# Intel® Ethernet Adaptive Virtual Function (AVF) Hardware Architecture Specification (HAS)

Networking Division

336311-002
Revision: 1.0
February 2018

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors which may cause deviations from published specifications.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

# Revisions

| Revision | Date | Notes |
|---|---|---|
| 0.4 | September 2017 | • First Release (Intel Public). |
| 1.0 | February 2018 | • Updated section 2.0 (LAN Queue Interface; descriptor tables). |

# Content

# 1.0 Introduction

This document describes the hardware interface of a Single Root I/O Virtualization (SR-IOV) Virtual Function (VF) that is compatible with the AVF driver. It also describes the interface between the AVF driver and a compliant Physical Function (PF) driver used to negotiate the capabilities of the VF driver/hardware/PF combination.

The Intel® Ethernet 700 Series supports an AVF compliant interface.

## 1.1 Features

The following minimal features are supported by the AVF driver:

1. A mailbox to the PF driver
2. 4 Rx and Tx queue pairs
3. 5 MSI-X interrupts
4. Per vector interrupt moderation
5. An RSS table of 64 entries that can point to up to 4 queues
6. Basic Rx and Tx offloads (checksum and TSO for non-tunneled packets).

The following features are optional and might be exposed by the hardware:

1. More queues or more interrupts.
2. Extension of the RSS table so that it can point to a larger number of queues/use more entries
3. RDMA support
4. VF VLAN trunking
5. VF promiscuous
6. Negotiate header-split
7. Negotiate Tx checksum/TSO for tunneled packets

### 1.1.1 Mailbox

The mailbox described in Section 4.0 is a descriptor ring initiated by the AVF driver and can be used for communication with the PF driver for capabilities query and for configurations.

### 1.1.2 Queue Pairs

The AVF driver exposes at least four Tx and Rx queues. Each queue is initialized by the PF driver upon a request from the AVF driver. The AVF driver then manages the descriptor ring independently. Receive queues are described in Section 2.1 and transmit queues in Section 2.2.

### 1.1.3 Interrupts

The AVF driver exposes at least five MSI-X vectors mapped in BAR3 of the config space. These vectors can be associated with queue write-back events, or with control events using registers accessible by the AVF driver. The interrupts can also be directly moderated using registers accessible by the AVF driver. The interrupt mechanism is described in Section 3.0.

**Note:** While MSI-X vectors are provided by the hardware and the AVF kernel drivers uses these interrupts, a poll mode driver might be implemented on an AVF compliant hardware. Refer to Appendix A for details of a polling mode driver operation.

### 1.1.4 Rx Offloads

The AVF driver exposes a set of receive offloads. Namely, it supports RSS, receive CRC and checksum, and VLAN extraction to the receive descriptor as described in Section 2.1.6.

### 1.1.5 Tx Offloads

The AVF driver exposes a set of stateless transmit offloads. Namely, it supports LSO, transmit checksum, VLAN insertion from the transmit descriptor as described in Section 2.2.5.

## 1.2 AVF Driver Software Flows

The AVF driver software flows (init and run time) are described in Section 6.0.

### 1.2.1 Driver Identification

The AVF driver recognizes devices according to the following identification strings.

**Note:** The following driver identification string device IDs can be used for all Intel devices that support AVF.

**Table 1-1. Driver Identification Strings**

| Vendor ID | Device ID | Sub Vendor ID | Sub Device ID | Code Name | Branding String |
|---|---|---|---|---|---|
| 0x8086 | 0x1889 | * | * | Virtualization AVF device for non-Windows* hypervisors | Intel® Ethernet Adaptive Virtual Function |
| 0x8086 | 0x1889 | 0x8086 | 0x0001 | Virtualization AVF device for Windows HyperV | Intel® Ethernet Adaptive Virtual Function |

### 1.2.2 Feature Query

The AVF driver can discover which features are available in hardware as described in Section 6.1.

### 1.2.3 Device Configuration

The AVF driver can request from the PF configurations required for its operation like queue init or new filters as described in Section 6.2.

## 1.3 Supported Hardware

The following devices are compatible with AVF drivers:

- Intel® Ethernet Controller X710, XL710, and XXV710
- Intel® Ethernet Connection X722

Other products will be added in the future.

**NOTE: This page has been intentionally left blank.**

# 2.0 LAN Queue Interface

## 2.1 Receive Queues

This section includes the following topics:

- Storage of receive packets in system memory.
- Receive descriptor queues description (such as descriptor rings).
- Ring management — Indication of free descriptors to the hardware and indication of completed descriptors back to the software.

### 2.1.1 Receive Queue Ring

Received packets are posted to host memory through a set of queues. Each queue is a cyclic ring made of a sequence of receive descriptors in contiguous memory. These queues are also called descriptor rings. The AVF driver supports up to four receive queues allocated to the VF. Receive queues are defined by a set of parameters called the queue context. The queue context is managed by the PF driver and it is out of the scope of this document. Only the tail registers (needed at run time) are accessible to the AVF driver. Part of these context parameters are kept in hardware (like the tail register, queue enable / disable flags and interrupt related context).

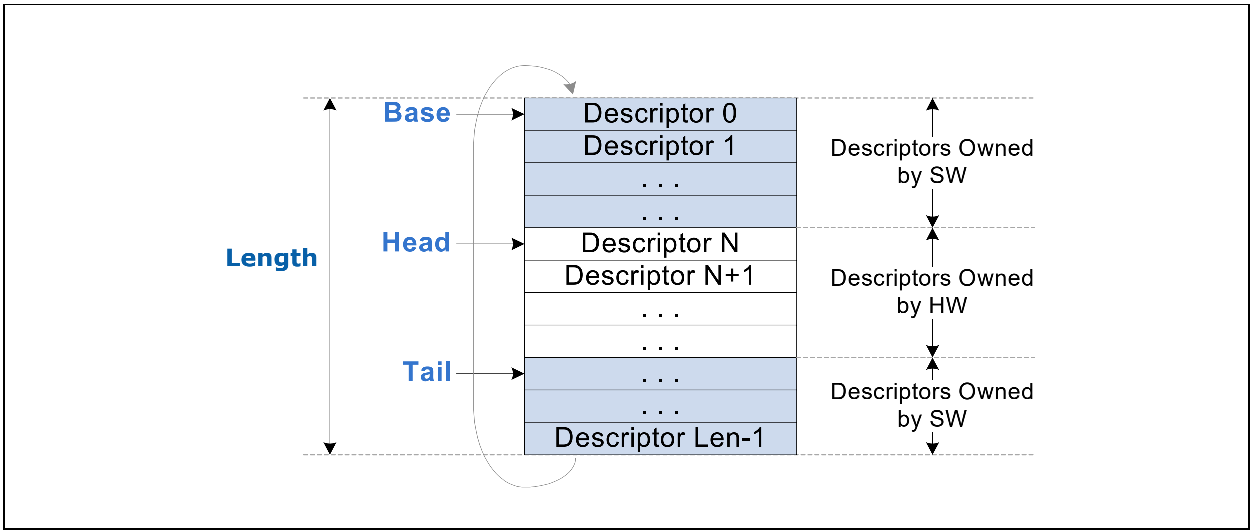


**Figure 2-1. Receive Descriptor Ring Structure**

Receive buffers prepared in the system (host) memory buffers are indicated to hardware by descriptors in the ring. There are several types of descriptors detailed in Section 2.1.2. These include pointers to the data buffers and status indications of the received packets. Figure 2-2 shows two examples of receive packets in host memory composed of two buffers (indicated by two matched descriptors).

A few rules related to receive packet posting to host memory are:

- Receive packets might span one to five buffers (descriptors).
- Receive packets shorter than 64 bytes are never posted to host memory.

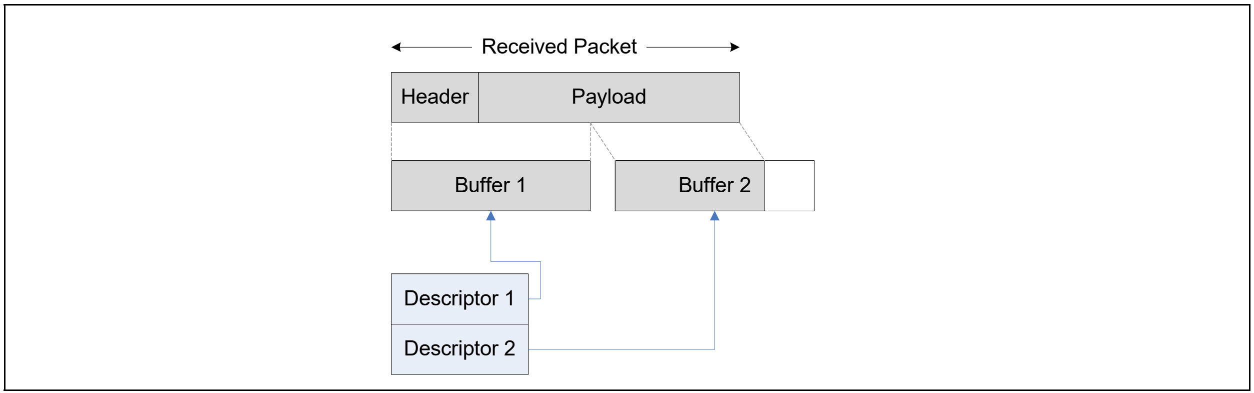


**Figure 2-2. Receive Packet in System Memory**

# 2.1.2 Receive Queue Descriptor

## 2.1.2.1 Receive Descriptor - Read Format

### 2.1.2.1.1 16-byte Receive Descriptors Read Format

Following is the 16-byte receive descriptor read format prepared by software.

**Table 2-1. 16-byte Receive Descriptors Read Format**

| Quad Word | 63 | 0 |
|---|---|---|
| 0 | Packet Buffer Address | |
| 1 | Reserved (0x0) | |
| | 63 | 0 |

**Packet Buffer Address (64)**

The physical address of the packet buffer defined in byte units. The packet buffer size is defined by the DBUFF parameter in the receive queue context.

### 2.1.2.1.2 32-byte Receive Descriptors Read Format

Following is the 32-byte receive descriptor read format prepared by software.

### Table 2-2. 32-byte Receive Descriptors Read Format

| Quad Word | 63 | 0 |
|---|---|---|
| 0 | Packet Buffer Address | |
| 1 | Reserved (0x0) | |
| 2 | Reserved (0x0) | |
| 3 | Reserved (0x0) | |
| | 63 | 0 |

The fields in first 16 bytes are identical to the 16-byte descriptors described in Section 2.1.2.1.1.

## 2.1.2.2 Receive Descriptor (Write Back Format)

The following subsections describe the fields of the Receive Descriptor as written back by hardware when using 16-byte and 32-byte descriptors. In both cases, a single packet might span on a single buffer or multiple buffers as reported by their matched descriptors. If a packet is described by a single descriptor then all the fields are valid. Following are some rules that apply for a packet that is described by multiple descriptors:

- The following fields are valid in all descriptors of a packet: DD flag (Done); EOP flag (End of Packet) and PKTL field (Packet content length).
- All other fields are valid only in the last descriptor of a packet.

### 2.1.2.2.1 16-byte Legacy Receive Descriptors WB Format

Following is the 16-byte receive descriptor Write Back (WB) format.

<table>
<tr><td>Quad Word</td><td colspan="2">63 ... 32</td><td colspan="4">31 ... 16</td><td>15 14</td><td>13 ... 0</td></tr>
<tr><td>0</td><td colspan="2">Filter Status</td><td colspan="4">L2TAG1</td><td>rsv</td><td>MIRR</td></tr>
<tr><td>1</td><td>Length</td><td colspan="2">PTYPE</td><td>rsv</td><td>Error</td><td colspan="3">Status</td></tr>
<tr><td></td><td>63 ... 38</td><td colspan="2">37 ... 30</td><td>29 ... 27</td><td>26 ... 19</td><td colspan="3">18 ... 16 15 ... 0</td></tr>
</table>

### Reserved/RSV (Quad Word 1, 3 bits, Quad Word 0, 2 bits)

Reserved.

### Status Field (Quad Word 1, 19 bits)

| Bits | Name | Functionality |
|---|---|---|
| 0 | DD | Descriptor done indication flag. |
| 1 | EOP | End of packet flag is set to 1b indicating that this descriptor is the last one of a packet. |
| 2 | L2TAG1P | VLAN presence indication while the VLAN tag is stripped from the packet and reported in the L2TAG1 field in the descriptor. |
| 3 | L3L4P | For IP packets, this flag indicates that detectable L3 and L4 integrity check is processed by hardware.<br>See Section 2.1.6.3 for details on which headers are processed and how. |
| 4 | CRCP | CRCP indicates that the Ethernet CRC is posted with data to the host buffer. Note that strip CRC is enabled by the VIRTCHNL_VF_OFFLOAD_L2 capability.<br>If the RXE error flag is set, the CRC bytes are not stripped regardless of the CRCStrip flag in the queue context. Loop back packets originated by another local VSI for which the hardware computes the CRC are never posted with the CRC bytes regardless of the CRCStrip setting in the queue context. |
| 8:5 | Reserved | Reserved. Set to 0x0 by hardware. |
| 9:10 | UMBCAST | Destination address can be one of the following:<br>• 00b = Unicast<br>• 01b =- Multicast<br>• 10b = Broadcast<br>• 11b = Reserved<br>Non-parsed packets are indicated by PTYPE equals to PAYLOAD (non identified MAC header). |
| 11 | Reserved | Reserved |
| 12:13 | FLTSTAT | The FLTSTAT indicates the reported content in the *Filter Status* field.<br>00b = Reserved.<br>10b = Reserved.<br>11b = Hash filter signature (RSS). |
| 14 | LPBK | Loop back indication meaning that the packet is originated from this system rather than the network. |
| 15 | IPV6EXADD | Set when an IPv6 packet contains a Destination Options Header or a Routing Header. If the packet contains two IPv6 headers (tunneling), the IPV6EXADD is a logic 'OR' function of the two IP headers. |
| 16:17 | Reserved | Reserved |
| 18 | INT_UDP_0 | This flag is set for received UDP packets on which the UDP checksum word equals to zero.<br>Note that UDP checksum zero is an indication that there is no checksum. This option is valid only for IPv4 packets and considered an exception error for IPv6 packets (reported to the stack by the driver). |

### Error Field (Quad Word 1, 8 bits)

### Table 2-3. Error Bits

| Bits | Name | Functionality |
|---|---|---|
| 2:0 | RSV | Reserved |
| 3:5 | L3L4E | For IP packets processed by hardware, the L3L4E flag has the following encoding:<br>Bit 3 = IPE: IP checksum error indication.<br>Bit 4 = L4E: L4 integrity error indication.<br>Bit 5 = Reserved for the purpose of these bits, a tunneled packet is a packet with an inner IP header. For example, a VXLAN packet without an inner IP is not considered as a tunnel packet. |
| 6 | OVERSIZE | Oversize packet error indicates that the packet is larger than five descriptors. In this case the portions of the packet that exceeds the permitted number of descriptor(s) is not posted to host memory. |
| 7 | RSV | Reserved |

### L2TAG1 (Quad Word 0, 16 bits)

Stripped L2 VLAN Tag from the receive packet. This field is valid if the L2TAG1P flag in this descriptor is set (see additional description of the L2TAG1P flag). The L2TAG1 includes the VLAN tag if enabled via VIRTCHNL_VF_OFFLOAD_VLAN capability.

### Filter Status (Quad Word 0, 32 bits)

- If the packet matches the Hash filter, then FLTSTAT equals 11b and this field contains the hash signature (RSS).
- Else, FLTSTAT equals 00b and this field is set to zero.

### Length (Quad Word 1, 26 bits)

| Bits | Name | Functionality |
|---|---|---|
| 0:13 | PKTL | Packet content length in the packet buffer defined in byte units. |
| 14:25 | Reserved | Reserved |

### PTYPE (Quad Word 1, 8 bits)

Packet Type field encodes supported packet types as listed in Table 2-4.

### Table 2-4. Packet Types

PTYPES that are preserved for base-mode support.

| PTYPE | Description | PTYPE | Description |
|---|---|---|---|
| **L2 Packet Types** | | | |
| 0 | Reserved | 11 | MAC, ARP |
| 1 | MAC, PAY2 | 12 | MAC, PAY3[1] |

[1] PTYPE 12 should be ignored.

| PTYPE | Description | PTYPE | Description |
|---|---|---|---|
| **Non-tunneled IPv4** | | **Non-tunneled IPv6** | |
| 22 | MAC, IPvV4FRAG, PAY3 | 88 | MAC, IPv6+IPv6FRAG, PAY3 |
| 23 | MAC, IPv4, PAY3 | 89 | MAC, IPv6, PAY3 |
| 24 | MAC, IPv4, UDP, PAY4 | 90 | MAC, IPv6, UDP, PAY4 |
| 25 | Reserved | 91 | Reserved |
| 26 | MAC, IPv4, TCP, PAY4 | 92 | MAC, IPv6, TCP, PAY4 |
| 27 | MAC, IPv4, SCTP, PAY4 | 93 | MAC, IPv6, SCTP, PAY4 |
| 28 | MAC, IPv4, ICMP, PAY4 | 94 | MAC, IPv6, ICMP, PAY4 |

#### 2.1.2.2.2 32-byte Receive Descriptors Write Back Format

The 32-byte descriptor is composed of four quad words. The first two Qwords are identical to the 16-byte descriptor write back.

<table>
<tr><td>Quad Word</td><td>6</td><td colspan="14"></td><td>4</td><td>4</td><td colspan="8"></td><td>3</td><td>3</td><td colspan="4"></td><td>3</td><td>3</td><td>3</td><td>2</td><td colspan="1"></td><td>2</td><td>2</td><td colspan="9"></td><td>1</td><td>1</td><td colspan="1"></td><td>1</td><td>1</td><td colspan="12"></td></tr>
<tr><td></td><td>3</td><td colspan="14"></td><td>8</td><td>7</td><td colspan="8"></td><td>8</td><td>7</td><td colspan="4"></td><td>2</td><td>1</td><td>0</td><td>9</td><td colspan="1"></td><td>7</td><td>6</td><td colspan="9"></td><td>6</td><td>5</td><td colspan="1"></td><td>4</td><td>3</td><td colspan="11"></td><td>0</td></tr>
<tr><td>0</td><td colspan="32">Filter Status</td><td colspan="16">L2TAG1</td><td colspan="3">rsv</td><td colspan="13">MIRR</td></tr>
<tr><td>1</td><td colspan="26">Length</td><td colspan="8">PTYPE</td><td colspan="3">rsv</td><td colspan="8">Error</td><td colspan="19">Status</td></tr>
<tr><td>2</td><td colspan="16">L2TAG2 (2nd)</td><td colspan="16">L2TAG2 (1st)</td><td colspan="15">rsv</td><td colspan="6">Cryptorsv<br>rsv</td><td colspan="11">Ext_Status</td></tr>
<tr><td>3</td><td colspan="32">FD Filter ID / Flexible Bytes High</td><td colspan="32">Flexible Bytes Low</td></tr>
<tr><td rowspan="2"></td><td>6</td><td colspan="14"></td><td>4</td><td>4</td><td colspan="14"></td><td>3</td><td>3</td><td colspan="13"></td><td>1</td><td>1</td><td colspan="4"></td><td>1</td><td>1</td><td colspan="9"></td><td>0</td></tr>
<tr><td>3</td><td colspan="14"></td><td>8</td><td>7</td><td colspan="14"></td><td>2</td><td>1</td><td colspan="13"></td><td>8</td><td>7</td><td colspan="4"></td><td>2</td><td>1</td><td colspan="10"></td></tr>
</table>

## 2.1.3 Dummy Receive Descriptors

In some cases, hardware might reserve one more descriptor than needed for a given packet. In this case, this dummy descriptor is written back as the last descriptor of the packet with a data length field == 0. For this packet, the dummy descriptor carries all the flags that are normally written at the last descriptor of a multi-descriptors packets. A dummy descriptor might occur in both 16- and 32-byte formats.

## 2.1.4 Receive Init Flows

The receive init flow is described in Section 6.1.2.

## 2.1.5 Packet Receive Flow

### 2.1.5.1 Receive Descriptors Preparation and Tail Bump

During normal operation, the AVF driver accesses hardware directly using the following flow:

- Prepare receive descriptors by clearing the *DD* bit and setting the buffer pointer(s). Start at the descriptor indicated by the TAIL pointer in the relevant QRX_TAIL register (Section 7.2.4.2).
- Software should never set the TAIL to a value above the descriptors owned by hardware minus 1. The descriptors considered as owned by hardware are those ones already indicated to hardware but not yet reported as completed.
- Bump the TAIL to the last prepared descriptor plus one.

### 2.1.5.2 Descriptor Write Back

Hardware reports a completion of receive packet in host memory by status indication in the receive descriptor(s) of the packet (descriptor write back). Hardware writes back completed descriptor status in one of the following cases:

- Entire lines of descriptors (4 x 32-byte descriptors or 8 x 16-byte descriptors) are completed.
- In systems that supports a cache based queue context, all completed descriptors of a queue evicted from the internal cache
- Upon assertion of the interrupt associated with the queue.
- If the queue is configured for No Expire then every completed descriptor is written back immediately.

**Note:** If the number of free descriptors available to hardware is lower than a threshold set by the PF driver, then an immediate interrupt is triggered.

In some systems, while in polling mode, the write backs might not fit into any of the previously mentioned categories and the AVF driver might detect that there are a few descriptor write backs (less than a cache line) pending for longer than expected. As a result, the AVF driver triggers a software interrupt to force a write back.

Most applications use interrupts to invoke the AVF driver. Before initiating the interrupt, hardware posts all completed descriptors of the queue that might be kept in the device caches. Following an interrupt assertion, the device masks any further interrupts preventing interrupt nesting. When the interrupts are re-enabled (by software), additional interrupts that trigger additional write-back of completed descriptors can be initiated.

In some applications, the AVF driver works in polling mode (with no interrupts). Configuring this option is done by asking for the VIRTCHNL_VF_OFFLOAD_RX_POLLING AVF capability.

## 2.1.6 Basic Receive Offloads

### 2.1.6.1 Strip Ethernet CRC Bytes

The AVF driver checks the integrity of the Ethernet CRC and possibly strips it from packets that are posted to LAN queues. This capability is exposed in the capability exchange as VIRTCHNL_VF_OFFLOAD_L2.

### 2.1.6.2 VLAN Extraction

Before a packet is stored in host memory, the VLAN tag might be stripped and optionally stored in the receive descriptor. The action done is defined by the PF. This capability is exposed in the capability exchange as VIRTCHNL_VF_OFFLOAD_VLAN.

### 2.1.6.3 Receive L3 and L4 Integrity Check Offload

The AVF driver compliant hardware offloads the following L3 and L4 integrity checks: IPv4 header(s) checksum, UDP checksum, and SCTP CRC integrity. Hardware identifies the packet type and then checks the matched integrity scheme. The identified packet type is reported in the *PTYPE* field in the receive descriptor. Processing indication of the L3 and L4 headers is reported on the L3L4P flag in the receive descriptor. Potential IPv4 checksum error, L4 integrity error and outer IPv4/UDP checksum error are reported by the IPE, L4E and the EIPE error flags in the legacy receive descriptor respectively. In the advanced receive descriptor it is reported in the flexible error flags.

Some rules for integrity check offload are listed in the text that follows. If the following rules are not met, integrity offload is not provided and the L3L4P is not set.

- IPv4 header is assumed to be at least 20 bytes long (the length of the basic header).
- IPv4 headers might have any IP option headers that fit within the maximum header size (60 bytes).
- IPv6 support: The pseudo header for the L4 checksum takes into account the addresses in the IPv6 header ignoring the optional extension headers. Packets with Routing Header type 2 and Destination Options Header with Home Address option contain an alternative IP address in the extension header. Therefore, checksum calculation for such packets most probably results in erroneous value. The AVF driver indicates the existence of a Destination Options Header or a Routing Header in the IPV6EXADD bit of the RX descriptor. Software can then do one of the following:
  - Ignore the checksum done by the device.
  - Parse the extension header and identifying if it contains an IP address. Then ignore the checksum done by the device only in this case.
- Fragmented packets — the AVF driver parses fragmented receive packets up to including the IP header (for IPv4) or up to including the fragmentation extension header (for IPv6):
  - L4 checksum offload is not supported for IPv6 fragmented packets and the L3L4P flag in the receive descriptor is not set.
  - Fragmented IPv4 packet is offloaded up to including the IP header.
- TCP header is assumed to be at least 20 bytes long (the length of the basic header).
- The TCP header might have any option headers that fit within the maximum header size (60 bytes).

- VM-to-VM loopback traffic is processed by the hardware for L3/L4 integrity check as any other packet received from the network.

Table 2-5 lists all supported packet formats and the processed integrity. The table uses the following notations:

- IP is a generic term for IPv4 header or IPv6 header. The IPv4 header can have IP option headers and the IPv6 header can have IPv6 extension headers.
- L4 is a generic term for UDP, TCP or SCTP headers.
- IP checksum is meaningful only for IPv4.
- Checksum is a generic term for UDP and TCP checksum as well as SCTP CRC integrity.
- Zero UDP checksum: Zero UDP checksum for IPv4 packet is treated as no checksum and is reported by the hardware as no error and done. Zero UDP checksum for IPv6 packet is illegal and is reported by the hardware as L4 checksum error.

**Table 2-5. Integrity Offload Check for Receive Packet Types**

| Packet Type | Supported Integrity Offload | Reported L3L4P |
|---|---|---|
| IP -> [data / Unknown / fragmented] | IP checksum offload | 1 (for IPv4/0 (for IPv6) |
| IP -> L4 | IP and L4 checksum offload | 1 |
| IP -> IP -> [data / Unknown / fragmented] | 2 x IP checksum offload | 1 if at least one of the IP headers is IPv4 |
| IP -> IP -> L4 | IP and L4 checksum offload | 1 |
| IP -> [tunnel header] -> IP -> data / Unknown / fragmented | Only IP checksum offload | 1 if at least one of the IP headers is IPv4 |
| IP -> [tunnel header] -> IP -> L4 | 1. IP and L4 checksum offload[1] | 1 |
| IP -> [tunnel header] -> data and<br>IP -> [tunnel header] -> MAC -> data | IP checksum offload | 1 (for IPv4/0 (for IPv6) |
| IP -> [tunnel header] -> MAC -> IP -> data | IP checksum (relevant only for IPv4) | 1 if at least one of the IP headers is IPv4 |
| IP -> [tunnel header] -> MAC -> IP -> L4 | IP and L4 checksum offload[1] | 1 |

[1] The L4 checksum offload relates to the inner header:
- For UDP or TCP protocols, the hardware calculates the expected checksum including the pseudo IP header.
- For SCTP protocol, the hardware calculates the expected SCTP CRC.

[2] Tunneling headers can be one of the following: GRE, Teredo, VXLAN UDP header.

## 2.1.6.4 RSS Support

The AVF driver supports by default an RSS table of 64 entries pointing to 4 queues. This capability is exposed in the capability exchange as VIRTCHNL_VF_OFFLOAD_RSS_PF.

The configurations of the table and the key are done through the PF mailbox using the VIRTCHNL_OP_CONFIG_RSS_KEY and VIRTCHNL_OP_CONFIG_RSS_LUT respectively.

The RSS hash value of the receive packet is exposed in the receive descriptor write back in the *Filter Status* field as described in Section 2.1.2.2.

# 2.2 Transmit Queues

## 2.2.1 Transmit Queue Ring

Software prepares structures for transmission in system memory indicated to hardware by a list of consecutive descriptors. These descriptors are organized in a contiguous memory handled as a cyclic queue which is also called a Transmit Descriptor Ring (TDR). Descriptors are initialized by software, and posted to hardware for processing via a write to a doorbell register. The AVF driver supports up to four transmit queues allocated to the VFs.

Transmit queue state and behavior is initialized by PF software, which programs a set of parameters collectively called the queue context. The main parameters are the queue pointers shown in Figure 2-3. The software interface to the queue for initialization, normal operation, and queue disable is described in Section 2.2.3.

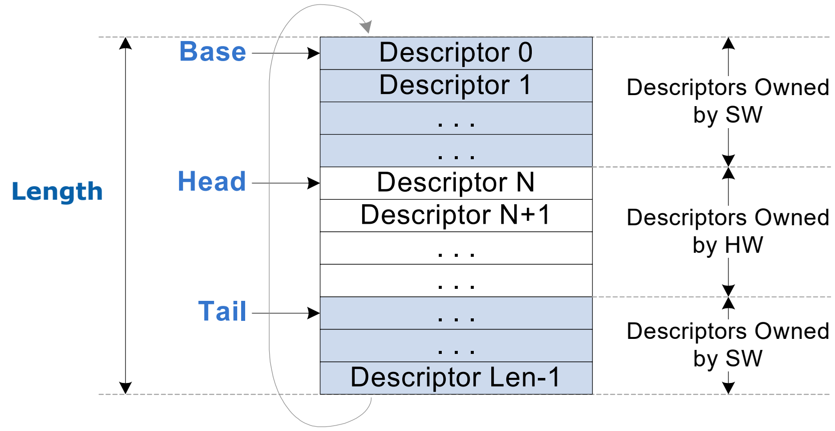


**Figure 2-3. Transmit Descriptor Ring Structure**

When software posts a packet for transmission it can add some specific rules and commands per the transmitted packet. This is done by adding extra descriptors on top of the Data descriptors which point to the transmitted packet. The following sections describes the various descriptor types.

Transmit packets are stored in single or multiple non-contiguous buffers in host memory. The location of these buffers is provided by software to hardware for transmission using descriptors (16-byte structures described in Section 2.1.2.2). These descriptors include pointer and length pairs to the data buffers as well as control fields for the transmit data processing. In some cases additional control parameters that cannot fit within the data descriptors are needed to process the packet(s). In this case, an additional context descriptor (16-byte structures described in Section 2.2.2.3) is posted by software to hardware prior to posting the data descriptors. Figure 2-4 shows an example of a transmit packet in host memory composed of 2 buffers (header buffer and payload buffer), indicated by 2 matched data descriptors and an optional context descriptor.

A few rules related to the transmit packet in host memory are:

- The total size of a single packet in host memory must be at least 17 bytes and up to the Max Frame Size of the port as configured by the PF.

  - — Packets outside this range are considered malicious. The respective queue is stopped and an interrupt is issued to the PF.
  - — This rule applies for single packet send as well as any packet within a transmit segmentation (TSO).
- A single transmit packet may span up to 8 buffers (up to 8 data descriptors per packet including both the header and payload buffers).
  - — When a packet span on multiple buffers, all the descriptor of that packet must be filled similarly.
- The total number of data descriptors for the whole TSO (explained later on in this chapter) is considered unlimited (Limited only by TX queue length) as long as each segment within the TSO obeys the previous rule (up to 8 data descriptors per segment for both the TSO header and the segment payload buffers).
- If a packet or TSO spans on multiple transmit data descriptors, the fields in all the data descriptors must be valid.
- The TSO message header should not span on more than three buffers (Max 3 Descriptors).

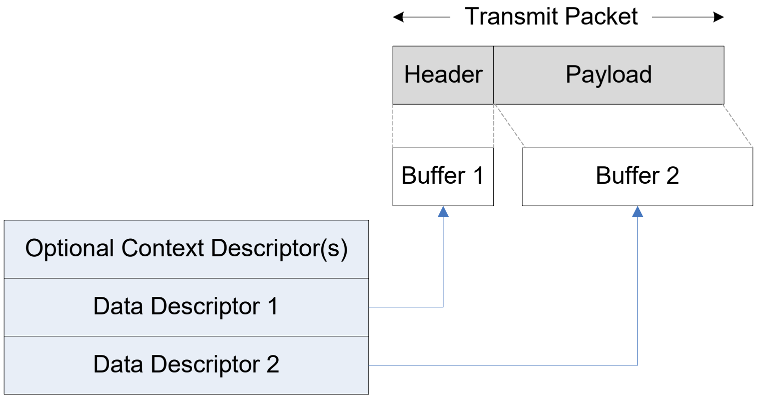


**Figure 2-4. Transmit Packet in System Memory (Example Using 2 Buffers)**

## 2.2.2 Transmit Queue Descriptor

### 2.2.2.1 General Descriptors

**Table 2-6. LAN Descriptor Types**

| Type | DTYP value | Description | Reference |
|---|---|---|---|
| Transmit Data Descriptor | 0x0 | Regular data descriptor used to send LAN packets. | Section |

**Note:** For all descriptors: fields indicated as RSV (reserved) should be set to zero by software at programming time.

## 2.2.2.2 Transmit Data Descriptor

| Quad Word | 6 | | | | |
|---|---|---|---|---|---|
| | 3 | | | | 0 |
| 0 | Tx Packet Buffer Address | | | | |
| 1 | L2 Tag 1 | Tx Buffer Size | Offset | CMD | DTYP |
| | 6 4 | 4 3 | 3 1 | 1 | |
| | 3 8 | 7 4 | 3 6 | 5 4 | 3 0 |

### Descriptor Type - DTYP (Quad Word 1, bits 0:3)

0x0 stands for a Transmit Data Descriptor

### Command Field - CMD (Quad Word 1, bits 4:15) as detailed in the following table.

| Bits | Name | Functionality |
|---|---|---|
| 0 | EOP | End Of Packet. The EOP flag is set in the last descriptor of a packet or TSO. |
| 1 | RS | Report Status. When set, the hardware reports the DMA completion of the transmit descriptor and its data buffer. When it is reported by descriptor Write Back, the DTYP field is set to 0xF and the RS flag is set.<br>The RS flag can be set only on a last Transmit Data Descriptor of a packet or last Transmit Data Descriptor of a TSO. |
| 2 | RSV | Reserved, must be set to 1b. |
| 3 | IL2TAG1 | Insert a VLAN tag from the L2TAG1 field in this descriptor. The capability to add a tag is exposed through the VIRTCHNL_VF_OFFLOAD_VLAN capability. |
| 4 | DUMMY | When the Dummy flag is set, the packet is not transmitted (internal and external). Note that when using the Dummy option, the packet does not have to have a correct checksum. Software should set all the fields in the data descriptor and context descriptors describing the packet structure as it does for nominal packets. |
| 5:6 | IIPT | The IP header type and its offload.<br>00b = Non-IP packet or packet type is not defined by software.<br>01b = IPv6 packet.<br>10b = IPv4 packet with no IP checksum offload.<br>11b = IPv4 packet with IP checksum offload.<br>For an IPv4 TSO message, this field must be set to 11b. |

| Bits | Name | Functionality |
|---|---|---|
| 7 | Reserved | Reserved |
| 8:9 | L4T | L4T is the L4 packet type:<br>00b = Unknown / fragmented packet.<br>01b = TCP.<br>10b = SCTP.<br>11b = UDP.<br>When the L4T is set to other values than 00b, the L4LEN must be defined as well. When set to UDP or TCP, the hardware inserts the L4 checksum and when set to SCTP the hardware inserts the L4 CRC<br>Requesting SCTP CRC / TCP or UDP offload for a packet which was padded by software results in wrong SCTP CRC.<br>EOFT is the EOF Tag to be used:<br>00b = EOFn.<br>01b - EOFt.<br>10b = EOFni.<br>11b = EOFa. |
| 10:11 | RSV | Reserved |

### Header Offset Parameters - OFFSET (Quad Word 1, bits 16:33)

| Bits | Name | Functionality |
|---|---|---|
| 0:6 | MACLEN | MAC Header Length defined in Words.<br>MACLEN defines the L2 header length up to including the Ethertype. If L2 tag(s) are provided in the data buffers, then they are included in MACLEN. |
| 7:13 | IPLEN / FCoELEN | IP header length (including IP optional/extended headers) in the Tx buffer defined in Dwords |
| 14:17 | L4LEN / FCLEN | L4LEN is the L4 header length in the Tx buffer defined in Dwords. L4LEN should obey the following rules:<br>When the L4T field is set to 00b, L4LEN must be set to zero. Otherwise, it should be set to 8 / 12 for UDP / SCTP respectively and should be equal or larger than 20 for TCP. |

### L2 Tag 1 - L2TAG1 (Quad Word 1, 48:63)

A 16-bit VLAN Tag to be inserted into the packet if the IL2TAG1 flag is set. If IL2TAG1 is cleared, L2TAG1 should be set by software to zero. Valid only if VIRTCHNL_VF_OFFLOAD_VLAN capability is exposed.

### Transmit Buffer Description - BUFF (Quad Word 0, bits 0:63; Quad Word 1, bits 34:47)

**Table 2-7. Transmit Buffer**

| Bits | Name | Functionality |
|---|---|---|
| 34:47 | BSIZE | Buffer size in byte units from 1 byte up to 16KB minus 1 |
| 0:63 | BADDR | Buffer address in byte granularity |

## 2.2.2.2.1 Transmit Descriptors Write Back Format

| | | | | |
|---|---|---|---|---|
| Quad Word | 6 | | | |
| | 3 | | | 0 |
| 0 | Reserved | | | |
| 1 | Reserved | | DTYP | |
| | 6 | | | |
| | 3 | 4 | 3 | 0 |

### Descriptor Type - DTYP (Quad Word 1, bits 0:3)

Hardware indicates a completed descriptor by setting the DTYP field to a value of 0xF.

Hardware reports this status in the following two cases:

- For descriptors with the *RS* bit set (Report Status).
- For the last descriptor of a command that was executed before an interrupt of the queue is initiated.

### Completion Flags RS (Quad Word 1, bit 5)

- *RS* bit is set if it was active by software.

### 2.2.2.3 LAN Transmit Context Descriptors

A context descriptor might contain some additional setting options for a single packet or TSO defined by the Data descriptor(s) that follows. Following the transmission of the packet or TSO, the context provided by this descriptor is expired.

<table>
<tr><td>Quad Word</td><td colspan="2">63 … 48</td><td>47 … 32</td><td colspan="2">31 … 24</td><td colspan="4">23 … 0</td></tr>
<tr><td>0</td><td colspan="2">rsv</td><td>L2TAG2 (STag / VEXT)</td><td colspan="2">rsv</td><td colspan="4">Tunneling Parameters</td></tr>
<tr><td>1</td><td>MSS / TARGET_VSI</td><td>rsv</td><td colspan="2">TSO Total Length</td><td colspan="2">rsv</td><td>rsv</td><td>CMD</td><td>DTYP</td></tr>
<tr><td></td><td>63 … 50</td><td>49 48</td><td colspan="2">47 … 30</td><td colspan="2">29 … 18</td><td>17 … 11</td><td>10 … 4</td><td>3 … 0</td></tr>
</table>

#### Descriptor Type - DTYP (Quad Word 1, bits 0:3)

0x1 stands for a LAN Context Descriptor.

#### Command Field - CMD (Quad Word 1, bits 4:10)

| Bits | Name | Functionality |
|---|---|---|
| 0 | TSO | TSO is activated when the TSO flag is set. |
| 6:1 | RSV | Reserved |

#### Segmentation Parameters / Switching Parameter (Quad Word 1, bits 30:47; bits 50:63)

| Bits | Name | Functionality |
|---|---|---|
| 30:47 | TLEN | For a TSO message:<br>TSO Total Length. This field defines the L4 payload bytes that should be segmented. Note that the sum of all transmit buffer sizes of the TSO should match exactly the TLEN plus the TSO header size in host memory.<br>If the TSO flag is cleared, the TLEN should be set by software to zero.<br>If the TSO flag is set, the TLEN must be set by software to a larger value than zero. |
| 50:63 | MSS | When the TSO flag is set, the MSS field defines the Maximum Segment Size of the packet's payload in the TSO (excluding the L2, L3 and L4 headers). The MSS should not be set to a lower value than 88. Should be set to zero, otherwise. |

**RSV** Reserved bits that must be set to zero.

## 2.2.3 Transmit init Flows

The transmit init flow is described in Section 6.1.2.

## 2.2.4 Packet Transmission Flow

During normal operation, the AVF driver accesses the hardware directly. To transmit a packet, software prepares transmit descriptors starting at the descriptor indicated by the TQTAIL pointer in the relevant QTX_TAIL register. The Descriptors point to the Transmitted packet. After the Descriptors are ready in the TX queue, software notifies the hardware. This notification is called a Doorbell.

The following flow is used to send packets and to report back completion of the transmission:

- The AVF driver updates queue tail (TQTAIL) using the QTX_TAIL registers (Section 7.2.4.1).
- The AVF driver should update TQTAIL at packets boundaries. For TSO, it is the whole TSO; for a single packet, it is the whole packet. Updating TQTAIL to point into the middle of a multi-descriptor operation may trigger a malicious driver detection event and halt the queue.
- Software should never set the TQTAIL to a value above the descriptors owned by the hardware minus 1. Descriptors considered as owned by hardware are those already indicated to the hardware but are not yet reported as completed. Overrunning the queue triggers a Malicious Driver Detection (MDD) event.
- Hardware reports completed descriptors only for descriptors with the *RS* or *RE* (or both) bit set in the CMD field in the descriptor. As part of the ITR expire flow (described in Section 3.3.1), hardware writes back the latest processed descriptor, even if it is not marked with *RS* or *RE* bits.

## 2.2.5 Stateless Transmit Offloads

### 2.2.5.1 L2 Offloads

Hardware calculates and inserts the Ethernet CRC for all packets transmitted to the network. For small packets, padding is added before the CRC up to 60 bytes. This capability is exposed in the capability exchange as VIRTCHNL_VF_OFFLOAD_L2.

### 2.2.5.2 VLAN Insertion

A VLAN tag can be inserted to the packets in two ways:

1. As part of the packet buffer.
2. As part of the transmit descriptor in the *L2TAG1* field if the *IL2TAG1* field is set.

This capability is exposed in the capability exchange as VIRTCHNL_VF_OFFLOAD_VLAN.

If a packet is sent with a VLAN tag, either embedded in packet or via the descriptor, with a VLAN ID not equal zero, from a VF not allowed to add a tag, it will be dropped.

### 2.2.5.3 Transmit L3 and L4 Integrity Offload

The AVF driver can offload the following L3 and L4 integrity checks: IPv4 header(s) checksum and SCTP CRC integrity. Tunneled UDP headers and GRE header are not offloaded, and the AVF driver leaves their checksum field as is. In order to request L3 and L4 Integrity Offloads, software should define the packet format and the required offload in the transmit descriptors as shown/listed in Figure 2-1, Figure 2-5 and Table 2-9.

Some rules for integrity offload are:

- Offloads for headers to an IPv6 header with Fragment extension header or fragmented IPv4 packets do not make sense. Note that this rule is not enforced by the hardware.
- IPv6 support — The pseudo header for the L4 checksum takes into account the addresses in the IPv6 header ignoring the optional extension headers. Packets with Routing Header type 2 and Destination Options Header with Home Address option contain an alternative IP address in the extension header. Therefore the operating system should not request checksum nor segmentation offload for packets with routing extension header type 2.
- IP Header Length Requirements (IPLEN):
  - — IPLEN defines the IP header length in Dword units and the IIPT defines the IP header type.
  - — The *Length* field in the IP header is prepared by the software in the packet buffers. This length field includes the payload of the IP header.
  - — For IPv4 it should be at least 20 bytes (basic header size), and not more than 60 bytes.
  - — For IPv6 it should be at least 40 bytes (basic header size), and up to the maximum size enabled by the parsing fields for basic IPv6 header and its extension headers.
- L4 Header Length Requirements (L4LEN):
  - — For TCP, it should be at least 20 bytes (basic header size), and not more than 60 bytes.
  - — For SCTP. it should be set to 12 (SCTP common header size).
  - — For UDP. it should be set to 8 (UDP header size).

Table 2-9 lists all supported packet formats and the processed integrity. Table 2-9 uses the following notations:

- IP is a generic term for IPv4 header or IPv6 header. The IPv4 header can have IP option headers and the IPv6 header can have IPv6 extension headers.
- L4 is a generic term for UDP, TCP or SCTP headers.
- IP checksum is meaningful only for IPv4.
- Checksum is a generic term for UDP and TCP checksum as well as SCTP CRC integrity.

Offload Details:

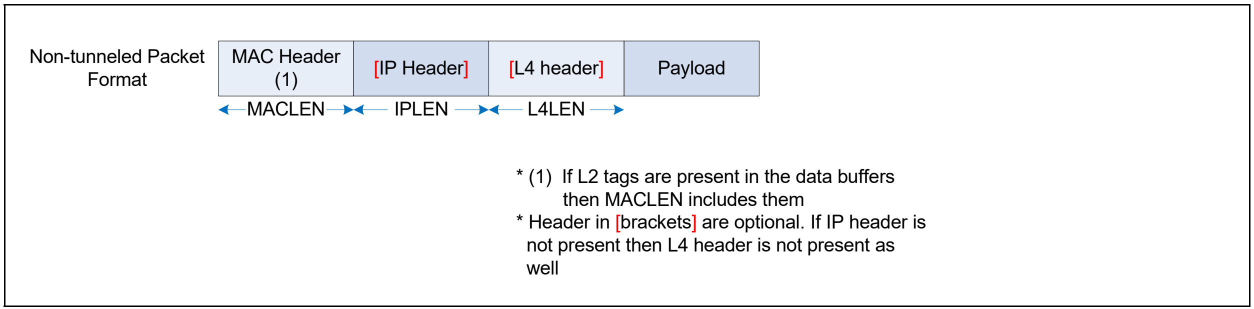
- IPv4 checksum calculation
  - — Software should set the IPv4 checksum to zero
  - — Hardware calculates the IPv4 header checksum starting at the beginning of the IPv4 header up to the end of the header
- UDP and TCP checksum calculation
  - — Software provides the pseudo IP header checksum in the L4 header.
  - — Hardware calculates the L4 checksum starting at the beginning of the L4 offset up to the end of the packet which includes the pseudo header provided by software.
- See Table 2-8 for details of the fields that the software device driver should fill when sending a packet.
- SCTP CRC calculation

— Software should set the CRC in the header to zero.

— Hardware calculates the CRC according to SCTP standard starting at the beginning of the SCTP header up to the end of the packet.

### Table 2-8 Pseudo Header Pre-loaded Values

| Field | Single Send packet (No TSO) | Transmit Segmentation offload (TSO) |
|---|---|---|
| Single IP Length | **IPv4:** IP header +payload size.<br>**IPv6:** payload size only, including header extensions.<br>Should match the data size stored in host memory (hardware modifies it upon offloads). | Don't care (calculated by hardware). |
| Single L4 (UDP/TCP) Length | Header size + payload size<br>Should match the data size that stored in host memories (hardware modifies it upon offloads). | Don't care (calculated by hardware). |
| Single L4 CS Field | IP header pseudo checksum, calculated on IP header fields:<br>**IPv4:** Source addr + destination addr + protocol + L4 length.<br>**IPv6:** Source addr + destination addr + NextHeader + L4 length.<br>Protocol or NextHeader values are: 0x11 for UDP and 0x6 for TCP.<br>***Note:*** L4 Length is the payload that follows the IP header. It includes the size of L4 header and data. | IP header pseudo checksum, calculated on IP header fields. Not including L4 length:<br>**IPv4:** Source addr + destination addr + protocol.<br>**IPv6:** Source addr + destination addr + NextHeader.<br>Protocol or NextHeader values are: 0x11 for UDP and 0x6 for TCP. |



* (1) If L2 tags are present in the data buffers then MACLEN includes them

* Header in [brackets] are optional. If IP header is not present then L4 header is not present as well

### Figure 2-5. Transmit L3 and L4 Header Lengths (in Host Memory) for Integrity Offload

### Table 2-9. Transmit Integrity Offload for Packet Types

| Packet Type | Parsing Hints and Offload Enablement in the Transmit Descriptors (1) (2) | Supported Transmit Checksum Offload |
|---|---|---|
| Fragmented IPv4 or IPv4 -> Unknown | IIPT = 10b or 11b.<br>L4T = 00b (unknown). | IPv4 checksum if IIPT = 11b. |
| Fragmented IPv6 or IPv6 -> Unknown | IIPT = 01b.<br>L4T = 00b (unknown). | None. |
| IP -> L4 | IIPT = 10b or 11b for IPv4.<br>IIPT = 01b for IPv6.<br>IPLEN = the length of the IP header (including IP optional or extension headers).<br>L4T = 11b, 01b, 10b (UDP, TCP, SCTP).<br>L4LEN = L4 header length. | IPv4 checksum if IIPT = 11b:<br>L4 checksum if L4LEN is meaningful. For example, L4T <> 0. |

## 2.2.5.4 Transmit Segmentation Offload (TSO or LSO)

Transmit Segmentation Offload (TSO), also known as Large Send offload (LSO), enables the TCP/IP stack to pass a ULP[1] datagram larger than the Maximum Transmit Unit Size (MTU) to the network device. The AVF driver divides the large ULP datagram to multiple segments according to the MTU size as shown in Figure 2-6. The size of the ULP datagram supported for TSO can be as small as a single byte (transmitted on a single segment) and up to 256 KB (2^18) supporting TSO and giant TSO.

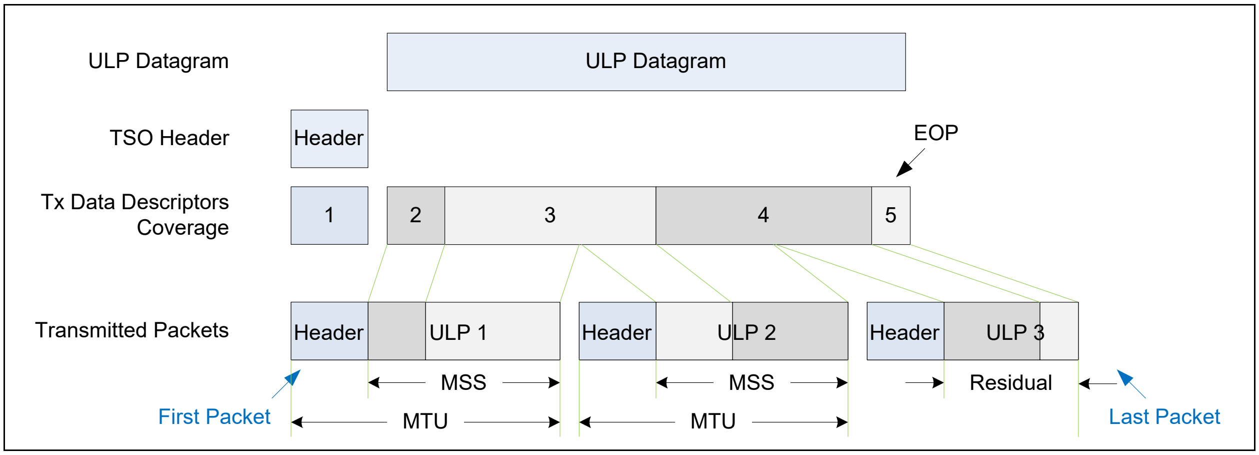


**Figure 2-6. TSO Functionality (Example)**

### 2.2.5.4.1 Frame Formats and Assumptions

The following packet formats are supported for TSO:

**Note:** For all the following formats, IP is IPv4 or IPv6 with/without option/extension headers. L4 is TCP.

- IP -> L4
- IP -> IP -> L4

SNAP packet formats are not supported for TSO.

The following assumptions apply to the TCP segmentation implementation in the AVF driver:

- TSO is activated by setting the TSO flag in the transmit context descriptor. On top of it, software should follow the data and context descriptor settings for integrity offload as listed in Table 2-9.
- The TSO header (L2, L3 and L4) should be provided by a maximum three descriptors, while still allowed to mix header and data in the last header buffer. The maximum size of the TSO header is 512 bytes.
- The maximum size of a single TSO can be as large as 256 KB minus 1 (defined by the *TLEN* field in the transmit context descriptor).
- The *RS* bit in the data descriptor can be set only on the last data descriptor of the TSO (on which the *EOP* bit is set).

[1] ULP = Upper Layer Protocol

- It is assumed that software initializes the pseudo header checksum excluding the TCP length (as opposed to single send on which the pseudo header checksum includes the TCP length).
- An MSS segment should not span more than 8 buffers.

### 2.2.5.4.2 Transmit Segmentation Flow

The TSO flow includes the following:

- The protocol stack receives from an application a ULP datagram to be transmitted.
- The protocol stack calculates the number of packets required to be transmitted based on the MSS.
- The stack interfaces with the device driver and passes the block down with the appropriate header information: Ethernet, IP header(s), and the L4 header.
- The stack interfaces with the device driver and commands the driver to send the entire datagram. The device driver sets up the interface to the hardware (via descriptors) for the segmentation.
- Hardware fetches the segmentation parameters as well as the data and header buffers description by the transmit context and data descriptors.
- Hardware fetches the header and data buffers and then transmits them by segments according to the TSO parameters.
- Dynamic fields set by the software:
  - For IPv4 the IP header checksum should be set to zero.
  - The *Total Length* field in the IP header(s) should be set to zero.
  - The IP ID of the first segment to be transmitted.
  - The pseudo header checksum of the TCP header should be calculated and placed as part of the packet data in the TCP or UDP checksum offset.
  - Initial value of the TCP flags.

- Dynamic fields in the IPv4 header(s) that are modified by hardware:
  - — The *Total Length* field should reflect the IP payload size plus the IP header length. The L4 payload size is MSS for all packets but the last one which contains the rest of the data. So for the inner IP header it is the L4 Payload + L4LEN + IPLEN.
  - — The *Identification* field in the IP header is taken from the TSO header in the first segment and then it is increased by one for each transmitted segment.
  - — The *Identification* field (in the external IP header) is taken from the TSO header in the first segment and it is increased by one for each transmitted segment.
  - — The header checksum is calculated after the other parameters in the IP header are updated.
- Dynamic fields in the IPv6 header(s) that are modified by hardware:
  - — The payload length should reflect the payload size. It is the MSS for all packets but the last one which contains the rest of the data. The payload length should reflect the IP payload size. So for the IP header it is the L4 payload + L4LEN + IP extensions. The IP extensions length equals to IPLEN minus 40.
- Dynamic fields in the TCP header that are modified by hardware:
  - — The sequence number in the TCP header is taken from the TSO header in the first segment. It is then incremented by MSS for each transmitted segment.
  - — The TCP flags are taken from the TSO header. If the TSO is composed of a single segment, then it is processed the same as the last segment of a multiple segment TSO.
  - — The TCP checksum is calculated starting by the initial value (of the pseudo header). The checksum includes the updated TCP header, TCP payload and the calculated TCP length (equals to the payload size plus the TCP header size).

### 2.2.5.4.3 Segmentation Indication to Hardware

Software indicates a TCP segmentation by a transmit context descriptor just before the data descriptor with the following parameters:

- TSO flag is set, indicating a TSO is requested.
- MSS should be set to the required size of the L4 payload on each segment (MTU minus the size of the headers).
  - — MSS smaller than 88 bytes or larger than allowed, are considered malicious. The respective queue is stopped and an interrupt might be issued to the PF.
- TLEN is the total ULP datagram length.
- Other parameters in the data descriptor(s) and the context descriptor are defined the same as a single send.

The data descriptors indicate the TSO header as well as the ULP datagram while following the Frame Formats and Assumptions described in Section 2.2.5.4.1.

**NOTE: This page has been intentionally left blank.**

# 3.0 Interrupts

## 3.1 MSI-X Vectors

MSI-X enables multiple interrupts for the VFs. MSI-X is exposed in the MSI-X capability structure in the PCI configuration space of all VFs. The number of supported MSI-X vectors is defined by the table size parameters in the MSI-X capability structure in the PCI configuration space of the VFs. The table size parameters are loaded for each VF following FLR.

### 3.1.1 Interrupt Enable Procedure

MSI-X is enabled by the *MSI-X Enable* flag in the MSI-X Capability structure in the PCI config space per VF and further enablement by the *Mask* bit per MSI-X vector in the MSI-X Table Structure.

- Interrupt enablement by the AVF driver is via the INTENA flag in the VFINT_DYN_CTL0 and VFINT_DYN_CTLN registers.
  - The software driver sets the INTENA flag and clears WB_ON_ITR to enable the relevant interrupt signal. Upon interrupt assertion, the INTENA flag.
- Interrupt enablement per cause (Section 3.2) is done by the PF and is beyond the scope of this document.
- Interrupt moderation
  - Interrupt Throttling (ITR) is described in Section 3.3.1.

### 3.1.2 Pending Bit Array (PBA)

A bit in the PBA is set to one when an interrupt is triggered internally and cleared when the MSI-X vector is sent on the PCIe bus.

## 3.1.3 Interrupt Sequence

This section describes the interrupt sequence of events starting by an internal event that triggers an interrupt until it is sent to the PCIe bus and the expected software response.

1. Any of the interrupt causes has an event that sets an internal *INTEVENT* flag for the matched interrupt signal.
2. If the interrupt is enabled by *INTENA* and also enabled by the interrupt moderation policy, hardware executes the following steps in the following order:
   a. Scan all queues associated with this interrupt by a linked list explained in Section 3.3. Write back the status of all completed descriptors that were not reported so far and clear the internal *EVENT* flags of these queues.
   b. Set the matched bit in the pending interrupt block array (PBA).
   c. *INTEVENT* and *INTENA* are auto-cleared.
3. If the interrupt is enabled by the operating system (by PCIe setting), the interrupt message is sent to the PCIe.
   — The interrupt indication in the PBA is auto-cleared.
4. During the interrupt handler, software processes each individual interrupt cause. Specific to interrupt zero of the function the other causes interrupt the software can optionally read the ICR0 register identifying the interrupt causes to be processed. Reading the ICR0 register clears; it making it ready to reflect the events of the next interrupt.
5. At the end of the interrupt handler, software re-enables the interrupts by setting INTENA.
   — On the same register, software sets also the *CLEARPBA* flag that clears the matched bit in the PBA.
   — On the same register, software can update one of three ITRs of this interrupt by setting the *ITR_INDX* and *INTERVAL* fields. Setting *ITR_INDX* to 11b does not impact the ITRs. ITR is described in Section 3.3.1.

### 3.1.3.1 MSI-X Interrupts While Interrupts are Disabled by the Operating System

1. Refer to steps 1 and 2 described in the previous section.
2. The operating system polls the PBA and schedules the interrupt handler.
3. Refer to steps 4 and 5 described in the previous section.

## 3.2 Interrupt Causes

This section lists all interrupts causes (sources). The VF is required to request its cause mapping programming from the PF by Admin command using the VIRTCHNL_OP_CONFIG_IRQ_MAP opcode or any other sideband channel that is out of scope of this document.

### 3.2.1 LAN Transmit Queues

Each transmit queue is a potential interrupt cause. A status reporting of a completed descriptor with the *EOP* bit set is considered as a transmit event that can trigger an interrupt (if the interrupt is enabled).

LAN transmit queue 'n' can be enabled for interrupts by the PF. The queues are mapped to any interrupt vector within the function space and mapped to any of it's ITRs (or immediate interrupt). See Section 3.3.1 for more detail.

### 3.2.2 LAN Receive Queues

Each receive queue is a potential interrupt cause. A DMA completion of a descriptor with an *EOP* flag and its buffer is considered as a receive event that triggers an interrupt (if the interrupt is enabled).

Similar to the LAN transmit queues, the LAN receive queues are mapped to any ITR of any interrupt vector.

### 3.2.3 Admin Queues

The AVF driver supports VF Mailbox queues.

A completion of an Admin command on the transmit queue or posting a structure on the receive queue is considered an event for the queue pair that might trigger an interrupt (depending on if it was configured to do so in the relevant descriptor). Mailbox queues are described in Section 4.0.

### 3.2.4 Other Interrupt Causes

The AVF driver supports asynchronous events that can generate interrupts for the VFs. The other interrupt events supported by the VFs are indicated in the VFINT_ICR0 register and enabled by the VFINT_ICR0_ENA register (per VF) as listed in Table 3-1.

The other interrupt cause is mapped to MSI-X vector zero of the function. It is mapped to any of its ITRs (or immediate interrupt) as programmed by the *OTHER_ITR_INDX* field in the VFINT_STAT_CTL0 register for the VF. See Section 3.3.1 for more ITR detail.

During normal operation it is expected that VFINT_ICR0 is used only as a read/clear register by software. Setting the flags in the VFINT_ICR0 register (other than the queue flags and the SWINT flag), emulates an interrupt event of the specific cause (if enabled by the *VFINT_ICR0_ENA* register).

**Table 3-1. Other Interrupt Causes of the VFs**

| PF Other Cause | Description |
|---|---|
| ADMINQ | Send / receive admin queues interrupt. |
| SWINT | Software interrupt (detailed in Section 3.2.5). |

### 3.2.5 Software Initiated Interrupt

In some cases, software might not be able to process all events in a single interrupt handler. In such cases, software might schedule another interrupt to complete processing all interrupt events. Software can trigger an interrupt on any vector and any of its ITRs or NoITR. The software interrupt is mapped to one of three ITRs or immediate interrupt by the *SW_ITR_INDX* field in the VFINT_DYN_CTLx registers (‘x’ stands for ‘0’ or ‘N’). When programming the *SW_ITR_INDX* parameter, the *SW_ITR_INDX_ENA* flag in this register should be set as well.

Software initiates the interrupt by setting the *SWINT_TRIG* flag in the VFINT_DYN_CTLx registers.

Software has the option of setting the *SWINT_TRIG* flag in the VFINT_DYN_CTLx registers with or without setting *INTENA* by using the *INTENA_MSK* flag in the same register. Setting or clearing *INTENA*, not as part of an interrupt handling routine, might lead to race conditions and therefore it is expected that software never clears *INTENA* and clearing of *INTENA* always be done by hardware. Also, it is expected that software sets *SWINT_TRIG* together with *INTENA* only as part of the interrupt handling routine. For example, at the end of handling an interrupt.

## 3.3 Interrupt Moderation and Link List

### 3.3.1 Interrupt Throttling (ITR)

Interrupt Throttling (ITR) is a mechanism that guarantees a minimum gap between two consecutive interrupts (other than possible jitter caused by handling the interrupts). If an event associated with this ITR happens before the ITR expires, the interrupt assertion is delayed until the ITR expires. If the ITR expires before any event associated with this interrupt, the interrupt logic is armed and the interrupt can be asserted the moment the event happens. The ITR intervals per vector are programmed by the VFINT_ITRx registers.

The interrupt causes are mapped to one of the ITRs by the PF. The ITR intervals can be programmed directly to the VFINT_ITRx registers or via the VFINT_DYN_CTLx registers. When any ITR interval of an interrupt with pending event is expired, hardware does the following:

- Clears the other ITRs of the same interrupt
- Processes all causes of the same interrupt (associated to all ITRs) in the linked list

# 4.0 Mailbox

## 4.1 Mailbox Registers and Command Format

The mailbox queue is comprised of a pair of mailbox transmit queues and mailbox receive queues. The AVF driver commands are posted on the mailbox Transmit Queue (ATQ). The mailbox completes the AVF driver commands by writing back onto the command descriptor. Incoming messages are written to the mailbox Receive Queue (ARQ). The AVF driver posts empty buffers to the mailbox Receive Queue (ARQ), and the mailbox fills them with messages.

Both ATQ and ARQ support direct and indirect commands. The mailbox queue direct command is one that fits entirely in the queue descriptor, while an extended or indirect command, is one that uses an additional buffer, which is specified in the descriptor. When a command needs an additional external buffer it marks the *BUF* flag, if the buffer contains data that the mailbox needs to read, the *RD* flag is used, a buffer bigger than 512 bytes (AQ_LARGE_BUF) must have the *LB* flag set. The maximum buffer size supported in this version of the mailbox queues is 4096 bytes.

Both queues use the same descriptor structure. All descriptors and commands are defined using Little Endian notation with 32-bit words. AVF drivers using other conventions should take care to do the proper conversions.

**Table 4-1. Mailbox Queue Descriptor Structure (in LE 32 order)**

| +3 | | | | | | | | +2 | | | | | | | | +1 | | | | | | | | +0 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Opcode | | | | | | | | | | | | | | | | FE | IE | SI | BUF | VFC | RD | LB | Reserved | | | | | VFE | ERR | CMP | DD |
| Return Value | | | | | | | | | | | | | | | | Data Len | | | | | | | | | | | | | | | |
| Cookie High | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Cookie Low | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Param0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Param1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Data address high | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Data address Low | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

### Table 4-2. Mailbox Descriptor Field Descriptions

| Name | Bytes.Bits | Description |
|---|---|---|
| Flags.DD | 0.0 | Set by mailbox to mark entry done. |
| Flags.CMP | 0.1 | Set by mailbox to mark entry as completion. |
| Flags.ERR | 0.2 | Set by mailbox to mark entry as an error indication. |
| Flags.VFE | 0.3 | Set by mailbox to mark entry as an event forwarded from a VF driver. |
| Flags.Reserved | 0.4-1.0 | Reserved, must be zeroed by sender and ignored by receiver. |
| Flags.LB | 1.1 | Set by AVF driver to indicate that indirect buffer is longer than AQ_LARGE_BUF. |
| Flags.RD | 1.2 | Set by AVF driver to indicate that mailbox needs to read indirect buffer. |
| Flags.VFC | 1.3 | Set by AVF driver to indicate command on behalf of a VF. |
| Flags.BUF | 1.4 | This command uses additional data. |
| Flags.SI | 1.5 | Do not interrupt when this command completes. |
| Flags.EI | 1.6 | Interrupt on error - supersedes Flags.SI in case of an error. |
| Flags.FE | 1.7 | If previous command completed in error, flush this one. |
| Opcode | 2-3 | Command opcode (see Table 4-9). |
| Datalen | 4-5 | Indirect data length in bytes (can be used for other uses if Flags.RD is unset). |
| Return value | 6-7 | Return value. Used by mailbox ARQ to reflect the error code as described in Section 4.1.2 and Table 4-4. |
| Cookie High | 8-11 | Opaque data, echoed by receiver, high half. |
| Cookie Low | 12-15 | Opaque data, echoed by receiver, Low half. |
| Param0 | 16-19 | First general use parameter. |
| Param1 | 20-23 | Second general use parameter. |
| Data Address high | 24-27 | Indirect data pointer (can be used for other uses if Flags.RD is unset). |
| Data Address low | 28-31 | Indirect data pointer (can be used for other uses if Flags.RD is unset). |

## 4.1.1 Mailbox Queue CSRs

The mailbox queues have 32-byte descriptors. They are serviced by the following registers (Table 4-3).

### Table 4-3. Mailbox Queue Registers

| Name | Width | Comment |
|---|---|---|
| VF_ATQBAH | 32 bits | High bytes of ATQ base address. |
| VF_ATQBAL | 32 bits | Low bytes of ATQ base address, address must be 64 byte aligned. |
| VF_ATQLEN | 10+4 bits | ATQ length in descriptors, MSB is set for queue enable, three error bits (critical error, overflow error and VF error) are set by mailbox to indicate error conditions (see further Section 4.1.2.1). |
| VF_ATQH | 10 bits | ATQ head pointer (mailbox updates) - might not be implemented. |
| VF_ATQT | 10 bits | ATQ tail pointer (AVF driver updates). |
| VF_ARQBAH | 32 bits | High bytes of ARQ base address. |
| VF_ARQBAL | 32 bits | Low bytes of ARQ base address, address must be 64 byte aligned. |

**Table 4-3. Mailbox Queue Registers**

| Name | Width | Comment |
|---|---|---|
| VF_ARQLEN | 10+4 bits | ARQ length in descriptors, MSB is set for queue enable, three error bits (critical error, overflow error and VF error) are set by mailbox to indicate error conditions (see further Section 4.1.2.1). |
| VF_ARQH | 10 bits | ARQ head pointer (mailbox updates) - might not be implemented. |
| VF_ARQT | 10 bits | ARQ tail pointer (AVF driver updates). |

## 4.1.2 Error Codes

When mailbox completes a command it must use the following error codes.

**Table 4-4. Control Queue Return Values and Error Codes**

| Error Code | Value | Meaning |
|---|---|---|
| (No Error) | 0 | No error (success). |
| EPERM | 1 | Operation not permitted - VF not allowed to access the PF. |
| ENOENT | 2 | No such element. |
| ESRCH | 3 | Bad opcode - Not a mailbox queue opcode or incorrect use of opcode. |
| EINTR | 4 | Operation interrupted. |
| EIO | 5 | I/O error. |
| ENXIO | 6 | No such resource. |
| E2BIG | 7 | Arg too long - buffer longer than 4 KB. |
| EAGAIN | 8 | Try again. |
| ENOMEM | 9 | Out of memory. |
| EACCES | 10 | Permission denied. |
| EFAULT | 11 | Bad address. |
| EBUSY | 12 | Device or resource busy. |
| EEXIST | 13 | Attempt to create something that exists. |
| EINVAL | 14 | Invalid argument - Flags.buf & flags.rd are set, but datalen is equal to zero. |
| ENOTTY | 15 | Not a typewriter. |
| ENOSPC | 16 | No space left or allocation failure.<br>No valid descriptor at the destination queue or the source buffer is bigger than the destination buffer. |
| ENOSYS | 17 | Function not implemented.<br>Destination queue is disabled or the destination function is disabled. |
| ERANGE | 18 | Parameter out of range. |
| EFLUSHED | 19 | Command flushed because a previous command completed in error. |
| BAD_ADDR | 20 | Internal error, descriptor contains a bad pointer. |
| EMODE | 21 | Operation not allowed in current device mode. |
| EFBIG | 22 | File too big. |
| ESBCOMP | 23 | Cannot find enough space for the message in the mailbox queue. |

### 4.1.2.1 Critical Error Indication

- On any error that prevents data placement to a queue, bits ATQLEN.ATQCRIT (or ARQLEN.ARQCRIT) are set by the mailbox and the queue is stopped (by clearing its enable bit), then an interrupt is sent by the mailbox to the AVF driver.
- AVF software reads and reports the error code and then resets the queue.
- If an overflow occurs and a message to the queue is dropped because the queue is full, the mailbox sets ARQLEN.ARQOVFL and interrupts the AVF driver. Note that the mailbox does not stop the queue since, depending on the AVF driver mode, this might be a recoverable error.

**Note:** This error can currently only happen on the receive queue, but to simplify the hardware design the bit is present on both queues.

- When a VF has an event that causes the mailbox to set an error bit in its queue, the mailbox sets ATQLEN.ATQVFE in the corresponding PFs queue and interrupts it. The mailbox does not stop the PF queue in this case.

**Note:** Events and completions that have already been posted before the error are still readable and can be handled by software.

## 4.1.3 Commands Description

### 4.1.3.1 Direct Command

#### 4.1.3.1.1 Direct Command Template

The template for a command that is fully contained in the descriptor and does not need an additional data buffer.

**Table 4-5. Direct command structure**

| Name | Bytes.Bits | Value | Remarks |
|---|---|---|---|
| Flags.DD | 0.0 | 0 | AVF driver must clear. |
| Flags.CMP | 0.1 | 0 | AVF driver must clear. |
| Flags.ERR | 0.2 | 0 | AVF driver must clear. |
| Flags.VFE | 0.3 | 0 | AVF driver must clear. |
| Flags.Reserved | 0.4-1.0 | 0 | Reserved, must be zeroed by sender and ignored by receiver. |
| Flags.LB | 1.1 | 0 | A direct command has no additional buffer. |
| Flags.RD | 1.2 | 0 | A direct command has no additional buffer. |
| Flags.VFC | 1.3 | 0 | AVF driver must clear. |
| Flags.BUF | 1.4 | 0 | A direct command does not have an additional write buffer. |
| Flags.SI | 1.5 | Driver might set | Do not interrupt when this command completes. |
| Flags.EI | 1.6 | Driver might set | Interrupt on error - supersedes Flags.SI in case of error. |

### Table 4-5. Direct command structure (Continued)

| Name | Bytes.Bits | Value | Remarks |
|---|---|---|---|
| Flags.FE | 1.7 | Driver might set | If set, command is flushed if the preceding command resulted in an error. |
| Opcode | 2-3 | Opcode | Command opcode (see Table 4-9). |
| Datalen | 4-5 | 0 | |
| Return Value | 6-7 | | Return value and error code in the completion of the command. |
| Cookie High | 8-11 | Cookie | Opaque value, is copied by the mailbox into the completion of this command. |
| Cookie Low | 12-15 | Cookie | Opaque value, is copied by the mailbox into the completion of this command. |
| Param0 | 16-19 | | First command parameter. |
| Param1 | 20-23 | | Second command parameter. |
| Data Address High | 24-27 | | Can be used for an additional command parameter. |
| Data Address Low | 28-31 | | Can be used for an additional command parameter. |

## 4.1.3.1.2 Direct Command Completion Template

### Table 4-6. Direct Command Completion Event Template

| Name | Bytes.Bits | Value | Remarks |
|---|---|---|---|
| Flags.DD | 0.0 | 1 | Mailbox must set. |
| Flags.CMP | 0.1 | 1 | Mailbox must set. |
| Flags.ERR | 0.2 | 0 or 1 | Mailbox must set only if it reporting an error. |
| Flags.VFE | 0.3 | 0 | Mailbox must clear. |
| Flags.Reserved | 0.4 -1.0 | 0 | Reserved, must be zeroed by sender and ignored by receiver. |
| Flags.LB | 1.1 | Echo | Mailbox copies value from command. |
| Flags.RD | 1.2 | Echo | Mailbox copies value from command. |
| Flags.VFC | 1.3 | Echo | Mailbox copies value from command. |
| Flags.BUF | 1.4 | Echo | Mailbox copies value from command. |
| Flags.SI | 1.5 | Echo | Mailbox copies value from command. |
| Flags.EI | 1.6 | Echo | Mailbox copies value from command. |
| Flags.FE | 1.7 | Echo | Mailbox copies value from command. |
| Opcode | 2-3 | Opcode | Command opcode (see Table 4-9). |
| Datalen | 4-5 | | Can be used for an additional command parameter. |
| Return value | 6-7 | | Mailbox return value 0=no error (for error codes see Table 4-4). |
| Cookie High | 8-11 | Echo | Opaque value, is copied by the mailbox from the command. |
| Cookie Low | 12-15 | Echo | Opaque value, is copied by the mailbox from the command. |
| Param0 | 16-19 | | First command parameter. |
| Param1 | 20-23 | | Second command parameter. |
| Data Address High | 24-27 | | Can be used for an additional command parameter. |
| Data Address Low | 28-31 | | Can be used for an additional command parameter. |

## 4.1.3.2 Indirect Command

### 4.1.3.2.1 Indirect Command Template

An indirect write command uses an additional DMA buffer specified in the descriptor.

The *BUF* flag must be set by the AVF driver. If the buffer is larger than 512 bytes the *LB* flag must be set.

This version of the mailbox is limited to buffers up to 4096 bytes. If the command uses the buffer to pass data the *RD* flag needs to be set.

**Table 4-7. Indirect command template**

| Name | Bytes.Bits | Value | Remarks |
|---|---|---|---|
| Flags.DD | 0.0 | 0 | AVF driver must clear. |
| Flags.CMP | 0.1 | 0 | AVF driver must clear. |
| Flags.ERR | 0.2 | 0 | AVF driver must clear. |
| Flags.VFE | 0.3 | 0 | AVF driver must clear. |
| Flags.Reserved | 0.4-1.0 | 0 | Reserved, must be zeroed by sender and ignored by receiver. |
| Flags.LB | 1.1 | 0 or 1 | Set by the AVF driver if buffer is longer than AQ_LARGE_BUF (512). |
| Flags.RD | 1.2 | 0 or 1 | Set by the AVF driver to indicate that mailbox needs to read indirect buffer. |
| Flags.VFC | 1.3 | 0 | AVF driver must clear. |
| Flags.BUF | 1.4 | 1 | This command uses additional data buffer. AVF driver must set this flag on an indirect command. |
| Flags.SI | 1.5 | Driver might set | Do not interrupt when this command completes. |
| Flags.EI | 1.6 | Driver might set | Interrupt on error - supersedes Flags.SI in case of error. |
| Flags.FE | 1.7 | Driver might set | If set, command is flushed if the preceding command resulted in an error. |
| Opcode | 2-3 | Opcode | Command opcode (see Table 4-9). |
| Datalen | 4-5 | Buffer len | Usable length of additional buffer in bytes. |
| Return Value | 6-7 | | |
| Cookie High | 8-11 | Cookie | Opaque value, is copied by the mailbox into the completion of this command. |
| Cookie Low | 12-15 | Cookie | Opaque value, is copied by the mailbox into the completion of this command. |
| Param0 | 16-19 | | First command parameter. |
| Param1 | 20-23 | | Second command parameter. |
| Data Address High | 24-27 | Buff Addr | High bits of buffer address. |
| Data Address Low | 28-31 | Buff Addr | Low bits of buffer address. |

### 4.1.3.2.2 Indirect Command Completion

When completing an indirect command, firmware overwrites datalen with the actual length of data returned by the command.

**Table 4-8. Indirect Command Completion Template**

| Name | Bytes.Bits | Value | Remarks |
|---|---|---|---|
| Flags.DD | 0.0 | 1 | Mailbox must set. |
| Flags.CMP | 0.1 | 1 | Mailbox must set. |
| Flags.ERR | 0.2 | 0 or 1 | Mailbox must set only if it reporting an error. |
| Flags.VFE | 0.3 | 0 | Mailbox must clear. |
| Flags.Reserved | 0.4 -1.0 | 0 | Reserved, must be zeroed by sender and ignored by receiver. |
| Flags.LB | 1.1 | Echo | Mailbox copies value from command. |
| Flags.RD | 1.2 | Echo | Mailbox copies value from command. |
| Flags.VFC | 1.3 | Echo | Mailbox copies value from command. |
| Flags.BUF | 1.4 | Echo | Mailbox copies value from command. |
| Flags.SI | 1.5 | Echo | Mailbox copies value from command. |
| Flags.EI | 1.6 | Echo | Mailbox copies value from command. |
| Flags.FE | 1.7 | Echo | Mailbox copies value from command. |
| Opcode | 2-3 | Opcode | Command opcode (see Table 4-9). |
| Datalen | 4-5 |  | Can be used for an additional command parameter. |
| Return Value | 6-7 |  | Mailbox returns value 0=no error (for error codes see Table 4-4). |
| Cookie High | 8-11 | Echo | Opaque value, is copied by firmware from the command. |
| Cookie Low | 12-15 | Echo | Opaque value, is copied by firmware from the command. |
| Param0 | 16-19 |  | First command parameter. |
| Param1 | 20-23 |  | Second command parameter. |
| Data Address High | 24-27 |  | Can be used for an additional command parameter. |
| Data Address Low | 28-31 |  | Can be used for an additional command parameter. |

## 4.2 Command opcodes

Opcodes are 16 bits, the following opcodes are relevant for the mailbox queue.

**Table 4-9. Mailbox Commands**

| Name | Opcode | Ref | Type |
|---|---|---|---|
| Send to PF | 0x0801 | Section 4.4.1 | Indirect /Direct |
| Message from PF | 0x0802 | Section 4.4.2 | Indirect /Direct |
| Queue Shutdown | 0x0003 | Section 4.4.3 | Direct |

# 4.3 Mailbox Initialization Flow

When initializing the queue, the AVF driver must do the following:

- The AVF driver allocates and set ups appropriately sized host memory for the queues.
- The AVF driver must post initialized buffers to the receive queue before it can use the transmit queue (see Section 4.3.1 for receive queue element initialization).
- The AVF driver clears the head and tail registers for each queue (head registers are VF_ATQH and VF_ARQH. Tail registers are VF_ATQT and VF_ARQT).
- The AVF driver then programs the base and length registers for each queue (VF_ATQBAL, VF_ATQBAH, VF_ATQLEN, VF_ARQBAL, VF_ARQBAH and VF_ARQLEN) and sets Length.Enable to 1b to signal the mailbox that the queue is now enabled.

## 4.3.1 Receive Queue Element Initialization by the AVF Driver

The AVF driver must clear any unused fields (including unused flags) and set data pointers and data length to a mapped DMA pointer.

The AVF driver might set the *SI* and the *EI* flags in the receive queue element. The AVF driver must not set the *FE* flag on receive queue elements.

**Table 4-10. Receive Queue Element - Initial Values**

| Name | Bytes.Bits | Value | Remarks |
|---|---|---|---|
| Flags.DD | 0.0 | 0 | AVF driver must clear. |
| Flags.CMP | 0.1 | 0 | AVF driver must clear. |
| Flags.ERR | 0.2 | 0 | AVF driver must clear. |
| Flags.VFE | 0.3 | 0 | AVF driver must clear. |
| Flags.Reserved | 0.4 1.0 | 0 | Reserved, must be zeroed by sender and ignored by receiver. |
| Flags.LB | 1.1 | 0 or 1 | Set by the AVF driver if buffer is longer than AQ_LARGE_BUF (512 bytes). |
| Flags.RD | 1.2 | 0 | Not applicable to receive queue. |
| Flags.VFC | 1.3 | 0 | AVF driver must clear. |
| Flags.BUF | 1.4 | 1 | Receive queue elements always have an additional buffer. |
| Flags.SI | 1.5 | Driver might set | Do not interrupt when this command completes. |
| Flags.EI | 1.6 | Driver might set | Interrupt on error - supersedes Flags.SI in case of error. |
| Flags.FE | 1.7 | 0 | AVF driver must clear. |
| Opcode | 2-3 | | AVF driver must clear. |
| Datalen | 4-5 | Buffer Len | Additional data length in bytes. |
| Return Value | 6-7 | | AVF driver must clear. |
| Cookie High | 8-11 | | AVF driver must clear. |
| Cookie Low | 12-15 | | AVF driver must clear. |

**Table 4-10. Receive Queue Element - Initial Values (Continued)**

| Name | Bytes.Bits | Value | Remarks |
|---|---|---|---|
| Param0 | 16-19 |  | AVF driver must clear. |
| Param1 | 20-23 |  | AVF driver must clear. |
| Data Address High | 24-27 | Buffer ADDR | indirect data pointer (can be used for other uses if Flags.RD is unset). |
| Data Address Low | 28-31 | Buffer ADDR | indirect data pointer (can be used for other uses if Flags.RD is unset). |

The values written by the mailbox when it uses the EAQ element are discussed in the sections that follow.

### 4.3.2 Driver Unload and Queue Shutdown

When shutting down the mailbox queue the AVF driver:

- Posts a Queue Shutdown command (0x0003). See Section 4.4.3 for more detail. In this command, the AVF driver sets the *Driver Unloading* flag if it intends to unload.
- Software must not send any additional commands on the queue until the flow completes
- Mailbox waits for any pending DMA transactions it generated to be acknowledged by hardware.
- Closes the Rx queue by clearing its *Enable* bit.
- Sends a completion to the AVF driver as usual, honoring all the interrupt control bits in the descriptor.

Software then closes the Tx queue by clearing its *Enable* bit.

If the AVF driver is unloading, it issues a function reset (VFR) to the device.

Note that before the mailbox queues are re-enabled, software should clear the head and tail registers of the transmit and receive mailbox queue.

## 4.4 Mailbox Commands

### 4.4.1 Send Message to PF Command

This command, together with the next one, implements a communication channel between PFs and their VFs. The data in the external buffer is copied into the buffer linked to the message sent on the PF receive queue. The command completes once the data is copied.

Since the value of the cookie is copied to the event, if 8 bytes are enough for needed message, the AVF driver might specify a length of zero, and not use an external buffer. In this case it should also not set the *BUF* flag.

**Note:** This version of the specification supports messages of up to 4096 bytes.

### Table 4-11. Send to PF command (Opcode: 0x0801)

| Name | Bytes.Bits | Value | Remarks |
|---|---|---|---|
| Flags | 1:0 | | See Section 4.1.3.1.1 or Section 4.1.3.2.1 for details. |
| Opcode | 2-3 | 0x0801 | Command opcode. |
| Length | 4-5 | buffer length | Length of message. |
| Return value/VFID | 6-7 | | Return value. Zeroed by the AVF driver. Written by firmware mailbox. |
| Cookie High | 8-11 | Cookie | Opaque value, is copied by the mailbox into the completion of this command. It is also copied to the descriptor of the target mailbox queue. |
| Cookie Low | 12-15 | Cookie | Opaque value, is copied by the mailbox into the completion of this command. It is also copied to the descriptor of the target mailbox queue |
| Reserved | 16-19 | | |
| Reserved | 20-23 | | |
| Data Address high | 24-27 | 0x0 | Address of data buffer. |
| Data Address low | 28-31 | 0x0 | Address of data buffer. |

After posting the event to the PF mailbox receive queue, the mailbox completes this command by updating the flags and returns a value (zero for success).

## 4.4.2 Incoming Message From PF Mailbox

This section describes the incoming messages on the VF mailbox receive queue. Messages that are generated by the PF toward the VF are copied by the mailbox into the VF mailbox receive queue. After posting the message to the VF mailbox receive queue, the mailbox will interrupt the VF for incoming message.

### Table 4-12. Message from PF (Opcode: 0x0802)

| Name | Bytes.Bits | Value | Remarks. |
|---|---|---|---|
| Flags | 1:0 | | See Section 4.1.3.1.2 or Section 4.1.3.2.2 for details. |
| Opcode | 2-3 | 0x0802 | Event code. |
| Length | 4-5 | buffer length | Length of message. |
| Return Value/ VFID | 6-7 | | Reserved. |
| Cookie High | 8-11 | Cookie | Opaque value is copied by the mailbox from the source descriptor. |
| Cookie Low | 12-15 | Cookie | Opaque value is copied by the mailbox from the source descriptor. |
| Reserved | 16-19 | | |
| Reserved | 20-23 | | |
| Data Address High | 24-27 | 0x0 | Address of data buffer. |
| Data Address Low | 28-31 | 0x0 | Address of data buffer. |

## 4.4.3 Queue Shutdown Command

This is the final command posted to the queue, closing the queue as described in Section 4.3.2. When this command completes, the AVF driver is allowed to free any host resources associated with the mailbox queue.

If the AVF driver is going to unload it must set the *Driver Unloading* flag to inform the mailbox.

Once this command is posted, the AVF driver is not allowed to issue any more commands on the queue before a reset is done.

**Note:** Interrupt generation and the interrupt control flags in this command are handled as usual by the mailbox. This means that if an interrupt was not inhibited by setting the *SI* flag, it happens. If the AVF driver is in polling mode and cannot handle an interrupt, it needs to either inhibit the interrupt or have interrupts disabled through the interrupt control registers.

**Table 4-13. Queue Shutdown Command (Opcode: 0x0003)**

| Name | Bytes.Bits | Value | Remarks. |
|---|---|---|---|
| Flags | 1-0 | | See Section 4.1.3.1.1 for details. |
| Opcode | 2-3 | 0x0003 | Command opcode. |
| Datalen | 4-5 | 0 | No external data. |
| Return value/VFID | 6-7 | | Return value. Zeroed by the AVF driver. Written by firmware. |
| Cookie High | 8-11 | Cookie | Opaque value, is copied by firmware into the completion of this command. |
| Cookie Low | 12-15 | Cookie | Opaque value, is copied by firmware into the completion of this command. |
| Driver unloading | 16.0 | | 1b if the AVF driver intends to unload, 0b otherwise. |
| Reserved | 17-31 | 0x0 | Reserved. |

**NOTE:  This page has been intentionally left blank.**

# 5.0 Extensibility Features

This section includes features that are not available in the basic AVF driver, but might be negotiated between the AVF driver and the PF.

## 5.1 Advanced Rx Offloads

See Section 2.2 for more detail.

## 5.2 RDMA

RDMA is an advanced feature not available in the base AVF driver and is covered in future versions of this specification. It is exposed in the capability exchange as VIRTCHNL_VF_OFFLOAD_IWARP.

**NOTE: This page has been intentionally left blank.**

# 6.0 Software Flows

## 6.1 Init Flow

Since the AVF driver is a dependent driver and needs to communicate with the PF driver for its initialization it can take a few seconds to fully initialize. Software initialization for the AVF driver happens in two steps.

- First, it registers a network Interface with the operating system and completes the init/probe call from the operating system.
- Second, it postpones all of its device initialization to a delayed work queue.

As part of work queue:

- It first checks to see if the VF instance is out of reset by checking the VFGEN_RSTAT register to see if the VIRTCHNL_VFR_VFACTIVE or VIRTCHNL_VFR_COMPLETED state is set. This register is written by the PF to inform the VF when it's out of reset.
- It then initializes a hardware mailbox to communicate to the PF to finish the remaining configuration.
- After initializing the mailbox send queue and receive queue buffers and ring structures with DMA-able memory, the software driver initializes the mailbox ring state registers in hardware. The mailbox initialization flow is described in Section 4.3. The following registers get initialized as part of mailbox setup (see Section 7.2.3):
- VF_ATQT
- VF_ATQH
- VF_ATQLEN
- VF_ATQBAL
- VF_ATQBAH
- VF_ARQT
- VF_ARQH
- VF_ARQLEN
- VF_ARQBAL
- VF_ARQBAH

At this point, the VF driver starts sending mailbox messages to the PF. The messages sent by the VF driver are asynchronously responded to by the PF driver. The responses coming from the PF driver are posted in the mailbox receive queue.

The first software opcode (VIRTCHNL_OP_VERSION) sent to the PF is to verify the version for the communication channel (virtchnl) used by the VF and PF.

- The VF sends its version number. In response, the PF sends it's virtchannel API version number. For AVF, the version for virtchnl used by PF and VF must be 1.1 or higher. A minor version mismatch, where the PF is running a version lower than the VFs API version, is ignored. A major version mismatch prevents the VF driver from loading on the device.

## 6.1.1 PF-VF Capability Exchange

The VF sends a config message to the PF (VIRTCHNL_OP_GET_VF_RESOURCES), this is part of a capability exchange between the PF and VF.

- The VF driver sends a list of offloads/capabilities that it can support to the PF driver. In exchange, the PF driver sends the offloads/capabilities that it can enable for this VF. The list supported by PF can be a subset of the ones requested by VF driver. After receiving the response from the PF driver, the VF driver must enable only the offloads that the PF agreed to support.
- Some of the offloads are treated as a base set of offloads/capabilities that all PFs must support for the AVF driver to work on all devices that support the AVF driver. The base offloads are:
  - VIRTCHNL_VF_OFFLOAD_L2
  - VIRTCHNL_VF_OFFLOAD_RSS_PF
  - VIRTCHNL_VF_OFFLOAD_VLAN
  - VF_OFFLOAD_RX_POLLING (for PMD drivers)
- Some of the advanced offloads that the PF and VF might decide to support are:
  - VIRTCHNL_VF_OFFLOAD_RSS_AQ
  - VIRTCHNL_VF_OFFLOAD_RSS_REG
  - VIRTCHNL_VF_OFFLOAD_WB_ON_ITR
  - VIRTCHNL_VF_OFFLOAD_RSS_PCTYPE_V2
  - VIRTCHNL_VF_OFFLOAD_ENCAP
  - VIRTCHNL_VF_OFFLOAD_ENCAP_CSUM
  - VIRTCHNL_VF_OFFLOAD_IWARP

  Further advanced offloads are added as exiting silicon exposes new offloads in the VF or when newer hardware devices get supported through the AVF driver.
- As a response to the VF config message request to the PF, the PF sends not just the offloads it can support but also other config information for the VF such as the number of queues and vectors it can use, the MAC address for the VF, etc.
- The VF driver then requests the operating system to enable the maximum number of vectors it intends to use.

## 6.1.2 Tx and Rx Initialization

At this point the VF driver allocates some software backing structures for data Tx and Rx queues:

- If the VF driver enables the Tx/Rx interrupt feature, it sends a message to the PF with opcode VIRTCHNL_OP_CONFIG_IRQ_MAP to set up the cause to interrupt mapping.
- The remainder of the Tx and Rx initialization happens as part of the interface being brought up explicitly from the operating system by the user.
  - The VF driver allocates DMA memory for Tx and Rx queues for the rings and the buffers associated with the queues.
  - The VF driver sends a virtchnl message (VIRTCHNL_OP_CONFIG_VSI_QUEUES) to set up the Tx and Rx queues. The information sent in this message enables the PF to know about:
    - vsi_id (relative to the VF's VSIs),
    - queue_id (relative to the VF's queues),
    - ring_len,
    - dma_ring_addr (guest physical address for where the ring starts)

- For Rx queues, in addition to the text previously mentioned, the VF driver also sends to the PF the following:
    - max_pkt_size
    - databuffer_size
- This information is used by the PF driver to setup the queue context for each of the queues on behalf of the VF driver.
- The VF driver requests the PF driver to add the MAC filter by using the VIRTCHNL_OP_ADD_ETHER_ADDRESS software opcode over the virtchnl.
- The VF driver sends a message to the PF with opcode VIRTCHNL_OP_ENABLE_QUEUES to enable Rx and Tx queues.
- The VF driver enables interrupts by writing to VFINT_DYN_CTLN1 registers.

At this stage, the VF driver is ready to receive and transmit packets.

# 6.2 AVF Driver PF-VF Software Protocol

The AVF driver relies on a mailbox mechanism to communicate with the PF driver for its configuration. The mailbox format used by the VF driver is a queue format with a generic message format.

The hardware mailbox mechanism used as the hardware framework of the PF-VF software protocol and the generic message format are described in Section 4.0.

## 6.2.1 Software Protocol

Above the hardware generic message format, there is a software protocol defined to request or provide different config information from/to the PF driver. The communication between the PF and VF driver is asynchronous. The VF driver sends a request over the mailbox with a software opcode that is agreed by the PF and VF using shared protocol definition, the request is acknowledged as DMA-ed by hardware. The response from the PF flows back as a separate message with the same opcode to the VF driver, which is received in the mailbox receive queue. Software opcodes that the AVF driver uses are extensible and are divided into two categories:

- Base opcodes — All PF drivers that support the AVF driver must support these opcodes.
- Advanced opcodes — These opcodes might/might not be supported by the PF driver depending on the offloads that the PF decides to enable given the device underneath. Newer ones are added as AVF functionality expands and newer devices that can support more offloads get supported on AVF.

The software mailbox message is of the type virtchnl_msg that is overlaid on top of the hardware mailbox descriptor format.

```
struct virtchnl_msg {
/* HW AQ flags/opcode/len/retval fields */
        u8 pad[8];
/* avoid confusion with desc->opcode */
        enum virtchnl_ops v_opcode;
/* ditto for desc->retval */
```

```
        enum virtchnl_status_code v_retval;
/* used by PF when sending to VF */
        u32 vfid;
        };
```

With every opcode there is a specific data structure that carries further information in the data buffer payload. This is passed as an indirect data buffer attached to the mailbox descriptor. Appendix A describes the different data structures used for each opcode.

#### 6.2.1.1 Software Opcodes

The AVF driver communicates with the PF driver using software opcodes. Each opcode has a unique data structure that goes along and contains more information to describe the request or a response.

List of Opcodes:

```
        VIRTCHNL_OP_UNKNOWN = 0,
        VIRTCHNL_OP_VERSION = 1,
        VIRTCHNL_OP_RESET_VF = 2,
        VIRTCHNL_OP_GET_VF_RESOURCES = 3,
        VIRTCHNL_OP_CONFIG_TX_QUEUE = 4,
        VIRTCHNL_OP_CONFIG_RX_QUEUE = 5,
        VIRTCHNL_OP_CONFIG_VSI_QUEUES = 6,
        VIRTCHNL_OP_CONFIG_IRQ_MAP = 7,
        VIRTCHNL_OP_ENABLE_QUEUES = 8,
        VIRTCHNL_OP_DISABLE_QUEUES = 9,
        VIRTCHNL_OP_ADD_ETH_ADDR = 10,
        VIRTCHNL_OP_DEL_ETH_ADDR = 11,
        VIRTCHNL_OP_ADD_VLAN = 12,
        VIRTCHNL_OP_DEL_VLAN = 13,
        VIRTCHNL_OP_CONFIG_PROMISCUOUS_MODE = 14,
        VIRTCHNL_OP_GET_STATS = 15,
        VIRTCHNL_OP_RSVD = 16,
        VIRTCHNL_OP_EVENT = 17, /* must ALWAYS be 17 */
        VIRTCHNL_OP_IWARP = 20, /* advanced opcode */
        VIRTCHNL_OP_CONFIG_IWARP_IRQ_MAP = 21, /* advanced opcode */
        VIRTCHNL_OP_RELEASE_IWARP_IRQ_MAP = 22, /* advanced opcode */
        VIRTCHNL_OP_CONFIG_RSS_KEY = 23,
        VIRTCHNL_OP_CONFIG_RSS_LUT = 24,
        VIRTCHNL_OP_GET_RSS_HENA_CAPS = 25,
        VIRTCHNL_OP_SET_RSS_HENA = 26,
```

### 6.2.1.2 Error Codes

Every mailbox message is completed asynchronously by the PF with one of these error codes:

```
enum virtchnl_status_code {
        VIRTCHNL_STATUS_SUCCESS        = 0,
        VIRTCHNL_ERR_PARAM                 = -5,
        VIRTCHNL_STATUS_ERR_OPCODE_MISMATCH = -38,
        VIRTCHNL_STATUS_ERR_CQP_COMPL_ERROR  = -39,
        VIRTCHNL_STATUS_ERR_INVALID_VF_ID  = -40,
        VIRTCHNL_STATUS_NOT_SUPPORTED  = -64,
};
```

### 6.2.1.3 Offloads/Capabilities

As mentioned earlier, the PF and VF driver negotiate offloads and capabilities through the VIRTCHNL_OP_GET_VF_RESOURCES opcode.

**Note:** VIRTCHNL_VF_OFFLOAD_RX_POLLING is part of the base set for poll mode AVF driver.

The base offloads are:

```
VIRTCHNL_VF_OFFLOAD_L2
VIRTCHNL_VF_OFFLOAD_RSS_PF
VIRTCHNL_VF_OFFLOAD_VLAN
```

Some of the advanced/optional offloads that PF and VF might decide to support are:

```
VIRTCHNL_VF_OFFLOAD_RSS_AQ
VIRTCHNL_VF_OFFLOAD_RSS_REG
VIRTCHNL_VF_OFFLOAD_WB_ON_ITR
VIRTCHNL_VF_OFFLOAD_RSS_PCTYPE_V2
VIRTCHNL_VF_OFFLOAD_ENCAP
VIRTCHNL_VF_OFFLOAD_ENCAP_CSUM
VIRTCHNL_VF_OFFLOAD_IWARP
```

The list of advanced offloads and software opcodes to help facilitate those offloads keep increasing as AVF evolves and more hardware devices get supported under the AVF umbrella. Once again the AVF driver carries the same device ID even when supporting virtual devices on newer hardware. The advantage of the PF-VF negotiation mechanism is that the AVF driver continues supporting increased offload functionalities without ever losing the basic network interface capability.

### 6.2.1.4 Adding New Offloads/Capabilities

If the AVF driver needs to be expanded to expose additional offloads, users must follow these steps:

- Define a new type VIRTCHNL_VF_OFFLOAD_XX in the common virtchnl.h header file.

- Change the VF driver to start advertising the new offload that it would like to enable by changing the offload bitmap sent to the PF using the VIRTCHNL_OP_GET_VF_RESOURCES message.
- The PF driver that should support this offload for the VF should be updated to enable this new offload in response to the VIRTCHNL_OP_GET_VF_RESOURCES from the VF.
- New opcodes might have to be added to support this offload to further get config information from the PF driver to enable this new offload. An example could be that the VF driver wants to support inline crypto offload. In which case the VF driver has to know about how many security associations it can program in hardware, the type of crypto offload that the device supports, etc.
- These new opcodes have to be called by the VF driver at init or runtime to set up this new offload.
- The PF driver has to be updated to set up/enable hardware for the VF to start exercising these offloads.
- The PF driver also has to send the appropriate config information to the VF driver using these new opcodes.

**Note:** When adding new advanced offload/capabilities and related opcodes to the AVF driver, the virtchannel version does not need to be updated.

### 6.2.2 Event Handling

The VIRTCHNL_OP_EVENT opcode is used by the PF to inform the VF driver of events that might affect it. The AVF driver doesn’t need to send a response after receiving the event. It should handle the event according to the event type, and might inform applications if needed.

## 6.3 PMD Driver for AVF

Instead of standard kernel drivers in the operating system, an AVF might also be driven by polling mode drivers (such as DPDK).

This section provides a high level description of the software flow used by the polling mode driver.

### 6.3.1 Initialization/Pre-configure Flow

The PMD driver initialization flow is similar to the regular driver initialization flow described in Section 6.1.

#### 6.3.1.1 PF-VF Capability Exchange

The exchange behavior obeys the rule defined in Section 6.1.1. A PF or AVF driver supporting polling mode should expose the VIRTCHNL_VF_OFFLOAD_RX_POLLING capability.

## 6.3.2 Tx and Rx Initialization

The PMD Tx and Rx initialization flow is similar to the regular driver initialization flow described in Section 6.1.2.

The association of queues to interrupts, and interrupt enable steps are skipped.

## 6.3.3 Receive and Transmit Flows

After device start up, the device is ready to receive and transmit packets.

AVF drivers can read/write transmit and receive registers (see Section 7.2.4) directly. If interrupt mode is enabled, interrupt enable/disable can be done by setting VF interrupts registers directly (see Section 7.2.2).

Rx burst and Tx burst are the basic receive and transmit functions in the PMD. They access the Rx and Tx descriptors directly without interrupts to quickly receive, process and deliver packets in the user's application. These are burst-oriented functions, thus multiple buffers can be allocated, and sometimes freed, at once, removing per-packet overhead. The burst size depends on the application.

In the Rx_burst and Tx_burst functions, there are some threshold parameters that are introduced to reduce overhead, such as:

- rx_free_thresh — Max free Rx descriptors to hold. Used to define when the Rx tail should be updated.
- tx_rs_thresh — Used to define if *RS* bit should be set in Tx descriptors.
- tx_free_thresh — Start freeing Tx buffers if there are less free descriptors than this value.

### 6.3.3.1 Receive Burst

Figure 6-1 shows the receive burst flow in the PMD driver.

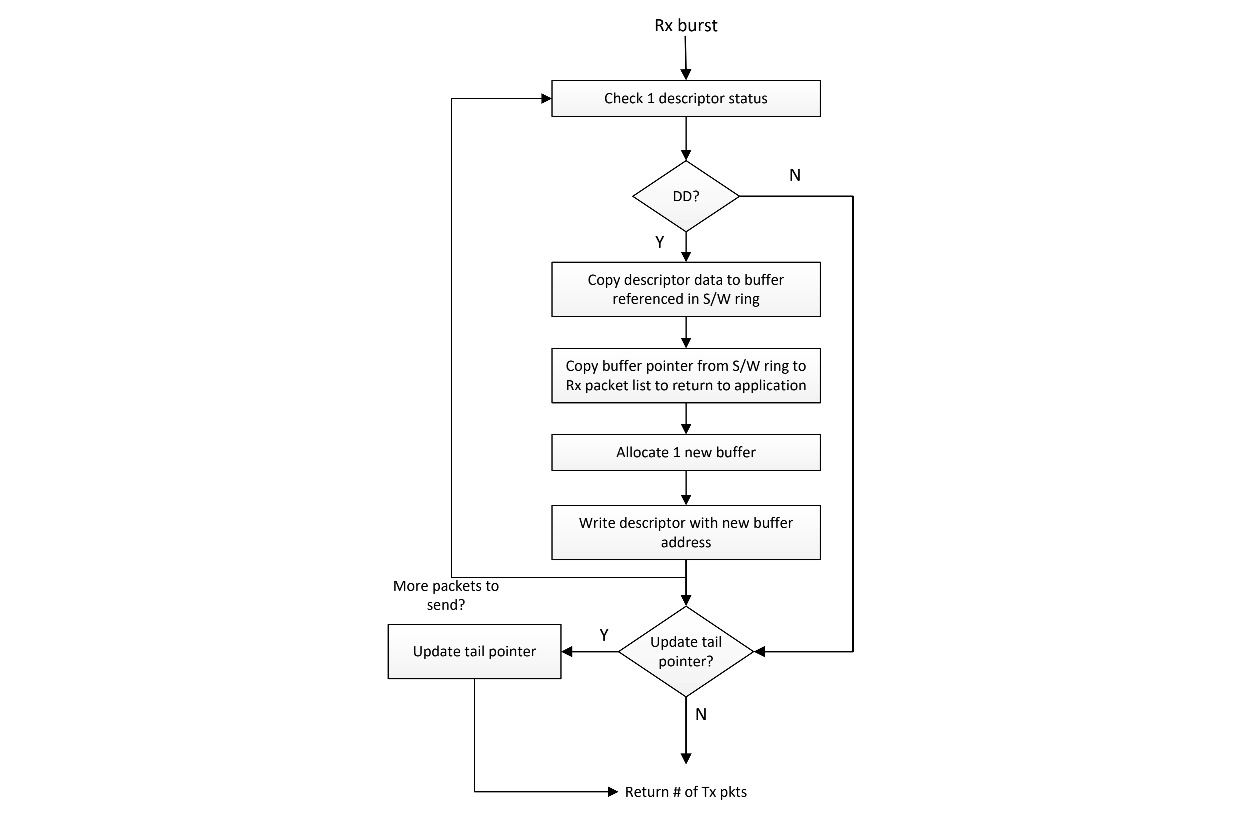

Rx burst
Check 1 descriptor status
DD?
N
Y
Copy descriptor data to buffer referenced in S/W ring
Copy buffer pointer from S/W ring to Rx packet list to return to application
Allocate 1 new buffer
Write descriptor with new buffer address
More packets to send?
Update tail pointer
Y
Update tail pointer?
N
Return # of Tx pkts

**Figure 6-1 Receive Burst Flow**

### 6.3.3.2 Transmit Burst Flow

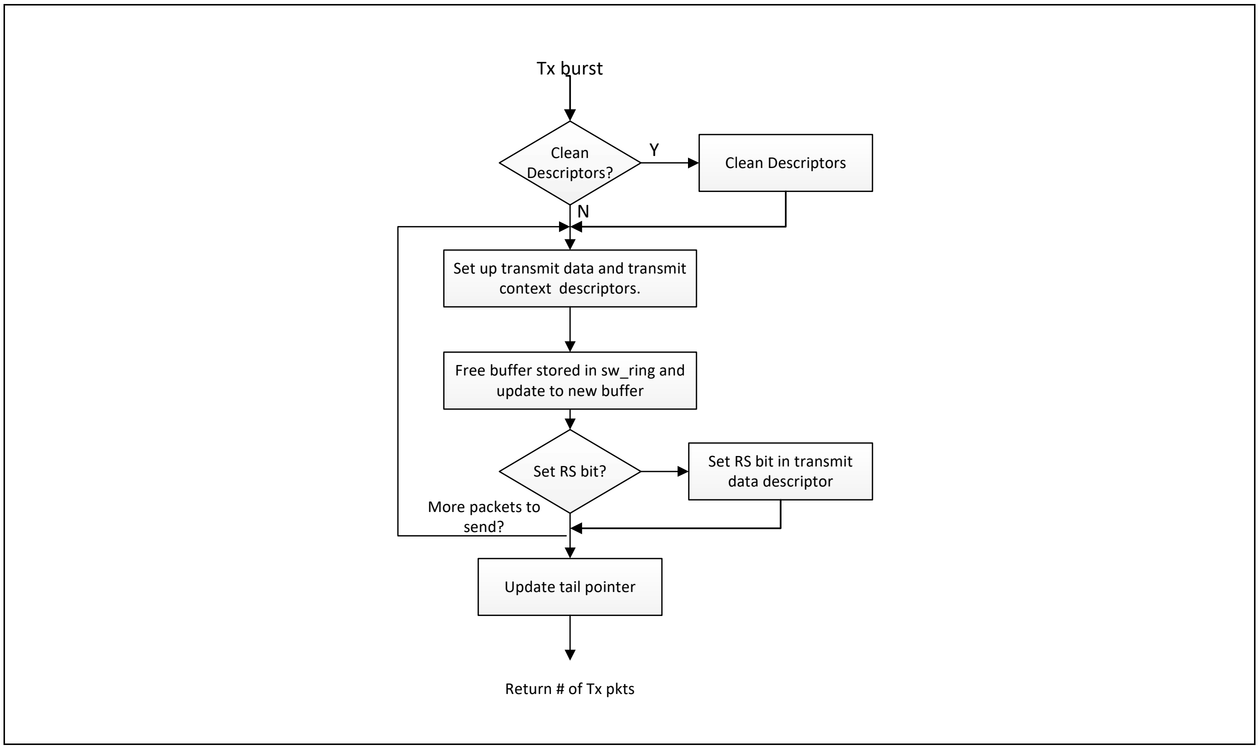

Tx burst
Clean Descriptors?
Y
Clean Descriptors
N
Set up transmit data and transmit context descriptors.
Free buffer stored in sw_ring and update to new buffer
Set RS bit?
Set RS bit in transmit data descriptor
More packets to send?
Update tail pointer
Return # of Tx pkts

**Figure 6-2 Transmit Burst Flow**

## 6.3.4 DPDK Forwarding Application Example

The DPDK environment for packet processing applications allows for two models, run-to-completion and pipe-line:

- In the run-to-completion model, a port's Rx descriptor ring is polled for packets through an API. Packets are then processed on the same core and placed on a port's Tx descriptor ring through an API for transmission. In this synchronous model, each logical core assigned to the DPDK executes a packet processing.
- In the pipe-line model, one core polls one or more port's Rx descriptor ring through an API. Packets are received and passed to another core via a ring. The other core continues to process the packet which then might be placed on a port's Tx descriptor ring through an API for transmission.

Avoiding lock contention is a key issue in a multi-core environment. To address this issue, PMDs are designed to work with per-core private resources as much as possible. For example, a PMD maintains a separate transmit queue per-core, per-port.

Figure 6-3 shows a simple forward application example in run-to-completion model.

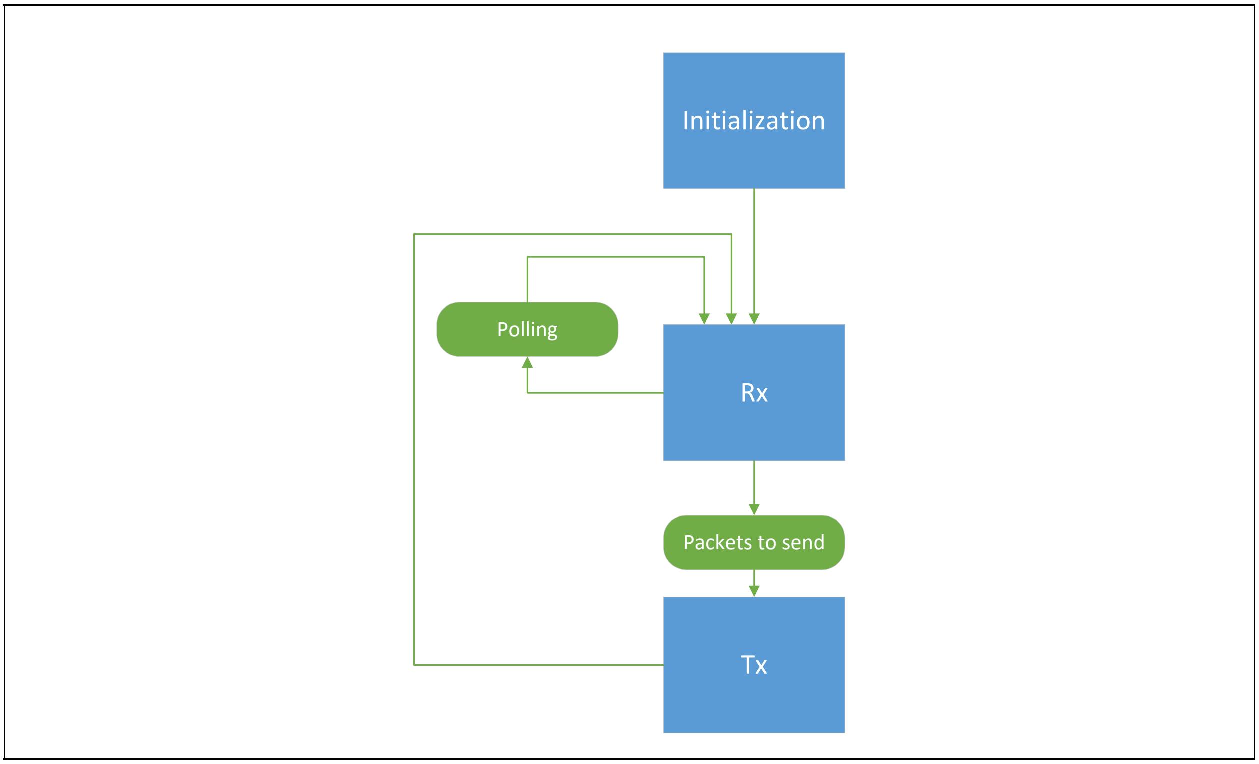


**Figure 6-3 Run to Complete Flow**

1. Initialization.
    a. Init memory zones, pools and devices.
    b. Setup device queues and start devices.
    c. Start packet forwarding application.
2. Packet reception (Rx).
    a. Poll devices' Rx queues and receive packets in bursts.
    b. Allocate new Rx buffers from per queue memory pools to stuff into descriptors.
3. Packet transmission (Tx).
    a. Transmit the received packets from Rx.
    b. Free the buffers that we used to store the packets.

# 7.0 Device Registers - VF

## 7.1 BAR0 Registers Summary

### Table 7-1. BAR0 Registers Summary

| Offset / Alias Offset | Abbreviation | Name | Section |
|---|---|---|---|
| **VF - General Registers** | | | |
| 0x00008800 | VFGEN_RSTAT | VF Reset Status | Section 7.2.1.1 |
| **VF - Interrupts** | | | |
| 0x00005C00 | VFINT_DYN_CTL0 | VF Interrupt Dynamic Control Zero | Section 7.2.2.1 |
| 0x00003800 + 0x4*n, n=0...63 | VFINT_DYN_CTLN[n] | VF Interrupt Dynamic Control N | Section 7.2.2.2 |
| 0x00004C00 + 0x4*n, n=0...2 | VFINT_ITR0[n] | VF Interrupt Throttling Zero | Section 7.2.2.3 |
| 0x00002800 + 0x4*n + 0xC*m, n=0...2, m=0...63 | VFINT_ITRN[n,m] | VF Interrupt Throttling N | Section 7.2.2.4 |
| **VF - Control Queues** | | | |
| 0x00007C00 | VF_ATQBAL | VF MailBox Transmit Queue Base Address Low | Section 7.2.3.1 |
| 0x00007800 | VF_ATQBAH | VF MailBox Transmit Queue Base Address High | Section 7.2.3.2 |
| 0x00006800 | VF_ATQLEN | VF MailBox Transmit Queue Length | Section 7.2.3.3 |
| 0x00006400 | VF_ATQH | VF MailBox Transmit Head | Section 7.2.3.4 |
| 0x00008400 | VF_ATQT | VF MailBox Transmit Tail | Section 7.2.3.5 |
| 0x00006C00 | VF_ARQBAL | VF MailBox Receive Queue Base Address Low | Section 7.2.3.6 |
| 0x00006000 | VF_ARQBAH | VF MailBox Receive Queue Base Address High | Section 7.2.3.7 |
| 0x00008000 | VF_ARQLEN | VF MailBox Receive Queue Length | Section 7.2.3.8 |
| 0x00007400 | VF_ARQH | VF MailBox Receive Head | Section 7.2.3.9 |
| 0x00007000 | VF_ARQT | VF MailBox Receive Tail | Section 7.2.3.10 |
| **VF - LAN Transmit and Receive Registers** | | | |
| 0x00000000 + 0x4*QTX, QTX=0...255 | QTX_TAIL[QTX] | Transmit Queue Tail | Section 7.2.4.1 |
| 0x00002000 + 0x4*QRX, QRX=0...255 | QRX_TAIL[QRX] | Receive Queue Tail | Section 7.2.4.2 |

# 7.2 Detailed Register Description - VF BAR0

## 7.2.1 VF - General Registers

This section describes the registers allocated to a VF for generic control and status. These registers are tied to the VF and are not dependent of any resource allocation.

### 7.2.1.1 VF Reset Status - VFGEN_RSTAT (0x00008800; RW)

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| VFR_STATE | 1:0 | 0x0 | The VFR state defines the VFR reset progress as follows:<br>00b = VFR in progress.<br>01b = VFR completed.<br>10b, 11b = Reserved.<br>This field is used to communicate the reset progress to the VF with no impact on hardware functionality. |
| RSVD | 31:2 | 0x0 | Reserved. |

## 7.2.2 VF - Interrupts

### 7.2.2.1 VF Interrupt Dynamic Control Zero - VFINT_DYN_CTL0 (0x00005C00; RW)

| Field | Bit(s) | Init. | Access Type | Description |
|---|---|---|---|---|
| INTENA | 0 | 0b | RW | Interrupt Enable.<br>1b = Interrupt is enabled.<br>0b = Interrupt is disabled.<br>This bit is meaningful only if the INTENA_MSK flag in this register is not set.<br>Note that INTENA and the WB_ON_ITR flags are mutually exclusive. |
| CLEARPBA | 1 | 0b | RW1C | When setting this bit, the matched PBA bit is cleared. This bit is auto-cleared by hardware. |
| SWINT_TRIG | 2 | 0b | RW1C | Trigger Software Interrupt. When the bit is set, a software interrupt is triggered. This bit is auto-cleared by hardware. |
| ITR_INDX | 4:3 | 0x0 | RW1C | This field define the ITR Index to be updated as follows. It is auto-cleared by hardware:<br>00b = ITR0.<br>01b = ITR1.<br>10b = ITR2.<br>11b = No ITR update. |
| INTERVAL | 16:5 | 0x0 | RW1C | The interval for the ITR defined by the ITR_INDX in this register. It is auto-cleared by hardware. |
| RSVD | 23:17 | 0x0 | RSV | Reserved. |
| SW_ITR_INDX_ENA | 24 | 0b | RW1C | This flag enables the programming of the SW_ITR_INDX in this register. This flag is auto cleared by hardware. |

| Field | Bit(s) | Init. | Access Type | Description |
|---|---|---|---|---|
| SW_ITR_INDX | 26:25 | 0x0 | RW | ITR Index of the Software Interrupt:<br>00b = ITR0.<br>01b = ITR1.<br>10b = ITR2.<br>11b = NoITR.<br>When programming this field, the SW_ITR_INDX_ENA flag in this register should be set as well. |
| RSVD | 29:27 | 0x0 | RSV | Reserved. |
| WB_ON_ITR | 30 | 0b | RW | When this bit is set, ITR expiration triggers write back of completed descriptors without an interrupt.<br>Note that INTENA and the WB_ON_ITR flags are mutually exclusive. |
| INTENA_MSK | 31 | 0b | RW1C | When the INTENA_MSK bit is set, the INTENA setting does not impact the device setting. This bit is auto-cleared by hardware. |

### 7.2.2.2 VF Interrupt Dynamic Control N - VFINT_DYN_CTLN[n] (0x00003800 + 0x4*n, n=0...63; RW)

| Field | Bit(s) | Init. | Access Type | Description |
|---|---|---|---|---|
| INTENA | 0 | 0b | RW | Interrupt Enable.<br>1b = Interrupt is enabled.<br>0 = Interrupt is disabled.<br>This bit is meaningful only if the INTENA_MSK flag in this register is not set.<br>Note that INTENA and the WB_ON_ITR flags are mutually exclusive. |
| CLEARPBA | 1 | 0b | RW1C | When setting this bit the matched PBA bit is cleared. This bit is auto-cleared by hardware. |
| SWINT_TRIG | 2 | 0b | RW1C | Trigger Software Interrupt. When the bit is set, a software interrupt is triggered. This bit is auto-cleared by hardware. |
| ITR_INDX | 4:3 | 0x0 | RW1C | This field defines the ITR index to be updated as follows. It is auto-cleared by hardware:<br>00b = ITR0.<br>01b = ITR1.<br>10b = ITR2.<br>11b = No ITR update. |
| INTERVAL | 16:5 | 0x0 | RW1C | The interval for the ITR defined by the ITR_INDX in this register. It is auto-cleared by hardware. |
| RSVD | 23:17 | 0x0 | RSV | Reserved. |
| SW_ITR_INDX_ENA | 24 | 0b | RW1C | This flag enables the programming of the SW_ITR_INDX in this register. This flag is auto cleared by hardware. |
| SW_ITR_INDX | 26:25 | 0x0 | RW | ITR Index of the Software Interrupt:<br>00b = ITR0.<br>01b = ITR1.<br>10b = ITR2.<br>11b = NoITR.<br>When programming this field, the SW_ITR_INDX_ENA flag in this register should be set as well. |
| RSVD | 29:27 | 0x0 | RSV | Reserved. |
| WB_ON_ITR | 30 | 0b | RW | When this bit is set, ITR expiration triggers write back of completed descriptors without an interrupt.<br>Note that INTENA and the WB_ON_ITR flags are mutually exclusive. |

| Field | Bit(s) | Init. | Access Type | Description |
|---|---|---|---|---|
| INTENA_MSK | 31 | 0b | RW1C | When the INTENA_MSK bit is set then the INTENA setting does not impact the device setting. This bit is auto-cleared by hardware. |

#### 7.2.2.3 VF Interrupt Throttling Zero - VFINT_ITR0[n] (0x00004C00 + 0x4*n, n=0...2; RW)

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| INTERVAL | 11:0 | 0x0 | ITR 'n' interval, while 'n' is the register index = 0,1,2 for the three ITRs per interrupt. It is defined in 2 µs units enabling interval range from zero to 8160 µs (0xFF0). Setting the INTERVAL to zero enable immediate interrupt. This register can be programmed also by setting the INTERVAL field in the matched xxINT_DYN_CTLx register. |
| RSVD | 31:12 | 0x0 | Reserved. |

#### 7.2.2.4 VF Interrupt Throttling N - VFINT_ITRN[n,m] (0x00002800 + 0x4*n + 0xC*m, n=0...2, m=0...63; RW)

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| INTERVAL | 11:0 | 0x0 | ITR 'n' interval, while 'n' is the register index = 0,1,2 for the three ITRs per interrupt. It is defined in 2 µs units enabling interval range from zero to 8160 µs (0xFF0). Setting the INTERVAL to zero enable immediate interrupt. This register can be programmed also by setting the INTERVAL field in the matched xxINT_DYN_CTLx register. |
| RSVD | 31:12 | 0x0 | Reserved. |

## 7.2.3 VF - Control Queues

VF exposed control queues.

### 7.2.3.1 VF MailBox Transmit Queue Base Address Low - VF_ATQBAL (0x00007C00; RW)

This register contains the lower bits of the 64-bit descriptor base address.

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| ATQBAL_LSB | 5:0 | 0x0 | Tied to zero to achieve alignment. |
| ATQBAL | 31:6 | 0x0 | Transmit Descriptor Base Address Low. Must be 64-byte aligned (together with ATQBAL_LSB). |

### 7.2.3.2 VF MailBox Transmit Queue Base Address High - VF_ATQBAH (0x00007800; RW)

This register contains the higher bits of the 64-bit descriptor base address.

| Field | Bit(s) | Init. | Access Type | Description |
|---|---|---|---|---|
| ATQBAH | 31:0 | 0x0 | RW | Transmit descriptor base address high. |

### 7.2.3.3 VF MailBox Transmit Queue Length - VF_ATQLEN (0x00006800; RW)

This register sets the size of the ring. Maximum size is 1024.

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| ATQLEN | 9:0 | 0x0 | Descriptor Ring Length. Maximum size is 1023. |
| RESERVED | 27:10 | 0x0 | Reserved. |
| ATQVFE | 28 | 0b | VF Error. Set by firmware on a PF queue when one of its VFs had an admin queue error. |
| ATQOVFL | 29 | 0b | Overflow Error. Set by firmware when a message was lost because there was no room on the queue. |
| ATQCRIT | 30 | 0b | Critical Error. This bit is set by firmware when a critical error has been detected on this queue. |
| ATQENABLE | 31 | 0b | Enable Bit. Set by the driver to indicate that the queue is active. When setting the enable bit, software should initialize all other fields. This flag is implemented by a FF and cleared by PFR. |

### 7.2.3.4 VF MailBox Transmit Head - VF_ATQH (0x00006400; RW)

Transmit queue head pointer. Note that this register might not be implemented.

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| ATQH | 9:0 | 0x0 | Transmit Queue Head Pointer. At queue initialization, software clears the head pointer and during normal operation firmware increments the head following command execution. |
| RESERVED | 31:10 | 0x0 | Reserved. |

### 7.2.3.5 VF MailBox Transmit Tail - VF_ATQT (0x00008400; RW)

Transmit queue tail pointer.

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| ATQT | 9:0 | 0x0 | Transmit Queue Tail. Incremented to indicate that there are new valid descriptors on the ring. Software might only write to this register once both transmit and receive queues are properly initialized. And clear to zero at queue initialization. |
| RESERVED | 31:10 | 0x0 | Reserved. |

### 7.2.3.6 VF MailBox Receive Queue Base Address Low - VF_ARQBAL (0x00006C00; RW)

This register contains the lower bits of the 64-bit descriptor base address.

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| ARQBAL_LSB | 5:0 | 0x0 | Tied to zero to achieve alignment. |
| ARQBAL | 31:6 | 0x0 | Transmit Descriptor Base Address Low. Must be 64-byte aligned (together with ATQBAL_LSB). |

### 7.2.3.7 VF MailBox Receive Queue Base Address High - VF_ARQBAH (0x00006000; RW)

This register contains the higher bits of the 64-bit descriptor base address.

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| ARQBAH | 31:0 | 0x0 | Transmit descriptor base address high. |

### 7.2.3.8 VF MailBox Receive Queue Length - VF_ARQLEN (0x00008000; RW)

This register sets the size of the ring. Maximum size is 1024.

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| ARQLEN | 9:0 | 0x0 | Descriptor Ring Length. Maximum size is 1023. |
| RESERVED | 27:10 | 0x0 | Reserved. |
| ARQVFE | 28 | 0b | VF Error. Set by firmware on a PF queue when one of its VFs had an admin queue error. |
| ARQOVFL | 29 | 0b | Overflow Error. Set by firmware when a message was lost because there was no room on the queue. |
| ARQCRIT | 30 | 0b | Critical Error. This bit is set by firmware when a critical error has been detected on this queue. |
| ARQENABLE | 31 | 0b | Enable Bit. Set by the driver to indicate that the queue is active. When setting the enable bit, software should initialize all other fields. This flag is implemented by a FF and cleared by PFR. |

### 7.2.3.9 VF MailBox Receive Head - VF_ARQH (0x00007400; RW)

Receive queue head pointer. Note that this register might not be implemented.

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| ARQH | 9:0 | 0x0 | Receive Queue Head Pointer. At queue initialization, software clears the head pointer and during normal operation firmware increments the head following command execution. |
| RESERVED | 31:10 | 0x0 | Reserved. |

### 7.2.3.10 VF MailBox Receive Tail - VF_ARQT (0x00007000; RW)

Transmit queue tail pointer.

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| ARQT | 9:0 | 0x0 | Transmit Queue Tail. Incremented to indicate that there are new valid descriptors on the ring. Software might only write to this register once both transmit and receive queues are properly initialized. And clear to zero at queue initialization. |
| RESERVED | 31:10 | 0x0 | Reserved. |

## 7.2.4 VF - LAN Transmit and Receive Registers

**Note:** Although the hardware interface might expose up to 256 queues, in the basic mode, only 4 queues are exposed to the driver.

### 7.2.4.1 Transmit Queue Tail - QTX_TAIL[QTX] (0x00000000 + 0x4*QTX, QTX=0...255; RW)

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| TAIL | 12:0 | 0x0 | The transmit tail defines the first descriptor that software prepares for hardware (it is the last valid descriptor plus one). The tail is a relative descriptor index to the beginning of the transmit descriptor ring. |
| RSVD | 31:13 | 0x0 | Reserved. |

### 7.2.4.2 Receive Queue Tail - QRX_TAIL[QRX] (0x00002000 + 0x4*QRX, QRX=0...255; RW)

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| TAIL | 12:0 | 0x0 | The receive tail defines the first descriptor that software handles to hardware (it is the last valid descriptor plus one). The tail is a relative descriptor index to the beginning of the receive descriptor ring. |
| RSVD | 31:13 | 0x0 | Reserved. |

# 7.3 BAR3 Registers Summary

**Table 7-2. BAR3 Registers Summary**

| Offset / Alias Offset | Abbreviation | Name | Section |
|---|---|---|---|
| 0x00000000 + 0x10*n, n=0...64 | MSIX_TADD[n] | MSI-X Message Address Low | Section 7.4.1.1 |
| 0x00000004 + 0x10*n, n=0...64 | MSIX_TUADD[n] | MSI-X Message Address High | Section 7.4.1.2 |
| 0x00000008 + 0x10*n, n=0...64 | MSIX_TMSG[n] | MSI-X Message Data | Section 7.4.1.3 |
| 0x0000000C + 0x10*n, n=0...64 | MSIX_TVCTRL[n] | MSI-X Vector Control | Section 7.4.1.4 |
| 0x00008000 + 0x4*n, n=0...2 | MSIX_PBA[n] | MSI-X PBA Structure | Section 7.4.1.5 |

# 7.4 Detailed Register Description - VF BAR3

## 7.4.1 MSI-X Table Registers

This category contains registers in the separate MSI-X BAR.

### 7.4.1.1 MSI-X Message Address Low - MSIX_TADD[n] (0x00000000 + 0x10*n, n=0...64; RW)

Message address for MSI-X table entries.

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| MSIXTADD10 | 1:0 | 0x0 | Message Address. For proper Dword alignment, software must always write zeros to these two bits; otherwise, the result is undefined. The state of these bits after reset must be 0b. These bits are permitted to be read-only or read/write. |
| MSIXTADD | 31:2 | 0x0 | Message Address. System-specified message lower address.<br>For MSI-X messages, the contents of this field from an MSI-X table entry specifies the lower portion of the Dword-aligned address (AD[31:02]) for the memory write transaction. This field is read/write. |

### 7.4.1.2 MSI-X Message Address High - MSIX_TUADD[n] (0x00000004 + 0x10*n, n=0...64; RW)

Message upper address for MSI-X table entries.

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| MSIXTUADD | 31:0 | 0x0 | Message Upper Address. System-specified message upper address bits. If this field is zero, Single Address Cycle (SAC) messages are used. If this field is non-zero, Dual Address Cycle (DAC) messages are used. This field is read/write. |

### 7.4.1.3 MSI-X Message Data - MSIX_TMSG[n] (0x00000008 + 0x10*n, n=0...64; RW)

Message Data for MSI-X Table Entries

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| MSIXTMSG | 31:0 | 0x0 | Message Data. System-specified message data.<br>For MSI-X messages, the contents of this field from an MSI-X table entry specifies the data driven on AD[31:0] during the memory write transaction’s data phase. This field is read/write. |

### 7.4.1.4 MSI-X Vector Control - MSIX_TVCTRL[n] (0x0000000C + 0x10*n, n=0...64; RW)

Vector control for MSI-X table entries.

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| MASK | 0 | 1b | Mask Bit. When this bit is set, the function is prohibited from sending a message using this MSI-X table entry. However, any other MSI-X table entries programmed with the same vector are still capable of sending an equivalent message unless they are also masked.<br>This bit’s state after reset is 1b (entry is masked). |
| RESERVED | 31:1 | 0x0 | Reserved. After reset, the state of these bits must be 0b.<br>However, for potential future use, software must preserve the value of these reserved bits when modifying the value of other vector control bits. If software modifies the value of these reserved bits, the result is undefined. |

### 7.4.1.5 MSI-X PBA Structure - MSIX_PBA[n] (0x00008000 + 0x4*n, n=0...2; RO)

Pending bits for MSI-X PBA entries.

| Field | Bit(s) | Init. | Description |
|---|---|---|---|
| PENBIT | 31:0 | 0x0 | MSI-X Pending Bits. Each bit is set to 1b when the appropriate interrupt request is set and cleared to 0b when the appropriate interrupt request is cleared. |

# 7.5 Register Subset for Modified Driver

The kernel AVF driver makes use of the registers previously described. Modified AVF drivers might be written that uses only a subset of the AVF capabilities. For example, a driver that uses only the fast path capability of the hardware and not the control mailbox or a polling mode driver. The tables that follow list which registers are needed to implement such drivers.

#### Table 7-3. BAR0 Registers Usage in Modified Drivers

| Offset / Alias Offset | Abbreviation | Section | Fast path | Polling |
|---|---|---|---|---|
| **VF - Interrupts** | | | | |
| 0x00005C00 | VFINT_DYN_CTL0 | Section 7.2.2.1 | Yes | No |
| 0x00003800 + 0x4*n, n=0...63 | VFINT_DYN_CTLN[n] | Section 7.2.2.2 | Yes | No |
| 0x00004C00 + 0x4*n, n=0...2 | VFINT_ITR0[n] | Section 7.2.2.3 | Yes | No |
| 0x00002800 + 0x4*n + 0xC*m, n=0...2, m=0...63 | VFINT_ITRN[n,m] | Section 7.2.2.4 | Yes | No |
| **VF - LAN Transmit and Receive Registers** | | | | |
| 0x00000000 + 0x4*QTX, QTX=0...255 | QTX_TAIL[QTX] | Section 7.2.4.1 | Yes | Yes |
| 0x00002000 + 0x4*QRX, QRX=0...255 | QRX_TAIL[QRX] | Section 7.2.4.2 | Yes | Yes |

#### Table 7-4. BAR3 Registers Usage in Modified Drivers

| Offset / Alias Offset | Abbreviation | Section | Fast path | Polling |
|---|---|---|---|---|
| 0x00000000 + 0x10*n, n=0...64 | MSIX_TADD[n] | Section 7.4.1.1 | Yes | No |
| 0x00000004 + 0x10*n, n=0...64 | MSIX_TUADD[n] | Section 7.4.1.2 | Yes | No |
| 0x00000008 + 0x10*n, n=0...64 | MSIX_TMSG[n] | Section 7.4.1.3 | Yes | No |
| 0x0000000C + 0x10*n, n=0...64 | MSIX_TVCTRL[n] | Section 7.4.1.4 | Yes | No |
| 0x00008000 + 0x4*n, n=0...2 | MSIX_PBA[n] | Section 7.4.1.5 | Yes | No |

**NOTE: This page has been intentionally left blank.**

**§ §**

# Appendix A Virtual Channel Protocol

```
/*******************************************************************************
 *
 * Intel Ethernet Controller XL710 Family Linux Virtual Function Driver
 * Copyright(c) 2013 - 2014 Intel Corporation.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms and conditions of the GNU General Public License,
 * version 2, as published by the Free Software Foundation.
 *
 * This program is distributed in the hope it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The full GNU General Public License is included in this distribution in
 * the file called "COPYING".
 *
 * Contact Information:
 * e1000-devel Mailing List <e1000-devel@lists.sourceforge.net>
 * Intel Corporation, 5200 N.E. Elam Young Parkway, Hillsboro, OR 97124-6497
 *
 ******************************************************************************/

#ifndef _VIRTCHNL_H_
#define _VIRTCHNL_H_

/* Description:
 * This header file describes the VF-PF communication protocol used
 * by the drivers for all devices starting from our 40G product line
 *
 * Admin queue buffer usage:
 * desc->opcode is always aqc_opc_send_msg_to_pf
 * flags, retval, datalen, and data addr are all used normally.
 * The Firmware copies the cookie fields when sending messages between the
 * PF and VF, but uses all other fields internally. Due to this limitation,
 * we must send all messages as "indirect", i.e. using an external buffer.
 *
 * All the VSI indexes are relative to the VF. Each VF can have maximum of
 * three VSIs. All the queue indexes are relative to the VSI.  Each VF can
 * have a maximum of sixteen queues for all of its VSIs.
 *
 * The PF is required to return a status code in v_retval for all messages
 * except RESET_VF, which does not require any response. The return value
 * is of status_code type, defined in the shared type.h.
 *
 * In general, VF driver initialization should roughly follow the order of
 * these opcodes. The VF driver must first validate the API version of the
 * PF driver, then request a reset, then get resources, then configure
 * queues and interrupts. After these operations are complete, the VF
 * driver may start its queues, optionally add MAC and VLAN filters, and
 * process traffic.
 */

/* START GENERIC DEFINES
 * Need to ensure the following enums and defines hold the same meaning and
 * value in current and future projects
 */

/* Error Codes */
enum virtchnl_status_code {
	VIRTCHNL_STATUS_SUCCESS= 0,
	VIRTCHNL_ERR_PARAM= -5,
	VIRTCHNL_STATUS_ERR_OPCODE_MISMATCH= -38,
	VIRTCHNL_STATUS_ERR_CQP_COMPL_ERROR= -39,
	VIRTCHNL_STATUS_ERR_INVALID_VF_ID= -40,
        VIRTCHNL_STATUS_NOT_SUPPORTED= -64,
};

#define VIRTCHNL_LINK_SPEED_100MB_SHIFT 0x1
#define VIRTCHNL_LINK_SPEED_1000MB_SHIFT 0x2
#define VIRTCHNL_LINK_SPEED_10GB_SHIFT 0x3
#define VIRTCHNL_LINK_SPEED_40GB_SHIFT 0x4
#define VIRTCHNL_LINK_SPEED_20GB_SHIFT 0x5
#define VIRTCHNL_LINK_SPEED_25GB_SHIFT 0x6

enum virtchnl_link_speed {
        VIRTCHNL_LINK_SPEED_UNKNOWN= 0,
        VIRTCHNL_LINK_SPEED_100MB= BIT(VIRTCHNL_LINK_SPEED_100MB_SHIFT),
        VIRTCHNL_LINK_SPEED_1GB= BIT(VIRTCHNL_LINK_SPEED_1000MB_SHIFT),
        VIRTCHNL_LINK_SPEED_10GB= BIT(VIRTCHNL_LINK_SPEED_10GB_SHIFT),
        VIRTCHNL_LINK_SPEED_40GB= BIT(VIRTCHNL_LINK_SPEED_40GB_SHIFT),
        VIRTCHNL_LINK_SPEED_20GB= BIT(VIRTCHNL_LINK_SPEED_20GB_SHIFT),
        VIRTCHNL_LINK_SPEED_25GB= BIT(VIRTCHNL_LINK_SPEED_25GB_SHIFT),
};

/* for hsplit_0 field of Rx HMC context */
/* deprecated with AVF 1.0 */
enum virtchnl_rx_hsplit {
        VIRTCHNL_RX_HSPLIT_NO_SPLIT      = 0,
        VIRTCHNL_RX_HSPLIT_SPLIT_L2      = 1,
        VIRTCHNL_RX_HSPLIT_SPLIT_IP      = 2,
        VIRTCHNL_RX_HSPLIT_SPLIT_TCP_UDP = 4,
        VIRTCHNL_RX_HSPLIT_SPLIT_SCTP    = 8,
};

/* END GENERIC DEFINES */

/* Opcodes for VF-PF communication. These are placed in the v_opcode field
 * of the virtchnl_msg structure.
 */
enum virtchnl_ops {
```

```
/* The PF sends status change events to VFs using
 * the VIRTCHNL_OP_EVENT opcode.
 * VFs send requests to the PF using the other ops.
 * Use of "advanced opcode" features must be negotiated as part of capabilities
 * exchange and are not considered part of base mode feature set.
 */
        VIRTCHNL_OP_UNKNOWN = 0,
        VIRTCHNL_OP_VERSION = 1, /* must ALWAYS be 1 */
        VIRTCHNL_OP_RESET_VF = 2,
        VIRTCHNL_OP_GET_VF_RESOURCES = 3,
        VIRTCHNL_OP_CONFIG_TX_QUEUE = 4,
        VIRTCHNL_OP_CONFIG_RX_QUEUE = 5,
        VIRTCHNL_OP_CONFIG_VSI_QUEUES = 6,
        VIRTCHNL_OP_CONFIG_IRQ_MAP = 7,
        VIRTCHNL_OP_ENABLE_QUEUES = 8,
        VIRTCHNL_OP_DISABLE_QUEUES = 9,
        VIRTCHNL_OP_ADD_ETH_ADDR = 10,
        VIRTCHNL_OP_DEL_ETH_ADDR = 11,
        VIRTCHNL_OP_ADD_VLAN = 12,
        VIRTCHNL_OP_DEL_VLAN = 13,
        VIRTCHNL_OP_CONFIG_PROMISCUOUS_MODE = 14,
        VIRTCHNL_OP_GET_STATS = 15,
        VIRTCHNL_OP_RSVD = 16,
        VIRTCHNL_OP_EVENT = 17, /* must ALWAYS be 17 */
        VIRTCHNL_OP_IWARP = 20, /* advanced opcode */
        VIRTCHNL_OP_CONFIG_IWARP_IRQ_MAP = 21, /* advanced opcode */
        VIRTCHNL_OP_RELEASE_IWARP_IRQ_MAP = 22, /* advanced opcode */
        VIRTCHNL_OP_CONFIG_RSS_KEY = 23,
        VIRTCHNL_OP_CONFIG_RSS_LUT = 24,
        VIRTCHNL_OP_GET_RSS_HENA_CAPS = 25,
        VIRTCHNL_OP_SET_RSS_HENA = 26,
};

/* This macro is used to generate a compilation error if a structure
 * is not exactly the correct length. It gives a divide by zero error if the
 * structure is not of the correct size, otherwise it creates an enum that is
 * never used.
 */
#define VIRTCHNL_CHECK_STRUCT_LEN(n, X) enum virtchnl_static_assert_enum_##X \
        { virtchnl_static_assert_##X = (n)/((sizeof(struct X) == (n)) ? 1 : 0) }

/* Virtual channel message descriptor. This overlays the admin queue
 * descriptor. All other data is passed in external buffers.
 */

struct virtchnl_msg {
        u8 pad[8];                      /* AQ flags/opcode/len/retval fields */
        enum virtchnl_ops v_opcode; /* avoid confusion with desc->opcode */
        enum virtchnl_status_code v_retval;  /* ditto for desc->retval */
        u32 vfid;                       /* used by PF when sending to VF */
};

VIRTCHNL_CHECK_STRUCT_LEN(20, virtchnl_msg);

/* Message descriptions and data structures.*/

/* VIRTCHNL_OP_VERSION
 * VF posts its version number to the PF. PF responds with its version number
 * in the same format, along with a return code.
 * Reply from PF has its major/minor versions also in param0 and param1.
 * If there is a major version mismatch, then the VF cannot operate.
 * If there is a minor version mismatch, then the VF can operate but should
 * add a warning to the system log.
 *
 * This enum element MUST always be specified as == 1, regardless of other
 * changes in the API. The PF must always respond to this message without
 * error regardless of version mismatch.
 */
#define VIRTCHNL_VERSION_MAJOR1
#define VIRTCHNL_VERSION_MINOR1
#define VIRTCHNL_VERSION_MINOR_NO_VF_CAPS0
```

```
struct virtchnl_version_info {
        u32 major;
        u32 minor;
};

VIRTCHNL_CHECK_STRUCT_LEN(8, virtchnl_version_info);

#define VF_IS_V10(_v) (((_v)->major == 1) && ((_v)->minor == 0))
#define VF_IS_V11(_ver) (((_ver)->major == 1) && ((_ver)->minor == 1))

/* VIRTCHNL_OP_RESET_VF
 * VF sends this request to PF with no parameters
 * PF does NOT respond! VF driver must delay then poll VFGEN_RSTAT register
 * until reset completion is indicated. The admin queue must be reinitialized
 * after this operation.
 *
 * When reset is complete, PF must ensure that all queues in all VSIs associated
 * with the VF are stopped, all queue configurations in the HMC are set to 0,
 * and all MAC and VLAN filters (except the default MAC address) on all VSIs
 * are cleared.
 */

/* VSI types that use VIRTCHNL interface for VF-PF communication. VSI_SRIOV
 * vsi_type should always be 6 for backward compatibility. Add other fields
 * as needed.
 */
enum virtchnl_vsi_type {
        VIRTCHNL_VSI_TYPE_INVALID = 0,
        VIRTCHNL_VSI_SRIOV = 6,
};

/* VIRTCHNL_OP_GET_VF_RESOURCES
 * Version 1.0 VF sends this request to PF with no parameters
 * Version 1.1 VF sends this request to PF with u32 bitmap of its capabilities
 * PF responds with an indirect message containing
 * virtchnl_vf_resource and one or more
 * virtchnl_vsi_resource structures.
 */

struct virtchnl_vsi_resource {
        u16 vsi_id;
        u16 num_queue_pairs;
        enum virtchnl_vsi_type vsi_type;
        u16 qset_handle;
        u8 default_mac_addr[ETH_ALEN];
};

VIRTCHNL_CHECK_STRUCT_LEN(16, virtchnl_vsi_resource);

/* VF offload flags
 * VIRTCHNL_VF_OFFLOAD_L2 flag is inclusive of base mode L2 offloads including
 * TX/RX Checksum offloading and TSO for non-tunneled packets.
 */
#define VIRTCHNL_VF_OFFLOAD_L2 0x00000001
#define VIRTCHNL_VF_OFFLOAD_IWARP 0x00000002
#define VIRTCHNL_VF_OFFLOAD_RSVD 0x00000004
#define VIRTCHNL_VF_OFFLOAD_RSS_AQ 0x00000008
#define VIRTCHNL_VF_OFFLOAD_RSS_REG 0x00000010
#define VIRTCHNL_VF_OFFLOAD_WB_ON_ITR 0x00000020
#define VIRTCHNL_VF_OFFLOAD_VLAN 0x00010000
#define VIRTCHNL_VF_OFFLOAD_RX_POLLING 0x00020000
#define VIRTCHNL_VF_OFFLOAD_RSS_PCTYPE_V2 0x00040000
#define VIRTCHNL_VF_OFFLOAD_RSS_PF 0x00080000
#define VIRTCHNL_VF_OFFLOAD_ENCAP 0x00100000
#define VIRTCHNL_VF_OFFLOAD_ENCAP_CSUM 0x00200000
#define VIRTCHNL_VF_OFFLOAD_RX_ENCAP_CSUM 0x00400000

#define VF_BASE_MODE_OFFLOADS (VIRTCHNL_VF_OFFLOAD_L2 | \
                VIRTCHNL_VF_OFFLOAD_VLAN | \
                VIRTCHNL_VF_OFFLOAD_RSS_PF)

struct virtchnl_vf_resource {
        u16 num_vsis;
        u16 num_queue_pairs;
        u16 max_vectors;
        u16 max_mtu;

        u32 vf_offload_flags;
        u32 rss_key_size;
        u32 rss_lut_size;

        struct virtchnl_vsi_resource vsi_res[1];
};

VIRTCHNL_CHECK_STRUCT_LEN(36, virtchnl_vf_resource);

/* VIRTCHNL_OP_CONFIG_TX_QUEUE
 * VF sends this message to set up parameters for one TX queue.
 * External data buffer contains one instance of virtchnl_txq_info.
 * PF configures requested queue and returns a status code.
 */

/* Tx queue config info */
struct virtchnl_txq_info {
        u16 vsi_id;
        u16 queue_id;
        u16 ring_len;    /* number of descriptors, multiple of 8 */
        u16 headwb_enabled; /* deprecated with AVF 1.0 */
        u64 dma_ring_addr;
        u64 dma_headwb_addr; /* deprecated with AVF 1.0 */
};

VIRTCHNL_CHECK_STRUCT_LEN(24, virtchnl_txq_info);

/* VIRTCHNL_OP_CONFIG_RX_QUEUE
 * VF sends this message to set up parameters for one RX queue.
 * External data buffer contains one instance of virtchnl_rxq_info.
 * PF configures requested queue and returns a status code.
 */

/* Rx queue config info */
struct virtchnl_rxq_info {
	u16 vsi_id;
	u16 queue_id;
	u32 ring_len;    /* number of descriptors, multiple of 32 */
	u16 hdr_size;
	u16 splithdr_enabled; /* deprecated with AVF 1.0 */
	u32 databuffer_size;
	u32 max_pkt_size;
	u32 pad1;
	u64 dma_ring_addr;
	enum virtchnl_rx_hsplit rx_split_pos; /* deprecated with AVF 1.0 */
	u32 pad2;
};

VIRTCHNL_CHECK_STRUCT_LEN(40, virtchnl_rxq_info);

/* VIRTCHNL_OP_CONFIG_VSI_QUEUES
 * VF sends this message to set parameters for all active TX and RX queues
 * associated with the specified VSI.
 * PF configures queues and returns status.
 * If the number of queues specified is greater than the number of queues
 * associated with the VSI, an error is returned and no queues are configured.
 */
struct virtchnl_queue_pair_info {
	/* NOTE: vsi_id and queue_id should be identical for both queues. */
	struct virtchnl_txq_info txq;
	struct virtchnl_rxq_info rxq;
};

VIRTCHNL_CHECK_STRUCT_LEN(64, virtchnl_queue_pair_info);

struct virtchnl_vsi_queue_config_info {
	u16 vsi_id;
```

```
        u16 num_queue_pairs;
        u32 pad;
        struct virtchnl_queue_pair_info qpair[1];
};

VIRTCHNL_CHECK_STRUCT_LEN(72, virtchnl_vsi_queue_config_info);

/* VIRTCHNL_OP_CONFIG_IRQ_MAP
 * VF uses this message to map vectors to queues.
 * The rxq_map and txq_map fields are bitmaps used to indicate which queues
 * are to be associated with the specified vector.
 * The "other" causes are always mapped to vector 0.
 * PF configures interrupt mapping and returns status.
 */
struct virtchnl_vector_map {
        u16 vsi_id;
        u16 vector_id;
        u16 rxq_map;
        u16 txq_map;
        u16 rxitr_idx;
        u16 txitr_idx;
};

VIRTCHNL_CHECK_STRUCT_LEN(12, virtchnl_vector_map);

struct virtchnl_irq_map_info {
        u16 num_vectors;
        struct virtchnl_vector_map vecmap[1];
};

VIRTCHNL_CHECK_STRUCT_LEN(14, virtchnl_irq_map_info);

/* VIRTCHNL_OP_ENABLE_QUEUES
 * VIRTCHNL_OP_DISABLE_QUEUES
 * VF sends these message to enable or disable TX/RX queue pairs.
 * The queues fields are bitmaps indicating which queues to act upon.
 * (Currently, we only support 16 queues per VF, but we make the field
 * u32 to allow for expansion.)
 * PF performs requested action and returns status.
 */
struct virtchnl_queue_select {
        u16 vsi_id;
        u16 pad;
        u32 rx_queues;
        u32 tx_queues;
};

VIRTCHNL_CHECK_STRUCT_LEN(12, virtchnl_queue_select);

/* VIRTCHNL_OP_ADD_ETH_ADDR
 * VF sends this message in order to add one or more unicast or multicast
 * address filters for the specified VSI.
 * PF adds the filters and returns status.
 */

/* VIRTCHNL_OP_DEL_ETH_ADDR
 * VF sends this message in order to remove one or more unicast or multicast
 * filters for the specified VSI.
 * PF removes the filters and returns status.
 */

struct virtchnl_ether_addr {
        u8 addr[ETH_ALEN];
        u8 pad[2];
};

VIRTCHNL_CHECK_STRUCT_LEN(8, virtchnl_ether_addr);

struct virtchnl_ether_addr_list {
        u16 vsi_id;
        u16 num_elements;
        struct virtchnl_ether_addr list[1];
};

VIRTCHNL_CHECK_STRUCT_LEN(12, virtchnl_ether_addr_list);

/* VIRTCHNL_OP_ADD_VLAN
 * VF sends this message to add one or more VLAN tag filters for receives.
 * PF adds the filters and returns status.
 * If a port VLAN is configured by the PF, this operation will return an
 * error to the VF.
 */

/* VIRTCHNL_OP_DEL_VLAN
 * VF sends this message to remove one or more VLAN tag filters for receives.
 * PF removes the filters and returns status.
 * If a port VLAN is configured by the PF, this operation will return an
 * error to the VF.
 */

struct virtchnl_vlan_filter_list {
        u16 vsi_id;
        u16 num_elements;
        u16 vlan_id[1];
};

VIRTCHNL_CHECK_STRUCT_LEN(6, virtchnl_vlan_filter_list);

/* VIRTCHNL_OP_CONFIG_PROMISCUOUS_MODE
 * VF sends VSI id and flags.
 * PF returns status code in retval.
 * Note: we assume that broadcast accept mode is always enabled.
 */
struct virtchnl_promisc_info {
        u16 vsi_id;
        u16 flags;
};
```

```
VIRTCHNL_CHECK_STRUCT_LEN(4, virtchnl_promisc_info);

#define FLAG_VF_UNICAST_PROMISC 0x00000001
#define FLAG_VF_MULTICAST_PROMISC 0x00000002

/* VIRTCHNL_OP_GET_STATS
 * VF sends this message to request stats for the selected VSI. VF uses
 * the virtchnl_queue_select struct to specify the VSI. The queue_id
 * field is ignored by the PF.
 *
 * PF replies with struct eth_stats in an external buffer.
 */

/* VIRTCHNL_OP_CONFIG_RSS_KEY
 * VIRTCHNL_OP_CONFIG_RSS_LUT
 * VF sends these messages to configure RSS. Only supported if both PF
 * and VF drivers set the VIRTCHNL_VF_OFFLOAD_RSS_PF bit during
 * configuration negotiation. If this is the case, then the RSS fields in
 * the VF resource struct are valid.
 * Both the key and LUT are initialized to 0 by the PF, meaning that
 * RSS is effectively disabled until set up by the VF.
 */
struct virtchnl_rss_key {
        u16 vsi_id;
        u16 key_len;
        u8 key[1];        /* RSS hash key, packed bytes */
};

VIRTCHNL_CHECK_STRUCT_LEN(6, virtchnl_rss_key);

struct virtchnl_rss_lut {
        u16 vsi_id;
        u16 lut_entries;
        u8 lut[1];        /* RSS lookup table*/
};
```

```
VIRTCHNL_CHECK_STRUCT_LEN(6, virtchnl_rss_lut);

/* VIRTCHNL_OP_GET_RSS_HENA_CAPS
 * VIRTCHNL_OP_SET_RSS_HENA
 * VF sends these messages to get and set the hash filter enable bits for RSS.
 * By default, the PF sets these to all possible traffic types that the
 * hardware supports. The VF can query this value if it wants to change the
 * traffic types that are hashed by the hardware.
 */
struct virtchnl_rss_hena {
        u64 hena;
};

VIRTCHNL_CHECK_STRUCT_LEN(8, virtchnl_rss_hena);

/* VIRTCHNL_OP_EVENT
 * PF sends this message to inform the VF driver of events that may affect it.
 * No direct response is expected from the VF, though it may generate other
 * messages in response to this one.
 */
enum virtchnl_event_codes {
        VIRTCHNL_EVENT_UNKNOWN = 0,
        VIRTCHNL_EVENT_LINK_CHANGE,
        VIRTCHNL_EVENT_RESET_IMPENDING,
        VIRTCHNL_EVENT_PF_DRIVER_CLOSE,
};

#define PF_EVENT_SEVERITY_INFO 0
#define PF_EVENT_SEVERITY_CERTAIN_DOOM 255

struct virtchnl_pf_event {
        enum virtchnl_event_codes event;
        union {
                struct {
                        enum virtchnl_link_speed link_speed;
                        bool link_status;
                } link_event;
        } event_data;

        int severity;
};

VIRTCHNL_CHECK_STRUCT_LEN(16, virtchnl_pf_event);

/* VIRTCHNL_OP_CONFIG_IWARP_IRQ_MAP
 * VF uses this message to request PF to map IWARP vectors to IWARP queues.
 * The request for this originates from the VF IWARP driver through
 * a client interface between VF LAN and VF IWARP driver.
 * A vector could have an AEQ and CEQ attached to it although
 * there is a single AEQ per VF IWARP instance in which case
 * most vectors will have an INVALID_IDX for aeq and valid idx for ceq.
 * There will never be a case where there will be multiple CEQs attached
 * to a single vector.
 * PF configures interrupt mapping and returns status.
 */

/* HW does not define a type value for AEQ; only for RX/TX and CEQ.
 * In order for us to keep the interface simple, SW will define a
 * unique type value for AEQ.
 */
#define QUEUE_TYPE_PE_AEQ  0x80
#define QUEUE_INVALID_IDX  0xFFFF

struct virtchnl_iwarp_qv_info {
        u32 v_idx; /* msix_vector */
        u16 ceq_idx;
        u16 aeq_idx;
        u8 itr_idx;
};

VIRTCHNL_CHECK_STRUCT_LEN(12, virtchnl_iwarp_qv_info);
```

```
struct virtchnl_iwarp_qvlist_info {
        u32 num_vectors;
        struct virtchnl_iwarp_qv_info qv_info[1];
};

VIRTCHNL_CHECK_STRUCT_LEN(16, virtchnl_iwarp_qvlist_info);

/* VF reset states - these are written into the RSTAT register:
 * VFGEN_RSTAT on the VF
 * When the PF initiates a reset, it writes 0
 * When the reset is complete, it writes 1
 * When the PF detects that the VF has recovered, it writes 2
 * VF checks this register periodically to determine if a reset has occurred,
 * then polls it to know when the reset is complete.
 * If either the PF or VF reads the register while the hardware
 * is in a reset state, it will return DEADBEEF, which, when masked
 * will result in 3.
 */
enum virtchnl_vfr_states {
        VIRTCHNL_VFR_INPROGRESS = 0,
        VIRTCHNL_VFR_COMPLETED,
        VIRTCHNL_VFR_VFACTIVE,
};

/**
 * virtchnl_vc_validate_vf_msg
 * @ver: Virtchnl version info
 * @v_opcode: Opcode for the message
 * @msg: pointer to the msg buffer
 * @msglen: msg length
 *
 * validate msg format against struct for each opcode
 */
static inline int
virtchnl_vc_validate_vf_msg(struct virtchnl_version_info *ver, u32 v_opcode,
                            u8 *msg, u16 msglen)
```

```
{
        bool err_msg_format = false;
        int valid_len = 0;

        /* Validate message length. */
        switch (v_opcode) {
        case VIRTCHNL_OP_VERSION:
                valid_len = sizeof(struct virtchnl_version_info);
                break;
        case VIRTCHNL_OP_RESET_VF:
                break;
        case VIRTCHNL_OP_GET_VF_RESOURCES:
                if (VF_IS_V11(ver))
                        valid_len = sizeof(u32);
                break;
        case VIRTCHNL_OP_CONFIG_TX_QUEUE:
                valid_len = sizeof(struct virtchnl_txq_info);
                break;
        case VIRTCHNL_OP_CONFIG_RX_QUEUE:
                valid_len = sizeof(struct virtchnl_rxq_info);
                break;
        case VIRTCHNL_OP_CONFIG_VSI_QUEUES:
                valid_len = sizeof(struct virtchnl_vsi_queue_config_info);
                if (msglen >= valid_len) {
                        struct virtchnl_vsi_queue_config_info *vqc =
                           (struct virtchnl_vsi_queue_config_info *)msg;
                        valid_len += (vqc->num_queue_pairs *
                                      sizeof(struct
                                             virtchnl_queue_pair_info));
                        if (vqc->num_queue_pairs == 0)
                                err_msg_format = true;
                }
                break;
        case VIRTCHNL_OP_CONFIG_IRQ_MAP:
                valid_len = sizeof(struct virtchnl_irq_map_info);
                if (msglen >= valid_len) {
```

```
                struct virtchnl_irq_map_info *vimi =
                   (struct virtchnl_irq_map_info *)msg;
                valid_len += (vimi->num_vectors *
                              sizeof(struct virtchnl_vector_map));
                if (vimi->num_vectors == 0)
                        err_msg_format = true;
        }
        break;
case VIRTCHNL_OP_ENABLE_QUEUES:
case VIRTCHNL_OP_DISABLE_QUEUES:
        valid_len = sizeof(struct virtchnl_queue_select);
        break;
case VIRTCHNL_OP_ADD_ETH_ADDR:
case VIRTCHNL_OP_DEL_ETH_ADDR:
        valid_len = sizeof(struct virtchnl_ether_addr_list);
        if (msglen >= valid_len) {
                struct virtchnl_ether_addr_list *veal =
                   (struct virtchnl_ether_addr_list *)msg;
                valid_len += veal->num_elements *
                   sizeof(struct virtchnl_ether_addr);
                if (veal->num_elements == 0)
                        err_msg_format = true;
        }
        break;
case VIRTCHNL_OP_ADD_VLAN:
case VIRTCHNL_OP_DEL_VLAN:
        valid_len = sizeof(struct virtchnl_vlan_filter_list);
        if (msglen >= valid_len) {
                struct virtchnl_vlan_filter_list *vfl =
                   (struct virtchnl_vlan_filter_list *)msg;
                valid_len += vfl->num_elements * sizeof(u16);
                if (vfl->num_elements == 0)
                        err_msg_format = true;
        }
        break;
case VIRTCHNL_OP_CONFIG_PROMISCUOUS_MODE:
```

```
            valid_len = sizeof(struct virtchnl_promisc_info);
            break;
    case VIRTCHNL_OP_GET_STATS:
            valid_len = sizeof(struct virtchnl_queue_select);
            break;
    case VIRTCHNL_OP_IWARP:
            /* These messages are opaque to us and will be validated in
             * the RDMA client code. We just need to check for nonzero
             * length. The firmware will enforce max length restrictions.
             */
            if (msglen)
                    valid_len = msglen;
            else
                    err_msg_format = true;
            break;
    case VIRTCHNL_OP_RELEASE_IWARP_IRQ_MAP:
            break;
    case VIRTCHNL_OP_CONFIG_IWARP_IRQ_MAP:
            valid_len = sizeof(struct virtchnl_iwarp_qvlist_info);
            if (msglen >= valid_len) {
                    struct virtchnl_iwarp_qvlist_info *qv =
                            (struct virtchnl_iwarp_qvlist_info *)msg;
                    if (qv->num_vectors == 0) {
                            err_msg_format = true;
                            break;
                    }
                    valid_len += ((qv->num_vectors - 1) *
                            sizeof(struct virtchnl_iwarp_qv_info));
            }
            break;
    case VIRTCHNL_OP_CONFIG_RSS_KEY:
            valid_len = sizeof(struct virtchnl_rss_key);
            if (msglen >= valid_len) {
                    struct virtchnl_rss_key *vrk =
                            (struct virtchnl_rss_key *)msg;
                    valid_len += vrk->key_len - 1;
```

```
                }
                break;
        case VIRTCHNL_OP_CONFIG_RSS_LUT:
                valid_len = sizeof(struct virtchnl_rss_lut);
                if (msglen >= valid_len) {
                        struct virtchnl_rss_lut *vrl =
                                (struct virtchnl_rss_lut *)msg;
                        valid_len += vrl->lut_entries - 1;
                }
                break;
        case VIRTCHNL_OP_GET_RSS_HENA_CAPS:
                break;
        case VIRTCHNL_OP_SET_RSS_HENA:
                valid_len = sizeof(struct virtchnl_rss_hena);
                break;
        /* These are always errors coming from the VF. */
        case VIRTCHNL_OP_EVENT:
        case VIRTCHNL_OP_UNKNOWN:
        default:
                return VIRTCHNL_ERR_PARAM;
        }
        /* few more checks */
        if ((valid_len != msglen) || (err_msg_format))
                return VIRTCHNL_STATUS_ERR_OPCODE_MISMATCH;

        return 0;
}
#endif /* _VIRTCHNL_H_ */
```