7.2.4.5 TCP and UDP Segmentation Indication

Software indicates a TCP / UDP segmentation transmission context to the hardware by setting up a TCP/IP or UDP/IP context transmit descriptor (see Section 7.2.3). The purpose of this descriptor is to provide information to the hardware to be used during the TCP / UDP segmentation offload process.

Setting the *TSE* bit in the DCMD field to one (in the data descriptor) indicates that this descriptor refers to the segmentation context (as opposed to the normal checksum offloading context). This causes the checksum offloading, packet length, header length, and maximum segment size parameters to be loaded from the descriptor into the device.

The TCP / UDP segmentation prototype header is taken from the packet data itself. Software must identity the type of packet that is being sent (IPv4/IPv6, TCP/UDP, other), calculate appropriate checksum off loading values for the desired checksums, and then calculate the length of the header that is prepended. The header can be up to 240 bytes in length.

Once the TCP / UDP segmentation context has been set, the next descriptor provides the initial data to transfer. This first descriptor(s) must point to a packet of the type indicated. Furthermore, the data it points to might need to be modified by software as it serves as the prototype header for all packets within the TCP / UDP segmentation context. The following sections describe the supported packet types and the various updates that are performed by hardware. This should be used as a guide to determine what must be modified in the original packet header to make it a suitable prototype header.

The following summarizes the fields considered by the driver for modification in constructing the prototype header.

IP Header

For IPv4 headers:

- Identification field should be set as appropriate for first packet of send (if not already).
- Header checksum should be zeroed out unless some adjustment is needed by the driver.

TCP Header

- Sequence number should be set as appropriate for first packet of send (if not already).
- PSH, and FIN flags should be set as appropriate for LAST packet of send.
- TCP checksum should be set to the partial pseudo-header checksum as follows (there is a more detailed discussion of this in Section 7.2.4.6:

Table 7-43 TCP Partial Pseudo-header Checksum for IPv4

IP Source Address		
IP Destination Address		
Zero	Layer 4 Protocol ID	Zero

Table 7-44 TCP Partial Pseudo-header Checksum for IPv6

IPv6 Source Address	
IPv6 Final Destination Address	
Zero	
Zero	Next Header

UDP Header

- Checksum should be set as in TCP header, as previously explained.

The following sections describe the updating process performed by the hardware for each frame sent using the TCP segmentation capability.

7.2.4.6 Transmit Checksum Offloading with TCP and UDP Segmentation

The 82599 supports checksum offloading as a component of the TCP / UDP segmentation off-load feature and as stand-alone capability. Section 7.2.5 describes the interface for controlling the checksum off-loading feature. This section describes the feature as it relates to TCP / UDP segmentation.

The 82599 supports IP and TCP header options in the checksum computation for packets that are derived from the TCP segmentation feature.

Two specific types of checksum are supported by the hardware in the context of the TCP/ UDP segmentation off-load feature:

- IPv4 checksum
- TCP / UDP checksum

Each packet that is sent via the TCP / UDP segmentation off-load feature optionally includes the IPv4 checksum and/or the TCP / UDP checksum.

All checksum calculations use a 16-bit wide one's complement checksum. The checksum word is calculated on the outgoing data.

Table 7-45 Supported Transmit Checksum Capabilities

Packet Type	HW IP Checksum Calculation	HW TCP / UDP Checksum Calculation
IPv4 packets	Yes	Yes
IPv6 packets (no IP checksum in IPpv6)	NA	Yes
Packet has 802.3ac tag	Yes	Yes
Packet has IP options (IP header is longer than 20 bytes)	Yes	Yes

Table 7-45 Supported Transmit Checksum Capabilities (Continued)

Packet Type	HW IP Checksum Calculation	HW TCP / UDP Checksum Calculation
Packet has TCP options	Yes	Yes
IP header's protocol field contains a protocol # other than TCP or UDP	Yes	No

7.2.4.7 IP/TCP / UDP Header Updating

IP/TCP and IP/UDP header is updated for each outgoing frame based on the header prototype that hardware DMA's from the first descriptor(s). The checksum fields and other header information are later updated on a frame-by-frame basis. The updating process is performed concurrently with the packet data fetch.

The following sections define what fields are modified by hardware during the TCP / UDP segmentation process by the 82599.

7.2.4.7.1 TCP/IP/UDP Header for the First Frame

The hardware makes the following changes to the headers of the first packet that is derived from each TCP segmentation context.

MAC Header (for SNAP)

- Type/Len field = MSS + MACLEN + IPLEN + L4LEN — 14

IPv4 Header

- IP Total Length = MSS + L4LEN + IPLEN
- Calculates the IP Checksum

IPv6 Header

- Payload Length = MSS + L4LEN + IPV6_HDR_extension[1]

TCP Header

- Sequence Number: The value is the sequence number of the first TCP byte in this frame.
- The flag values of the first frame are set by logic AND function between the flag word in the pseudo header and the DTXTCPFLGL.TCP_flg_first_seg. The default values of the DTXTCPFLGL.TCP_flg_first_seg are set. The flags in a TSO that ends up as a single segment are taken from the in the pseudo header in the Tx data buffers as is.
- Calculates the TCP checksum.

UDP Header

- Calculates the UDP checksum.

1. IPV6_HDR_extension is calculated as IPLEN — 40 bytes.

7.2.4.7.2 TCP/IP Header for the Subsequent Frames

The hardware makes the following changes to the headers for subsequent packets that are derived as part of a TCP segmentation context:

Number of bytes left for transmission = PAYLEN — (N * MSS). Where N is the number of frames that have been transmitted.

MAC Header (for SNAP packets)

Type/Len field = MSS + MACLEN + IPLEN + L4LEN — 14

IPv4 Header

- IP Identification: incremented from last value (wrap around)
- IP Total Length = MSS + L4LEN + IPLEN
- Calculate the IP Checksum

IPv6 Header

- Payload Length = MSS + L4LEN + IPV6_HDR_extension[1]

TCP Header

- Sequence Number update: Add previous TCP payload size to the previous sequence number value. This is equivalent to adding the MSS to the previous sequence number.
- The flag values of the subsequent frames are set by logic AND function between the flag word in the pseudo header with the DTXTCPFLGL.TCP_Flg_mid_seg. The default values of the DTXTCPFLGL.TCP_Flg_mid_seg are set.
- Calculate the TCP checksum

UDP Header

- Calculates the UDP checksum.

7.2.4.7.3 TCP/IP Header for the Last Frame

Hardware makes the following changes to the headers for the last frame of a TCP segmentation context:

Last frame payload bytes = PAYLEN — (N * MSS).

MAC Header (for SNAP packets)

- Type/Len field = Last frame payload bytes + MACLEN + IPLEN + L4LEN — 14

IPv4 Header

- IP Total length = last frame payload bytes + L4LEN + IPLEN
- IP identification: incremented from last value (wrap around based on 16-bit width)
- Calculate the IP checksum

IPv6 Header

- Payload length = last frame payload bytes + L4LEN + IPV6_HDR_extension[1]

TCP Header

- Sequence number update: Add previous TCP payload size to the previous sequence number value. This is equivalent to adding the MSS to the previous sequence number.
- The flag values of the last frames are set by logic AND function between the flag word in the pseudo header and the DTXTCPFLGH.TCP_Flg_lst_seg. The default values of the DTXTCPFLGH.TCP_Flg_lst_seg are set. The flags in a TSO that ends up as a single segment are taken from the in the pseudo header in the Tx data buffers as is.
- Calculate the TCP checksum

UDP Header

- Calculates the UDP checksum.

7.2.5 Transmit Checksum Offloading in Non-segmentation Mode

The previous section on TCP / UDP segmentation offload describes the IP/TCP/UDP checksum offloading mechanism used in conjunction with segmentation. The same underlying mechanism can also be applied as a stand-alone checksum offloading. The main difference in a single packet send is that only the checksum fields in the IP/TCP/UDP headers are calculated and updated by hardware.

Before taking advantage of the 82599's enhanced checksum offload capability, a checksum context must be initialized. For a single packet send, DCMD.TSE should be set to zero (in the data descriptor). For additional details on contexts, refer to Section 7.2.3.3.

Enabling checksum offload, software must also enable Ethernet CRC offload by the HLREG0.TXCRCEN since CRC must be inserted by hardware after the checksum has been calculated.

As mentioned in Section 7.2.3, transmit descriptors, it is not necessary to set a new context for each new packet. In many cases, the same checksum context can be used for a majority of the packet stream. In this case, some performance can be gained by only changing the context on an as needed basis or electing to use the off-load feature only for a particular traffic type, thereby avoiding all context descriptors except for the initial one.

Each checksum operates independently. Insertion of the IP and TCP / UDP checksum for each packet are enabled through the transmit data descriptor POPTS.TXSM and POPTS.IXSM fields, respectively.

7.2.5.1 IP Checksum

Three fields in the transmit context descriptor set the context of the IP checksum offloading feature:

- TUCMD.IPV4
- IPLEN
- MACLEN

TUCMD.IPV4=1 specifies that the packet type for this context is IPv4, and that the IP header checksum should be inserted. TUCMD.IPV4=0 indicates that the packet type is IPv6 (or some other protocol) and that the IP header checksum should not be inserted.

MACLEN specifies 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 IP header. The minimal allowed value for this field is 14. Note that the maximum value for this field is 127. This is adequate for typical applications.

Note: The MACLEN+IPLEN value must be less than the total DMA length for a packet. If this is not the case, the results are unpredictable.

IPLEN specifies the IP header length. Maximum allowed value for this field is 511 bytes.

MACLEN+IPLEN specify where the IP checksum should stop. The sum of MACLEN+IPLEN must be smaller equals to the first 638 (127+511) bytes of the packet and obviously must be smaller or equal to the total length of a given packet. If this is not the case, the result is unpredictable.

Note: For IPsec packets offloaded by hardware in Tx, it is assumed that IPLEN provided by software in the Tx context descriptor is the sum of the IP header length and the IPsec header length. Thus, for the IPv4 header checksum offload, hardware could no longer rely on the *IPLEN* field provided by software in the Tx context descriptor, but should rely on the fact that no IPv4 options is present in the packet. Consequently, for IPsec offload packets, hardware computes IP header checksum over a fixed amount of 20 bytes.

For IP tunnel packets (IPv4-IPv6), IPLEN must be defined as the length of the two IP headers. Hardware is able to offload the L4 checksum calculation while software should provide the IPv4 checksum.

The 16-bit IPv4 header checksum is placed at the two bytes starting at MACLEN+10.

As mentioned in Section 7.2.3.2.3, transmit contexts, it is not necessary to set a new context for each new packet. In many cases, the same checksum context can be used for a majority of the packet stream. In this case, some performance can be gained by only changing the context on an as needed basis or electing to use the off-load feature only for a particular traffic type, thereby avoiding all context descriptors except for the initial one.

7.2.5.2 TCP and UDP Checksum

Three fields in the transmit context descriptor set the context of the TCP / UDP checksum offloading feature:

- MACLEN
- IPLEN
- TUCMD.L4T

TUCMD.L4T=01b specifies that the packet type is TCP, and that the 16-bit TCP header checksum should be inserted at byte offset MACLEN+IPLEN+16. TUCMD.L4T=00b indicates that the packet is UDP and that the 16-bit checksum should be inserted starting at byte offset MACLEN+IPLEN+6.

MACLEN+IPLEN specifies 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 UDP/TCP header. See MACLEN table in Section 7.2.3.2.3 for its relevant values.

Note: The MACLEN+IPLEN+L4LEN value must be less than the total DMA length for a packet. If this is not the case, the results are unpredictable.

The TCP/UDP checksum always continues to the last byte of the DMA data.

Note: For non-TSO, software still needs to calculate a full checksum for the TCP/UDP pseudo-header. This checksum of the pseudo-header should be placed in the packet data buffer at the appropriate offset for the checksum calculation.

7.2.5.3 SCTP Transmit Offload

For SCTP packets, a CRC32 checksum offload is provided.

Three fields in the transmit context descriptor set the context of the STCP checksum offloading feature:

- MACLEN
- IPLEN
- TUCMD.L4T

TUCMD.L4T=10b specifies that the packet type is SCTP, and that the 32-bit STCP CRC should be inserted at byte offset MACLEN+IPLEN+8.

IPLEN+MACLEN specifies 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 STCP header. The minimal allowed value for this sum is 26.

The SCTP CRC calculation always continues to the last byte of the DMA data.

The SCTP total L3 payload size (PAYLEN - IPLEN - MACLEN) should be a multiple of four bytes (SCTP padding not supported).

Note: TSO is not available for SCTP packets.

Software must initialize the SCTP CRC field to zero (0x00000000).

7.2.5.4 Checksum Supported per Packet Types

The following table lists which checksums are supported per packet type.

Note: TSO is not supported for packet types for which IP checksum and TCP / UDP checksum cannot be calculated.

7.3 Interrupts

The 82599 supports the following interrupt modes. Mapping of interrupts causes is different in each of these modes as described in this section.

- PCI legacy interrupts or MSI or MSI-X and only a single vector is allocated — selected when GPIE.Multiple_MSIX is set to 0b.
- MSI-X with multiple MSI-X vectors in non-IOV mode — selected when GPIE.Multiple_MSIX is set to 1b and GPIE.VT_Mode is set to 00b.
- MSI-X in IOV mode — selected when GPIE.Multiple_MSIX is set (as previously stated) and GPIE.VT_Mode DOES NOT equal 00b.

The following sections describe the interrupt registers and device functionality at all operation modes.

7.3.1 Interrupt Registers

Physical Function (PF) Registers

The PF interrupt logic consists of the registers listed in the Table 7-46 followed by their description:

Table 7-46 PF Interrupt Registers

Acronym	Complete Name
EICR	Extended Interrupt Cause register
EICS	Extended Interrupt Cause Set register (enables software to initiate interrupts)
EIMS	Extended Interrupt Mask Set/Read register
EIMC	Extended Interrupt Mask Clear register
EIAC	Extended Interrupt Auto Clear register (following interrupt assertion)
EIAM	Extended Interrupt Auto Mask register (auto set/clear of the EIMS)
EITR	Extended Interrupt Throttling register [throttling and Low Latency Interrupt (LLI) setting]
IVAR	Interrupt Vector Allocation Registers (described in Section 7.3.4)
IVAR_MISC	Miscellaneous Interrupt Vector Allocation Register (described in Section 7.3.4)

These registers are extended to 64 bits by an additional set of two registers. EICR has an additional two registers EICR(1)... EICR(2) and so on for the EICS, EIMS, EIMC, EIAM and EITR registers. The EIAC register is not extended to 64 bits as this extended interrupt causes are always auto cleared. Any reference to EICR... EIAM registers as well as any global interrupt settings in the GPIE register relates to their extended size of 64 bits.

The legacy EICR[15:0] mirror the content of EICR(1)[15:0]. In the same manner the lower 16 bits of EICS, EIMS, EIMC, EIAC, EIAM mirror the lower 16 bits of EICS(1), EIMS(1), EIMC(1), EIAM(1). For more details on the use of these registers in the various interrupt modes (Legacy, MSI, MSI-X) see Section 7.3.4.

Virtual Function (VF) Registers

The VF interrupt logic has the same set of interrupt registers while each of them has three entries for three interrupt causes. The names and functionality of these registers are the same as those of the PF with a prefix of VT as follows: VFEICR, VFEICS, VFEIMS, VFEIMC, VFEIAM, VFEITR. The VFEIAC registers are not supported since interrupt causes are always auto cleared. Although each VF can generate up to three interrupts, only the first two registers are capable of interrupt throttling and are associated to VFEITR registers (see Section 7.3.4.3.2 for its proper usage). Each VF also has the mapping registers VFIVAR and VFIVAR_MISC. Note that any global interrupt setting by the GPIE register affect both interrupt settings of the PF as well as the VFs.

7.3.1.1 Extended Interrupt Cause (EICR) Registers

This register records the interrupt causes to provide software information on the interrupt source. Each time an interrupt cause happens, the corresponding interrupt bit is set in the EICR registers. An interrupt is generated each time one of the bits in these registers is set, and the corresponding interrupt is enabled via the EIMS registers. The possible interrupt causes are as follows:

- Each *RTxQ* bit represents the following events: Tx or Rx descriptor write back; Rx queue full and Rx descriptor queue minimum threshold.
    - The *RTxQ* interrupts can be throttled by ITR or LLI as configured in the EITR register (LLI does not impact Tx). Following interrupt assertion, software cannot distinguish between ITR or LLI events.
    - Mapping the Tx and Rx queues to EICR is done by the IVAR registers as described in Section 7.3.4. Each bit might represent an event on a single Tx or Rx queue or could represent multiple queues according to the IVAR setting. In the later case, software might not be able to distinguish between the interrupt causes other than checking all associated Tx and Rx queues.
    - The Multiple_MSIX = 1b setting is useful when multiple MSI-X vectors are assigned to the device. When the GPIE.Multiple_MSIX bit is set, the *RTxQ* bits are associated with dedicated MSI-X vectors. Bit 0 is Tx / Rx interrupt associated with MSI-X vector 0 and bit 15 is Tx / Rx interrupt associated with MSI-X vector 15.
- Bits 29:16 in the EICR are named in the EAS as the “other” interrupt causes. Please refer to the EICR register definition for the exact interrupt causes included in this group. All these causes are mapped to the same interrupt even in Multiple_MSIX mode. In Multiple_MSIX mode the “other” interrupt causes are mapped to a specific MSI-X vector by the INT_Alloc[1] in the IVAR_MISC register.
- Bit 30 in the EICR register is the “TCP Timer” interrupt usually used to wake the SW driver periodically according to the TCPTIMER setting. In Multiple_MSIX mode the “TCP Timer” interrupt is mapped to a specific MSI-X vector by the INT_Alloc[0] in the IVAR_MISC register.

Writing a 1b to any bit in the register clears it. Writing a 0b to any bit has no effect. The EICR is also cleared on read if GPIE.OCD bit is cleared. When the GPIE.OCD bit is set, then only bits 16...29 are cleared on read. The later setting is useful for MSI-X mode in which the Tx and Rx and possibly the timer interrupts do not share the same interrupt with the other causes. Bits in the register can be auto cleared depending on the EIAC register setting (detailed in Section 7.3.1.4).

7.3.1.2 Extended Interrupt Cause Set (EICS) Register

This register enables software to initiate a hardware interrupt. Setting any bit on the EICS sets its corresponding bit in the EICR register while bits written to 0b have no impact. It then causes an interrupt assertion if enabled by the EIMS register. Setting any bit generates either LLI or throttled interrupt depending on the GPIE.EIMEN setting: When the *EIMEN* bit is set, then setting the EICS register causes an LLI interrupt; When the *EIMEN* bit is cleared, then setting the EICS register causes an interrupt after the corresponding interrupt throttling timer expires.

Note: The *EIMEN* bit can be set high only when working in auto-mask mode (*EIAM* bit of the associated interrupt is set).

7.3.1.2.1 EICS Affect on RSC Functionality

Setting *EICS* bits causes interrupt assertion (if enabled). EICS settings have the same impact on RSC functionality as nominal operation:

- In ITR mode (GPIE.EIMEN = 0b), setting the *EICS* bits impact the RSC completion and interrupt assertion the same as any Rx packet. The functionality depends on the EICS setting schedule relative to the ITR intervals as described in Section 7.3.2.1.1.
- In LLI mode (GPIE.EIMEN = 1b), setting the *EICS* bits impact the RSC completion and interrupt assertion the same as any LLI Rx packet. Device behavior is described in Section 7.3.2.2.3 starting with the 2nd step.

7.3.1.3 Extended Interrupt Mask Set and Read (EIMS) Register, and Extended Interrupt Mask Clear (EIMC) Register

The Extended Interrupt Mask Set and Read (EIMS) register enables the interrupts in the EICR. When set to 1b, each bit in the EIMS register, enables its corresponding bit in the EICR. Software might enable each interrupt by setting bits in the EIMS register to 1b. Reading EIMS returns its value. Software might clear any bit in the EIMS register by setting its corresponding bit in the Extended Interrupt Mask Clear (EIMC) register. Reading the EIMC register does not return any meaningful data.

This independent mechanism of setting and clearing bits in the EIMS register saves the need for read modify write and also enables simple programming in multi-thread, multi-CPU core systems.

Note: The EICR register stores the interrupt events regardless of the state of the EIMS register.

7.3.1.4 Extended Interrupt Auto Clear Enable (EIAC) Register

Each bit in this register enables auto clearing of its corresponding bit in EICR following interrupt assertion. It is useful for Tx and Rx interrupt causes that have dedicated MSI-X vectors. When the Tx and Rx interrupt causes share an interrupt with the other or a timer interrupt, the relevant EIAC bits should not be set. Bits in the EICR register that are not enabled by auto clear, must be cleared by either writing a 1b to clear or a read to clear.

Note that there are no EIAC(1)...EIAC(2) registers. The hardware setting for interrupts 16...63 is always auto clear.

Note: Bits 29:16 should never be set to auto clear since they share the same MSI-X vector.

Writing to the EIAC register changes the setting of the entire register. In IOV mode, some of the bits in this register might affect VF functionality (VF-56...VF-63). It is recommended that software set the register in PF before VF's are enabled. Otherwise, a software semaphore might be required between the VF and the PF to avoid setting corruption.

7.3.1.5 Extended Interrupt Auto Mask Enable (EIAM) Register

Each bit in this register enables auto clearing and auto setting of its corresponding bit in the EIMS register as follows:

- Following a write of 1b to any bit in the EICS register (interrupt cause set), its corresponding bit in the EIMS register is auto set as well enabling its interrupt.
- A write to clear the EICR register clears its corresponding bits in the EIMS register masking further interrupts.
- A read to clear the EICR register, clears the *EIMS* bits (enabled by the EIAM) masking further interrupts. Note that if the GPIE.OCD bit is set, Tx and Rx interrupt causes are not cleared on read (bits 0:15 in the EICR). In this case, bits 0:15 in the EIMS are not cleared as well.
- In MSI-X mode the, auto clear functionality can be driven by MSI-X vector assertion if GPIE.EIAME is set.

Note: Bits 29:16 should never be set to auto clear since they share the same MSI-X vector.

Writing to the EIAM register changes the setting of the entire register. In IOV mode, some of the bits in this register might affect VF functionality. It is recommended that software set the register in PF before VF's are enabled. Otherwise, a software semaphore might be required between the VF and the PF to avoid setting corruption.

If any of the Auto Mask enable bits is set in the EIAM registers, the GPIE.EIAME bit must be set as well.

7.3.2 Interrupt Moderation

Interrupt rates can be tuned by the EITR register for reduced CPU utilization while minimizing CPU latency. In MSI or legacy interrupt modes, only EITR register 0 can be used. In MSI-X, non-IOV mode, the 82599 includes 64 EITR registers 0...63 that are mapped to MSI-X vectors 0...63, respectively. In IOV mode, there are an additional 65 EITR registers that are mapped to the MSI-X vectors of the virtual functions. The mapping of MSI-X vectors to EITR registers are described in Section 7.3.1.1.

The EITR registers include two types of throttling mechanisms: ITR and LLI. Both are described in the sections that follow.

7.3.2.1 Time-based Interrupt Throttling — ITR

Time-based interrupt throttling is useful to limit the maximum interrupt rate regardless of network traffic conditions. The ITR logic is targeted for Rx/Tx interrupts only. It is assumed that the timer, other and mail box (IOV mode) interrupts are not moderated. In non-IOV mode, all 64 interrupts can be associated with ITR logic. In IOV mode, the ITR logic is shared between the PF and VFs as shown in Figure 7-21. The ITR mechanism is based on the following parameters:

- ITR Interval field in the EITR registers — The minimum inter-interrupt interval is specified in 2 μs units (at 1 Gb/s or 10 Gb/s link). When the ITR Interval equals zero, interrupt throttling is disabled and any event causes an immediate interrupt. The field is composed of nine bits enabling a range of 2 μs up to 1024 μs. These ITR interval times correspond to interrupt rates in the range of 500 K INT/sec to 980 INT/sec. When operating at 100 Mb/s link, the ITR interval is specified in 20 μs units.
  - — Due to internal synchronization issues, the ITR interval can be shortened by up to 1 μs at
10 Gb/s or 1 Gb/s link and up to 10 μs at 100 Mb/s link when it is triggered by packet write back or interrupt enablement or the last interrupt was LLI.
- ITR Counter partially exposed in the EITR registers — Down counter that is loaded by the ITR interval each time the associated interrupt is asserted.
  - — The counter is decremented by one each 2 μs (at 1 Gb/s or 10 Gb/s link) and stops decrementing at zero. At 100 Mb/s link, the speed of the counter is decremented by one each 20 μs.
  - — If an event happens before the counter is zero, it sets the EICR. The interrupt can be asserted only when the ITR time expires (counter is zero).
  - — Else (no events during the entire ITR interval), the EICR register is not set and the interrupt is not asserted on ITR expiration. The next event sets the EICR bit and generates an immediate interrupt. See Section 7.3.2.1.1 for interrupt assertion when RSC is enabled.
  - — Once the interrupt is asserted, the ITR counter is loaded by the ITR interval and the entire cycle re-starts. The next interrupt can be generated only after the ITR counter expires once again.

7.3.2.1.1 ITR Affect on RSC Functionality

Interrupt assertion is one of the causes for RSC completion (see Section 7.11.6). When RSC is enabled on specific Rx queues, the associated ITR interval with these queues must be enabled and must be larger (in time units) than RSC delay. The ITR is divided to the two time intervals that are defined by the ITR interval and RSC delay. RSC completion is triggered after the first interval completes and the interrupt is asserted when the second interval completes.

The RSC_DELAY field is defined in the GPIE registers. *RSC Delay* can have one of the following eight values: 4 µs, 8 µs, 12 µs... 32 µs.

- The first ITR interval equals ITR interval minus RSC delay. The internal ITR counter starts at ITR interval value and counts down until it reaches the RSC delay value. Therefore, the ITR interval must be set to a larger value than the RSC delay.
- The second ITR interval equals RSC delay. The internal ITR counter continues to count down until it reaches zero.
- RSC completion can take some time (usually in the range of a few micro seconds). This time is composed by completing triggering latency and completing process latency. These delays should be considered when tuning the RSC delay. The clock frequency of the RSC completion logic depends on the link speed. As a result, the completion delay can as high as ~0.8 µs at 10 Gb/s link and ~8 µs at 1 Gb/s link. The RSC completion logic might take additional ~50 ns at 10 Gb/s link and ~0.5 µs at 1 Gb/s link per RSC. In addition, there is the PCIe bus arbitration latency as well as system propagation latencies from the device up to host memory.
- Recommended RSC delay numbers are: 8 µs at 10 Gb/s link and 28 µs at 1 Gb/s link.
- RSC is not recommended when operating at 100 Mb/s link.

Following are cases of packet reception with respect to the ITR intervals:

- Packets are received and posted (including their status) to the Rx queue in the first ITR interval. In this case, RSC completion is triggered at the end of the first ITR interval and the interrupt is asserted at the second interval expiration.
- a packet (and its status) is received and posted to the Rx queue only after the first ITR interval has expired (either on the second interval or after the entire ITR interval has expired). In this case, RSC completion is triggered almost instantly (other than internal logic latencies). The interrupt is asserted at RSC delay time after the non-coalesced Rx status is queued to be posted to the host.
- Due to internal synchronization issues, the RSC delay can be shorten by up to 1 µs when it is triggered by packet write back.

7.3.2.2 LLI

LLI provides low latency service for specific packet types, bypassing the ITR latency. LLIs are bound by a credit-based throttling mechanism that limits the maximum rate of low latency events that require a fast CPU response. Low latency events are triggered by the write back of the LLI packets. It then generates an immediate interrupt if LLI credits are not exhausted. See more details on the credit mechanism in the Section 7.3.2.2.2. Note also that in the case of RSC, the interrupt is not immediate as described in Section 7.3.2.2.3.

7.3.2.2.1 LLI Filters and Other Cases

Following is a list of all Rx packets that are defined as low latency events (LLI packets):

- LLI by 5-tuple / TCP flags / frame size — The 82599 supports a set of 128 filters that initiate LLI by a 5-tuple value and frame size. An LLI is issued if any of the filters are set for LLI matches against the enabled fields of 5-tuple, TCP flags, and frame size. Configuration is done via the FTQF, SDPQF, L34TIMIR, DAQF, and SAQF registers as follows per filter (more details about these filters can be found in Section 7.1.2.5). Note that if a packet matches multiple 5-tuple filters an LLI is initiated if it is enabled by any of the matched filters:
- 5-tuple fields (protocol, IP address, port) and mask options for these fields
- Pool and pool mask
- SizeThresh — A frame with a length below this threshold triggers an interrupt. Unlike other fields, the *SizeThresh* field is shared by all filters (like there is a single copy of it). Matching the frame size is enabled by the *Size_BP* bit.
- *Size_BP* bit, when set to 0b, equates to a match that is performed against the frame size.
- LLI field — When set, an LLI is issued for packets that match the filter.
- LLI by Ethertype — The 82599 supports eight Ethertype filters. Any filter has an LLI action defined by the LLI field in the ETQS registers.
- LLI by VLAN priority — The 82599 supports VLAN priority filtering as defined in the IMIRVP register. Packets with VLAN header that have higher priority tagging than the one defined by IMIRVP register generates an LLI.
- LLI by FCoE — FCoE FCP_RSP packets can trigger LLI as defined in the FCRXCTRL.RSCINT bit. The 82599 identifies FCoE packets by the Ethertype filters defined by the ETQF registers. FCP_RSP packets recognition is explained in Section 7.13.3.3.10.

The 82599 might initiate an LLI when the receive descriptor ring is almost empty (Rx descriptors below a specific threshold). The threshold is defined by SRRCTL[n].RDMTS per Rx queue. This mechanism can protect against memory resources being used up during reception of a long burst of short packets.

7.3.2.2.2 LLI Parameters

LLI generation is based on the following parameters:

- LLI Moderation bit in the EITR registers — When the LLI Moderation bit is cleared, any low latency event generates an immediate interrupt. When set, LLI moderation is based on the LLI credit and LLI interval.
- LLI Credit field in the EITR register — LLI packets might generate immediate interrupts as long as the LLI credits counter is greater than one (positive credit).
    - — The credit counter is incremented by one on each LL interval with a maximum ceiling of 31 credits. It then stops incrementing.

  - If an LLI packet is received and the credit counter is greater than zero, an immediate interrupt is triggered internally. The interrupt is asserted externally when an interrupt is enabled (EIMS setting) and PCI credits are available. Once the interrupt is asserted, the credit counter is decremented by one Note that the counter never goes below zero.
  - LLI assertion might be delayed due to: interrupt enablement, lack of LLI credits or lack of PCI credits. Each time the interrupt is asserted, the LLI credit is decremented by one regardless of the number of received LLI packets and regardless if the ITR timer expires in the mean time.
  - If an LLI packet is received and the credit counter is zero (no credits), an interrupt can be asserted only on the next LL interval or when the ITR timer expires, whichever comes first.
  - The LLI credit counter is not affected by the ITR timer. Conversely, LLI assertion initializes the ITR timer to its timer interval.
  - Note that during nominal operation software may not need to access the LL credit field.
- LL interval is defined in units of 4 ms (at 1 Gb/s or 10 Gb/s link) in the GPIE register. At 100 Mb/s link speed, the LL interval is defined in units of 40 ms. This parameter defines the clock that increments the LLI credit counter. The maximum rate of the LLI interrupts per second is bound by the LL interval, which equals to 1/LL Interval. When LLI moderation is enabled, the ITR interval of the same interrupt must be greater than the LL interval.

7.3.2.2.3 LLIs Affect on RSC Functionality

LLI packet reception requires instant CPU processing. Software might be able to access a specific descriptor only if all its proceeding descriptors complete. If RSC's are enabled, some of the preceding descriptors might be incomplete at the time that the LLI packet is received. Hardware overcomes this problem by:

- Following LLI packet completion, all RSC's on the same queue are completed as well.
- Then, the associated interrupt is asserted.
- Concurrently, hardware triggers RSC completion in all Rx queues associated with the same interrupt.
- Most likely these RSC(s) are completed to host memory after the interrupt is already asserted. In this case, it is guaranteed that an additional interrupt is asserted when the ITR expires.

7.3.3 TCP Timer Interrupt

7.3.3.1 Introduction

In order to implement TCP timers for IOAT, software needs to take action periodically (every 10 ms). Today, the driver must rely on software-based timers, whose granularity can change from platform to platform. This software timer generates a software NIC interrupt, which then enables the driver to perform timer functions, avoiding cache thrash and enabling parallelism. The timer interval is system-specific.

It would be more accurate and more efficient for this periodic timer to be implemented in hardware. The driver would program a timeout value (usual value of 10 ms), and each time the timer expires, hardware sets a specific bit in the EICR register. When an interrupt occurs (due to normal interrupt moderation schemes), software reads the EICR register and discovers that it needs to process timer events.

The timeout should be programmable by the driver, and the driver should be able to disable the timer interrupt if it is not needed.

7.3.3.2 Description

A stand-alone, down-counter is implemented. An interrupt is issued each time the value of the counter is zero.

Software is responsible for setting an initial value for the timer in the *Duration* field. Kick-starting is done by writing a 1b to the *KickStart* bit.

Following kick starting, an internal counter is set to the value defined by the *Duration* field. Then the counter is decreased by one each ms. When the counter reaches zero, an interrupt is issued. The counter re-starts counting from its initial value if the *Loop* field is set.

7.3.4 Mapping of Interrupt Causes

The following sections describe legacy, MSI and MSI-X interrupt modes.

7.3.4.1 Legacy and MSI Interrupt Modes

In legacy and MSI modes, an interrupt cause is reflected by setting one of the bits in the EICR register, where each bit reflects one or more causes. All interrupt causes are mapped to a single interrupt signal: either legacy INTA/B or MSI. This section describes the mapping of interrupt causes (that is a specific Rx or Tx queue event or any other event) to bits in the EICR.

The TCP timer and all other interrupt causes are mapped directly to EICR[30:16]. Note that the IVAR_MISC register is not used in legacy and MSI modes.

Mapping the Tx and Rx queues to interrupt bits in the EICR register is programmed in the IVAR registers as shown in Figure 7-21. Each entry in the IVAR registers is composed of two fields that identify the associated bit in the EICR[15:0] register. Software might map multiple Tx and Rx queues to the same EICR bit.

INT_Alloc - Defines one of the bits (0...15) in the EICR register that reflects the interrupt status indication.

INT_Alloc_val - Valid bit for this interrupt cause.

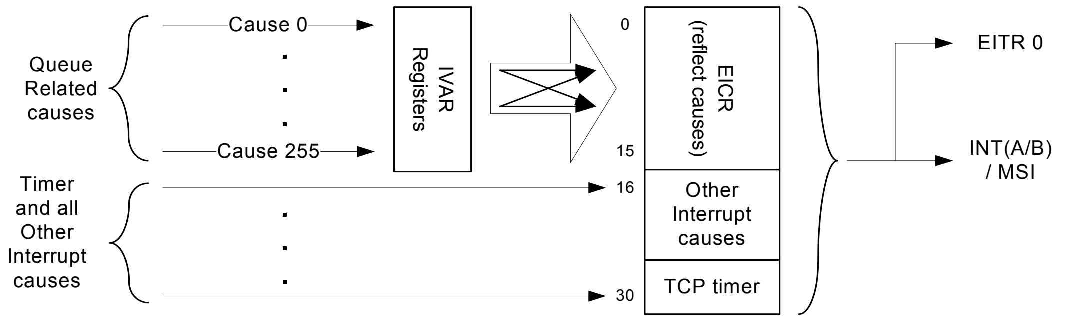

Figure 7-21 Cause Mapping in Legacy and MSI Modes

Mapping between the Tx and Rx queue to the IVAR registers is hardwired as shown in the Figure 7-22 below:

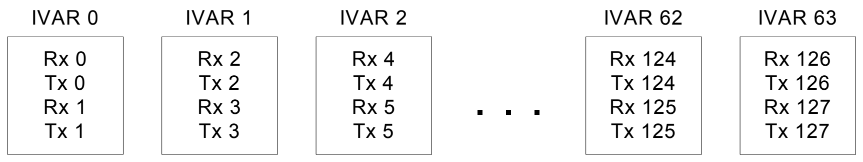

Figure 7-22 Rx and Tx Queue Mapping to IVAR Registers

7.3.4.2 MSI-X Mode in Non-IOV Mode

MSI-X defines a separate optional extension to basic MSI functionality. The number of requested MSI-X vectors is loaded from the MSI_X_N fields in the EEPROM up to maximum of 64 MSI-X vectors.

- Hardware indicates the number of requested MSI-X vectors in the table size in the MSI-X capability structure in the configuration space. This parameter is loaded from the *MSI_X _N* field in the EEPROM. The operating system might allocate any number of MSI-X vectors to the device from a minimum of one up to the requested number of MSI-X vectors.
- Enables interrupts causes allocation to the assigned MSI-X vectors. Interrupt allocation is programmed by the IVAR registers and are described in this section.
- Each vector can use an independent address and data value as programmed directly by the operating system in the MSI-X vector table.
- Each MSI-X vector is associated to an EITR register with the same index (MSI-X 0 to EITR[0], MSI-X 1 to EITR[1],...).

For more information on MSI-X, refer to the PCI Local Bus Specification, Revision 3.0.

MSI-X vectors can be used for several purposes:

1. Dedicated MSI-X vectors per interrupt cause (avoids the need to read the interrupt cause register).
2. Load balancing by MSI-X vectors assignment to different CPUs.
3. Optimized interrupt moderation schemes per MSI-X vector using the EITR registers.

The MSI-X vectors are used for Tx and Rx interrupt causes as well as the other and timer interrupt causes. The remainder of this section describes the mapping of interrupt causes (such as a specific Rx or Tx queue event or any other event) to the interrupts registers and the MSI-X vectors.

The TCP timer and other events are reflected in EICR[30:16] the same as the legacy and MSI mode. It is then mapped to the MSI-X vectors by the IVAR_MISC register as shown in Figure 7-23. The IVAR_MISC register includes two entries for the timer interrupt and an additional entry for all the other causes. The structure of each entry is as follows:

INT_Alloc - Defines the MSI-X vector (0...63) assigned to this interrupt cause.

INT_Alloc_val - Valid bit for the this interrupt cause.

The Tx and Rx queues are associated to the IVAR0...IVAR63 the same as legacy and MSI mode shown in Figure 7-22. The Tx and Rx queues are mapped by the IVAR registers to EICR(1),...EICR(2) registers and MSI-X vectors 0...63 illustrated in Figure 7-23. The IVAR entries have the same structure as the IVAR_MISC register previously shown. Each bit in EICR(1...2) registers is associated to MSI-X vector 0...63 as follows:

- EICR(i).bit_num is associated to MSI-X vector (n x 32 + bit_num).
- The legacy EICR[15:0] mirror the content of EICR(1)[15:0]. In the same manner the lower 16 bits of EICS, EIMS, EIMC, EIAC, EIAM mirror the lower 16 bits of EICS(1), EIMS(1), EIMC(1), EIAC(1), EIAM(1). The use of these registers depends on the number of assigned MSI-X interrupts as follows:
- 16 Tx and Rx Interrupts - When using up to 16 Tx and Rx interrupts, software might access the Tx and Rx interrupt bits in the legacy EICR, EICS,... registers.
- More than 16 Tx and Rx Interrupts - When using more then 16 Tx and Rx interrupts, software must use EICS(1)...EICS(2), EIMS(1)...EIMS(2),... In the later case, software should avoid modifying the lower 16 bits in the SEIC, EICS... registers when it accesses the higher bits of these registers as follows:
  - EICR, EICS, EIMS and EIMC — When software programs the higher 16 bits of these registers, it should set their lower 16 bits to zero’s keeping the EICR(1), EICS(1), EIMS(1) and EIMC(1) unaffected.
  - EIAM — When software programs the higher 16 bits, it should keep the lower 16 bits at their previous setting so the EIAM(1) is unaffected.
  - EIAC — When software programs the higher 16 bits, it should set the lower 16 bits to one’s.

Single MSI-X vector - If the operating system allocates only a single MSI-X vector, the driver might use the non-MSI-X mapping method (setting the GPIE.Multiple_MSIX to 0b). In this case, the INT_Alloc field in the IVAR registers might define one of the lower 16 bits in the EICR register while using MSI-X vector 0. The IVAR_MISC should be programmed to MSI-X vector 0.

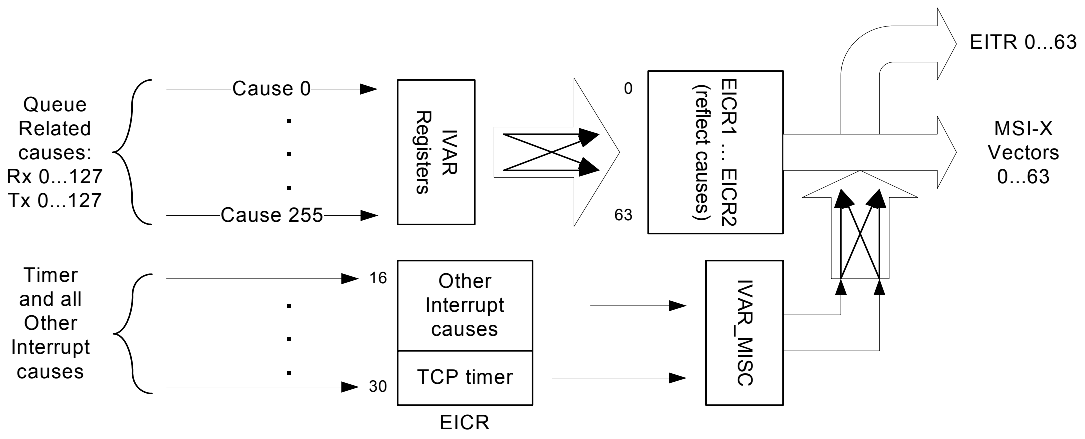

Figure 7-23 Cause Mapping in MSI-X Mode (non-IOV)

7.3.4.3 MSI-X Interrupts In IOV Mode

In IOV mode, interrupts must be implemented by MSI-X vectors. The 82599 supports up to 64 virtual functions VF(0...63). Each VF can generate up to three MSI-X vectors. The number of requested MSI-X vectors per VF is loaded from the *MSI-X Table* field in the EEPROM. It is reflected in the *Table Size* field in the PCIe MSI-X capability structure of the VF's. In addition, the PF requires its own interrupts. The number of requested MSI-X vectors for the PF is loaded from the *MSI_X_N* fields in the EEPROM up to maximum of 64 MSI-X vectors. It is reflected in the *Table Size* field in the PCIe MSI-X capability structure.

7.3.4.3.1 MSI-X Vectors Used by Physical Function (PF)

PF is responsible for the timer and other interrupt causes that include the VM to PF mailbox cause (explained in the virtualization sections). These events are reflected in EICR[30:16] and MSI-X vectors are the same as the non-IOV mode (illustrated in Figure 7-21). When there are less than the maximum possible active VF's, some of the Tx and Rx queues can be associated with the PF. These queues can be used for the sake of additional VM's serviced by the hypervisor (the same as VMDq mode) or some Kernel applications handled by the hypervisor. Tx and Rx mapping to the IVAR registers is shown in Figure 7-22 and mapping to the EICR, EICR(1),...EICR(2) registers as well as the MSI-X vectors is shown in Figure 7-23. See Section 7.3.4.3.3 for MSI-X vectors mapping of PF and VF's to the EITR registers.

Note: Software should not assign MSI-X vectors in the PF to Tx and Rx queues that are assigned to other VF's. In the case that VF's become active after the PF used the relevant Tx and Rx queues, it is the responsibility of the PF driver to clear all pending interrupts of the associated MSI-X vectors.

7.3.4.3.2 MSI-X Vectors Used by Virtual Functions (VFs)

Each of the VFs in IOV mode is allocated separate IVAR(s) called VFIVAR registers, and a separate IVAR_MISC called VFIVAR_MISC register. The VFIVAR_MISC maps the mailbox interrupt of the VF to its VFEICR and the MSI-X vector. The VFIVAR registers map the Tx and Rx interrupts of the VF to its VFEICR and the MSI-X vector. The mapping is similar to the mapping in the PF as shown in Figure 7-24 with the following comments:

- Each VF cannot have more than three MSI-X vectors. It has only three active bits in the VFEICR register while VFEICR.bit_num is associated with MSI-X vector (bit_num).
- The Tx and Rx interrupt can be mapped only to MSI-X 0 and MSI-X 1 (associated with VFEICR.0 and VFEICR.1).
- The mailbox interrupt can be mapped to any of the three MSI-X vectors. However, when all three of them are allocated by the operating system, software should map the mailbox to MSI-X 2 (associated with VFEICR.2). This rule should be kept since only VFEICR.0 and VFEICR.1 have ITR registers (VFEITR-0 and VFEITR-1).
- Association between the Tx and Rx queues and the VFIVAR registers is shown in the Figure 7-24, Figure 7-25 and Figure 7-26 for IOV-64 (64 VF's), IOV-32 and IOV-16. The colored boxes in the figures show the mapping between VF Rx and Tx queues to VFIVAR registers while the dashed boxes show the physical IVAR registers and the associated physical Rx and Tx queues.

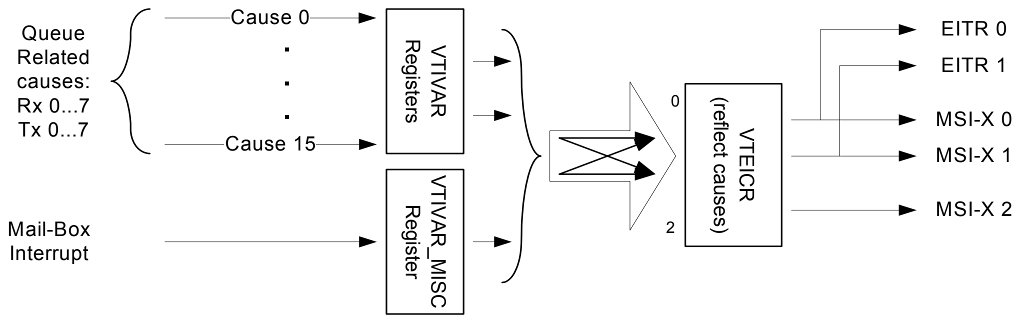

Figure 7-24 VF Interrupt Cause Mapping (MSI-X, IOV)

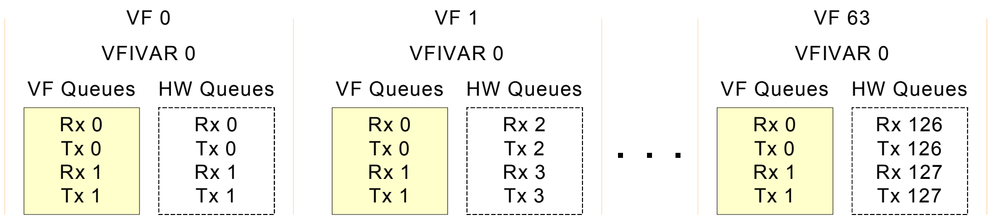

Figure 7-25 VF Mapping of Rx and Tx Queue to VFIVAR in 64 VF's Mode

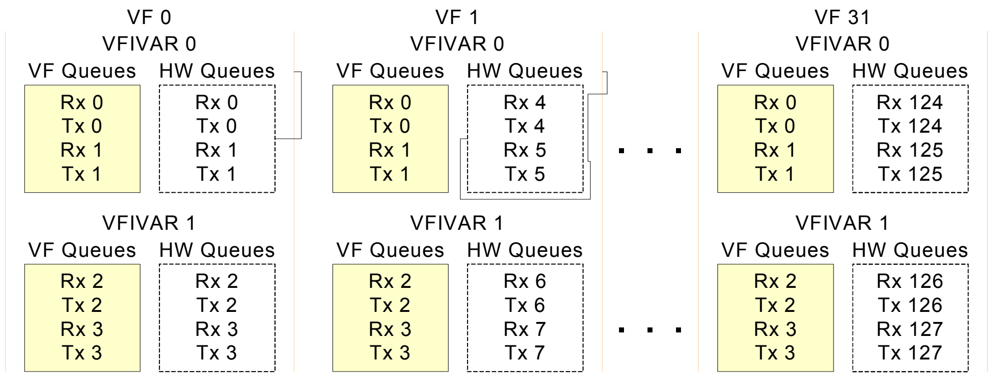

Figure 7-26 VF Mapping of Rx and Tx Queue to VFIVAR in 32 VF's Mode

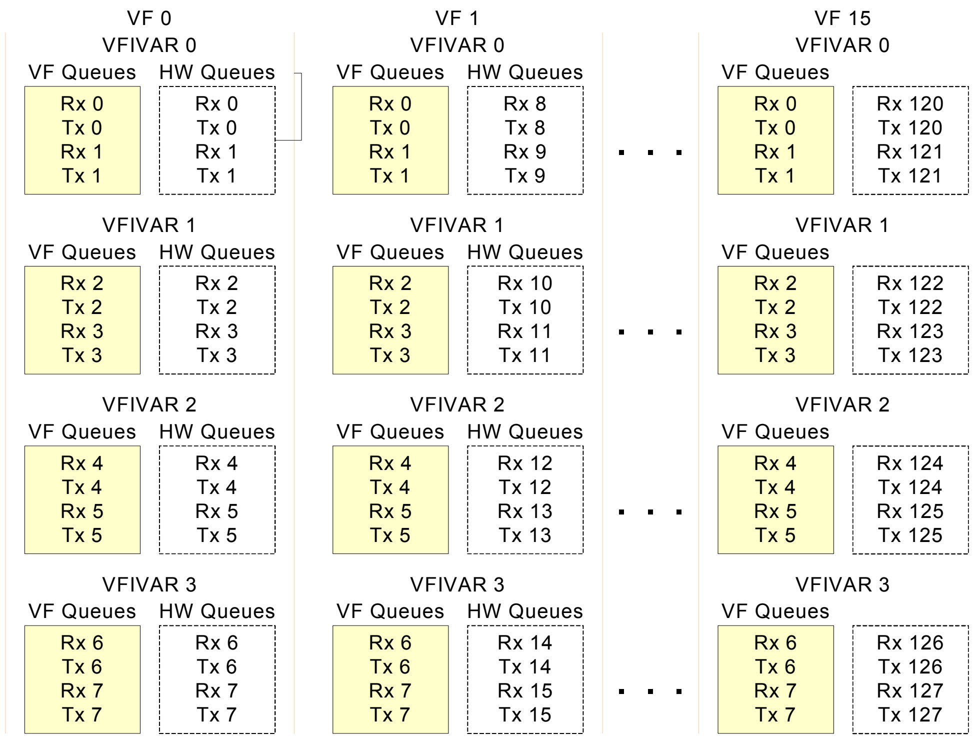

Figure 7-27 VF Mapping of Rx and Tx Queue to VFIVAR in 16 VF's Mode

7.3.4.3.3 MSI-X Vectors Mapping to EITR

EITR registers are aimed for Tx and Rx interrupt throttling. In IOV mode, the Tx and Rx queues might belong to either the PF or to the VF's. EITR(1...63) are multiplexed between the PF and the VF's as configured by the EITRSEL register. Figure 7-28 and Table 7-47 show the multiplexing logic and required software settings. For any active VF (starting from VF32 and above), software should program the matching bit in the EITRSEL to 1b. For any EITR that belongs to a VF, software should not map any interrupt causes in the PF to an MSI-X vector that is associated with the same EITR register.

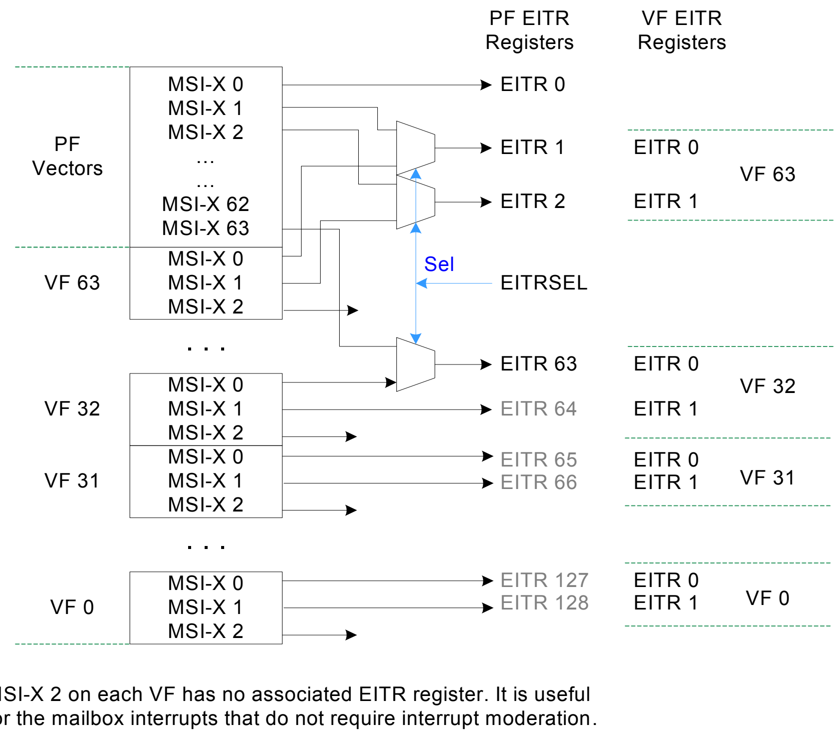

Figure 7-28 PF / VF MSI-X Vectors Mapping to EITR

Table 7-47 PF / VF MSI-X Vectors Mapping Table to EITR Registers

VM Active	EITRSEL.N Setting	MSI-X Routing to EITR
Non-IOV or VF(32...63) inactive	EITRSEL must be set to 0x0000	MSI-X(1...63) -> EITR(1...63)
VF(32) active	EITRSEL[0] must be set to 1b	VF(32) MSI-X(0) -> EITR(63)
VF(33) active	EITRSEL[1] must be set to 1b	VF(33) MSI-X(1) -> EITR(62) VF(33) MSI-X(0) -> EITR(61)
VF(34) active	EITRSEL[2] must be set to 1b	VF(34) MSI-X(1) -> EITR(60) VF(34) MSI-X(0) -> EITR(59)
	. . .	
VF(62) active	EITRSEL[30] must be set to 1b	VF(62) MSI-X(1) -> EITR(4) VF(62) MSI-X(0) -> EITR(3)
VF(63) active	EITRSEL[31] must be set to 1b	VF(63) MSI-X(1) -> EITR(2) VF(63) MSI-X(0) -> EITR(1)

7.4 802.1q VLAN Support

The 82599 provides several specific mechanisms to support 802.1q VLANs:

- Optional adding (for transmits) and stripping (for receives) of IEEE 802.1q VLAN tags.
- Optional ability to filter packets belonging to certain 802.1q VLANs.

7.4.1 802.1q VLAN Packet Format

The following table compares an untagged 802.3 Ethernet packet with an 802.1q VLAN tagged packet:

802.3 Packet	#Octets	802.1q VLAN Packet	#Octets
DA	6	DA	6
SA	6	SA	6
Type/Length	2	802.1q Tag	4
Data	46-1500	Type/Length	2
CRC	4	Data	46-1500
		CRC*	4

Note: The CRC for the 802.1q tagged frame is re-computed, so that it covers the entire tagged frame including the 802.1q tag header. Also, maximum frame size for an 802.1q VLAN packet is 1522 octets as opposed to 1518 octets for a normal 802.3z Ethernet packet.

7.4.2 802.1q Tagged Frames

For 802.1q, the *Tag Header* field consists of four octets comprised of the Tag Protocol Identifier (TPID) and Tag Control Information (TCI); each taking two octets. The first 16 bits of the tag header makes up the TPID. It contains the protocol type that identifies the packet as a valid 802.1q tagged packet.

The two octets making up the TCI contain three fields as follows:

- User Priority (UP)
- Canonical Form Indicator (CFI). Should be set to 0b for transmits. For receives, the device has the capability to filter out packets that have this bit set. See the *CFIEN* and *CFI* bits in the VLNCTRL
- VLAN Identifier (VID)

Octet 1								**Octet 2**							
UP			CFI	VID											

7.4.3 Transmitting and Receiving 802.1q Packets

Since the 802.1q tag is only four bytes, adding and stripping of tags can be done completely in software. (In other words, for transmits, software inserts the tag into packet data before it builds the transmit descriptor list, and for receives, software strips the 4-byte tag from the packet data before delivering the packet to upper layer software). However, because adding and stripping of tags in software adds overhead for the host, the 82599 has additional capabilities to add and strip tags in hardware. See Section 7.4.3.1 and Section 7.4.3.2.

7.4.3.1 Adding 802.1q Tags on Transmits

Software might instruct the 82599 to insert an 802.1q VLAN tag on a per-packet basis. If the *VLE* bit in the transmit descriptor is set to 1b, then the 82599 inserts a VLAN tag into the packet that it transmits over the wire. The Tag Protocol Identifier — TPID (VLAN Ether Type) field of the 802.1q tag comes from the DMATXCTL.VT, and the Tag Control Information (TCI) of the 802.1q tag comes from the VLAN field of the legacy transmit descriptor or the *VLAN Tag* field of the advanced data transmit descriptor.

7.4.3.2 Stripping 802.1q Tags on Receives

Software might instruct the 82599 to strip 802.1q VLAN tags from received packets. The policy whether to strip the VLAN tag is configurable per queue.

If the RXDCTL.VME bit for a given queue is set to 1b, and the incoming packet is an 802.1q VLAN packet (that is, its *Ethernet Type* field matched the VLNCTRL.VET), then the 82599 strips the 4-byte VLAN tag from the packet, and stores the TCI in the *VLAN Tag* field of the receive descriptor.

The 82599 also sets the *VP* bit in the receive descriptor to indicate that the packet had a VLAN tag that was stripped. If the RXDCTL.VME bit is not set, the 802.1q packets can still be received if they pass the receive filter, but the VLAN tag is not stripped and the VP bit is not set.

7.4.4 802.1q VLAN Packet Filtering

VLAN filtering is enabled by setting the VLNCTRL.VFE bit to 1b. If enabled, hardware compares the *Type* field of the incoming packet to a 16-bit field in the VLAN Ether Type (VET) register. If the *VLAN Type* field in the incoming packet matches the VET register, the packet is then compared against the VLAN Filter Table Array for acceptance.

The VLAN filter register VTFA, is a vector array composed of 4096 bits. The VLAN ID (VID) is a 12-bit field in the VLAN tag that is used as an index pointer to this vector. If the VID in a received packet points to an active bit (set to 1b), the packet matches the VLAN filter. The 4096-bit vector is comprised of 128 x 32 bit registers. The upper 7 bits of the VID selects one of the 128 registers while the lower 5 bits map the bit within the selected register.

Two other bits in the VLNCTRL register, *CFIEN* and *CFI*, are also used in conjunction with 802.1q VLAN filtering operations. *CFIEN* enables the comparison of the value of the *CFI* bit in the 802.1q packet to the Receive Control register *CFI* bit as acceptance criteria for the packet.

Note: The *VFE* bit does not effect whether the VLAN tag is stripped. It only effects whether the VLAN packet passes the receive filter.

7.4.5 Double VLAN and Single VLAN Support

The 82599 supports a mode where all received and sent packets have at least one VLAN tag in addition to the regular tagging that might optionally be added. In this document, when a packet carries two VLAN headers, the first header is referred to as an outer VLAN and the second header as an inner VLAN header (as listed in the table that follows). This mode is used for systems where the near end switch adds the outer VLAN header containing switching information. This mode is enabled by the following configuration:

- This mode is activated by setting the DMATXCTL.GDV and the *Extended VLAN* bit in the CTRL_EXT register.
- The Ethertype of the VLAN tag used for the additional VLAN is defined in the *VET EXT* field in the EXVET register.

Cross functionality with Manageability

The 82599 does not provide any stripping or adding VLAN header(s) to manageability packets. Therefore, packets that are directed to/from the manageability controller should include the VLAN headers as part of the Rx/Tx data. The manageability controller should know if the 82599 is set to double VLAN mode as well as the VLAN Ethertype(s). When operating in a double VLAN mode, control packets sent by the manageability controller with no VLAN headers should not activate any hardware offload other than LinkSec encapsulation.

Table 7-48 Transmit Handling of Packets with VLAN Header(s)

MAC Address	Outer VLAN	Inner VLAN	L2 Payload	Ethernet CRC