Handling in Rx

IP Fragment	IPv4 Option or IPv6 Extensions or SNAP	IP Version	SA Match	IPsec Offload in Hardware	Layer4/3 Offload in Hardware	Header Split	AH/ESP Reported in Rx Desc.
Yes	Yes	v4	Don't care	No	IP checksum only	Up to IPsec header included	Yes
Yes	Yes	v6	Don't care	No	No	Up to IP fragment extension included	No
Yes	No	v4	Don't care	No	IP checksum only	Up to IPsec header included	Yes
No	Yes	v4	Don't care	No	IP checksum only	No	Yes

Table 7-86 Summary of IPsec Packets Handling in Rx (Continued)

IP Fragment	IPv4 Option or IPv6 Extensions or SNAP	IP Version	SA Match	IPsec Offload in Hardware	Layer4/3 Offload in Hardware	Header Split	AH/ESP Reported in Rx Desc.
No	Yes	v6	Don't care	No	No	Up to first unknown or IPsec extension header, excluded	No[1]
No	No	v4	Yes	Yes	Yes[2]	Yes[3]	Yes
No	No	v4	No	No	IP checksum only	No	Yes
No	No	v6	Yes	Yes	Yes[4]	Yes[3]	Yes
No	No	v6	No	No	No	No	Yes

1. Exception to SNAP IPsec packets that are reported as AH/ESP in Rx descriptor.
2. No Layer4 offload done on packets with IPSec error.
3. According to definition made in PSRTYPE[n] registers.
4. No Layer4 offload done on packets with IPSec error.