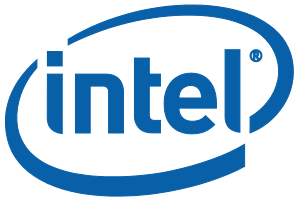


Intel® 82599 SR-IOV Driver Companion Guide

Overview of the Intel ixgbe SR-IOV Driver Implementation

LAN Access Division

323902-001
Revision 1.00
May 2010

Legal

Contents

1 Introduction

This document is designed to assist those wishing to use the SR-IOV capabilities within the Intel® 82599 10 Gigabit Ethernet Controller. The document is intended to be a companion to the *Intel® 82599 10 Gigabit Ethernet Controller Datasheet*. The datasheet documents the controller. It includes a description of the virtualization features and registers for the control of those features. See the listing at: http://developer.intel.com/products/ethernet/index.htm?iid=nc+ethernet#s1=10%20Gigabit%20Ethernet&s2=82599EB&s3=all.[1]

This document assumes knowledge of Ethernet device drivers and discusses SR-IOV portions of the Open Source Physical Function driver and the Linux* Virtual Function driver.

This document is intended to be a companion for the Intel ixgbe 2.0.75.7 and ixgbevf 1.0.8 drivers. As Intel releases updates to these drivers, there may be new features not discussed within this document. This document will be updated periodically to reflect additional capabilities within the drivers, please make sure you have the latest version of this document.

1.1 SR-IOV Overview

Current I/O Virtualization techniques have their advantages and disadvantages. None are based upon any sort of industry standard.

The industry recognizes the problems of the alternative architectures and is developing new devices that are natively shareable. These devices replicate the resources necessary for each VM to be directly connected to the I/O device so that the main data movement can occur without Hypervisor involvement.

Natively shared devices will typically provide unique memory space, work queues, interrupts, and command processing for each interface they expose, while utilizing common shared resources behind the host interface. These shared resources still need to be managed and will typically expose one set of management registers to a trusted partition in the Hypervisor.

[1] On the Developer picklist: select the device, then look for the datasheet.

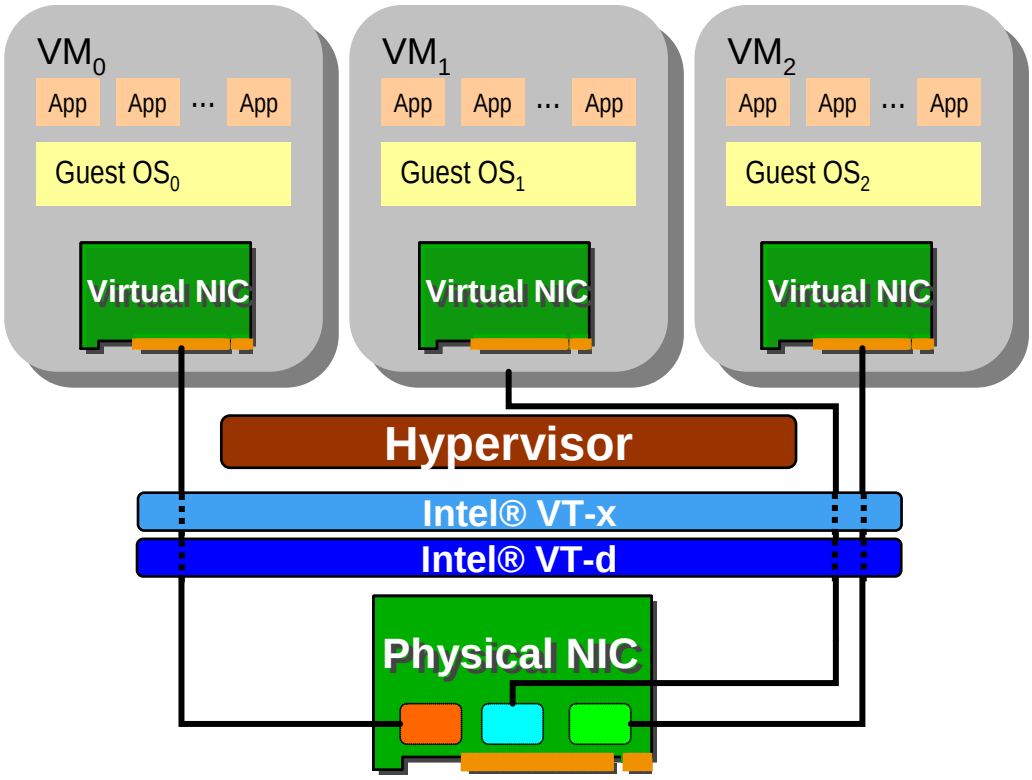


Figure 1. Natively Shared

By having separate work queues and command processing, these devices are able to simultaneously receive commands from multiple sources and merge them together in an intelligent fashion before passing them to the secondary fabric (i.e. Ethernet or SAS link). Virtualization software no longer has to multiplex the I/O requests into a serial stream and this reduces the Software overhead.

Natively shared devices can be implemented in numerous ways, both standardized and proprietary. Since most of these devices are accessed over PCI, the PCI-SIG decided to define a standard approach to creating and managing natively shared devices through the Single Root I/O Virtualization and Sharing (SR-IOV) specification.

The PCI-SIG *Single Root I/O Virtualization and Sharing* (SR-IOV) specification defines a standardized mechanism to create natively shared devices.

1.1.1 SR-IOV Goal

The goal of the PCI-SIG SR-IOV specification is to standardize on a way of sharing an I/O device in a virtualized environment. This is accomplished by bypassing the Hypervisor's involvement in data movement by providing independent Memory Space, Interrupts, and DMA streams for each virtual machine. The SR-IOV architecture is designed to allow a device to support multiple virtual functions and much attention was placed on minimizing the hardware cost of each additional function.

The SR-IOV specification introduces two new function types:

- Physical Functions (PF): This is a full PCIe function that includes the SR-IOV Extended Capability (used to configure and manage the SR-IOV functionality).
- Virtual Function (VF): This is a light weight PCIe function that contains the resources necessary for data movement (but minimizes the set of configuration resources).

1.1.2 High Level Overview of PCI-SIG SR-IOV

The Direct Assignment method of virtualization provides very fast I/O. However, it prevents the sharing of the I/O device. The SR-IOV specification provides a mechanism by which a Single Root Function (for example, a single Ethernet Port) can appear to be multiple separate physical devices.

A SR-IOV capable device can be configured (usually by the Hypervisor) to appear in the PCI Configuration space as multiple functions; each with its own configuration space, complete with Base Address Registers(BARs).

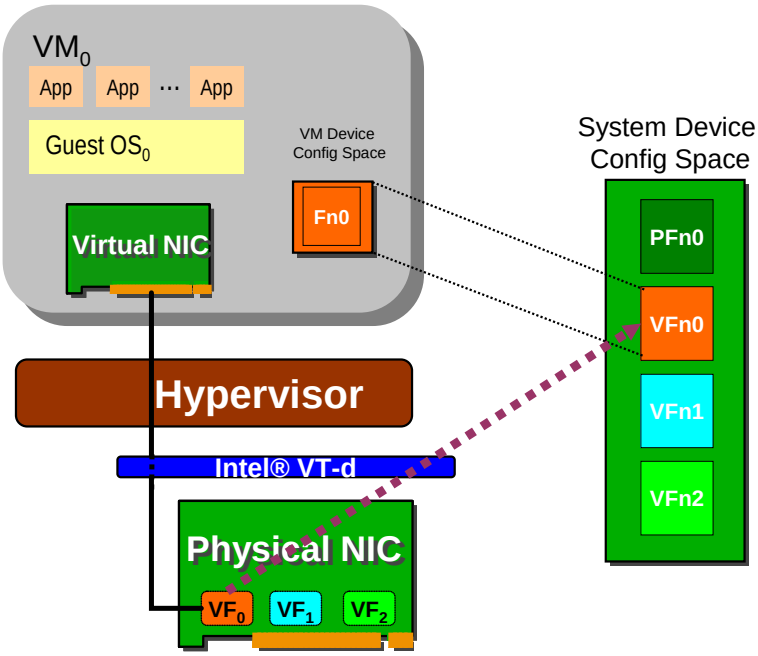


Figure 2. Mapping Virtual Function Configuration

The SR-IOV capable device provides a configurable number of independent Virtual Functions, each with its own PCI Configuration space. The Hypervisor assigns one or more Virtual Functions to a virtual machine. Memory Translation technologies such as those in Intel® VT-d provide hardware assisted techniques to allow direct DMA transfers.

The SR-IOV specification details how the PCI Configuration information is to appear; this information is different from standard PCI devices – the Hypervisor must know how to access and read this information in order to provide access to them from a VM.

Refer to the PCI SIG SR-IOV specification for details.

2 High Level Architecture

This section gives a high-level overview of how major pieces work together to provide SR-IOV functionality for Ethernet Connectivity with the 82599.

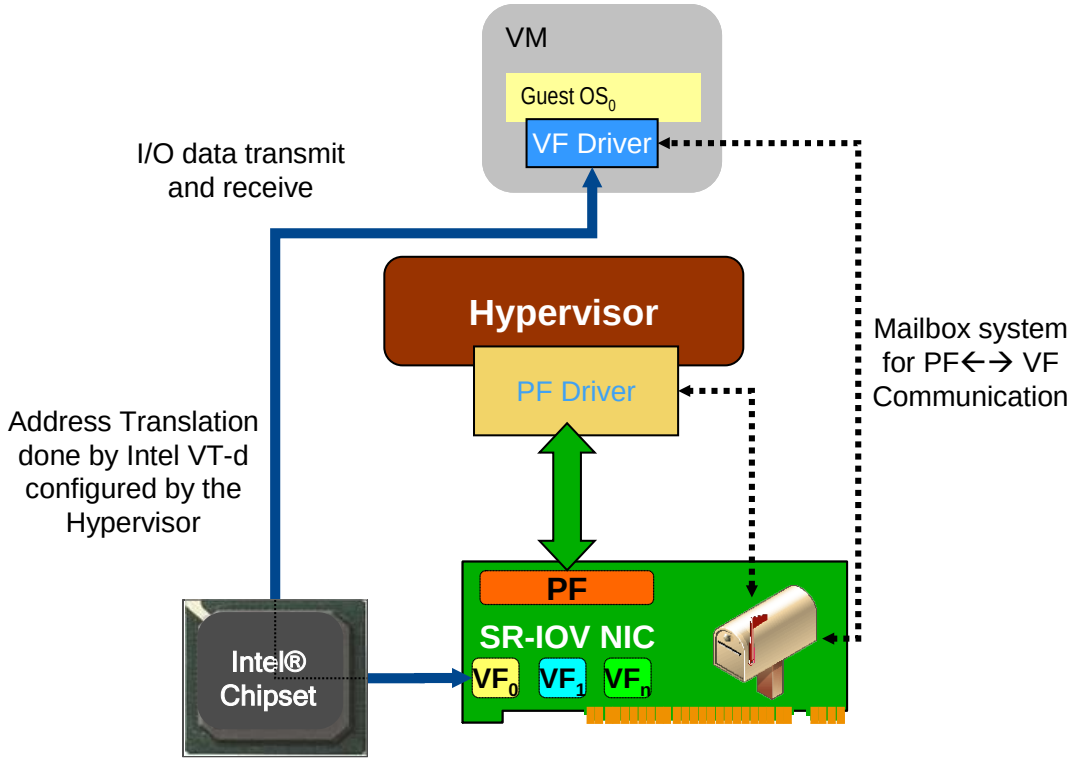


Figure 3. Intel SR-IOV Overview

2.1 Hypervisor

The Hypervisor (in this case Open Source Xen or KVM) must perform a number of activities in order to facilitate a SR-IOV capable environment.

2.1.1 PCI Configuration Access for the VM

As with all virtualized environments in a SR-IOV system, the Hypervisor must present PCI Configuration space to the Virtual Machine for the I/O device. In a shared I/O model, the Hypervisor usually presents a very simple Ethernet Controller to the VM. With SR-IOV, the Hypervisor presents the actual configuration space to a specific VF; granting the Virtual Function driver within the VM physical access to the VF resources.

2.1.2 PCI SIG SR-IOV PCI Configuration Area

Part of the SR-IOV specification involves the definition of how a SR-IOV capable device must advertise its SR-IOV capabilities and how the Virtual Function PCI Configuration information is to be stored and accessed. This is a new mechanism specific to SR-IOV; the Hypervisor must know how to read and parse this information. Linux and Xen Kernels have been updated to be able utilize this new capability.

2.1.3 VT-d Setup/Configuration

VT-d enables direct assignment of an I/O device to a VM by remapping the DMA operations. The Hypervisor must configure VT-d with the I/O addresses to be remapped.

2.2 Physical Function Driver

The Physical Function (PF) Driver is a specialized driver that manages global functions for SR-IOV devices and is responsible for configuring shared resources. The driver is specific to the Hypervisor and is expected to operate in a more privileged environment than a typical virtual machine driver. The PF Driver contains all traditional driver functionality to provide access to the I/O resource for the Hypervisor; it can also be called upon to perform operations that impact the entire device. The driver must be in a persistent environment, loaded before any Virtual Function Driver and unloaded (if necessary) after all Virtual Function drivers.

The Intel SR-IOV PF Driver contains all of the standard capabilities of any Intel® Ethernet Controller driver, such as VLAN filtering etc.

It has additional functionality specifically for SR-IOV utilization, such as:

- MAC Address Generation for VFs
- Communication with VF Drivers via Mailbox system for:
  - Configuration of VLAN Filter by VF Driver
  - Configuration of Multicast Address by VF Driver
  - Configuration of Long Packet Maximum Length by VF Driver
  - VF Driver requests a reset of its resources

2.3 Virtual Function Driver

Standard device drivers (drivers that are not aware of the virtualized environment) have specific expectations regarding how it can control the device and how the device will behave. With virtualization, a standard driver typically communicates with an intermediary software layer that abstracts and emulates the underlying physical device. In most cases, the driver is not aware that this indirection exists.

With directly assigned, natively shared devices; expectation and device behavior changes. The Virtual Function interface does not include the full spectrum of PCIe controls and typically does not have direct control of shared components and resources, such as setting the Ethernet link rate. In this environment, it is beneficial to enlighten or para-virtualize the Virtual Function Driver (VF Driver) so that it is aware of the virtualization environment.

These para-virtualized drivers communicate directly with the hardware to perform data movement operations, they also realize the device is a shared resource and rely on the services of the PF Driver to handle operations that can have global impact, like initialization and control of the secondary fabric.

The Intel Virtual Function driver is loaded the same way any other Intel Ethernet Device driver is loaded – based upon the device ID. The PCI Configuration information for the Intel Virtual Functions have a device ID identifying it as a virtual function, as such the appropriate driver is loaded.

2.4 PF Driver-VF Driver Communication

One necessary component for device sharing is the ability for the VF Driver to communicate with the PF Driver in order to request operations that have global effect. This communication channel needs this ability to pass messages and generate interrupts.

The SR-IOV specification does not define a mechanism for this communication path. Intel has chosen to implement this communication channel utilizing a set of hardware mailboxes and doorbells within the Intel Ethernet Controller for each Virtual Function.

The primary reason Intel chose to provide a hardware mailbox system was to ensure a communication mechanism that is Hypervisor independent and persistently available. However, a software based channel could also be used for this communication link if provided by the hypervisor subsystem. These channels will be vendor specific and require custom enabling to support. The hardware channel will always be available for backup communications unless specifically disabled by the Hypervisor. Some Hypervisors may wish to disable this capability for security concern reasons. At this time, not all Hypervisor vendors have committed to providing a software based messaging mechanism. Therefore the Intel reference drivers provide an API that utilizes the hardware channel.

3 Intel® 82599 10 Gigabit Ethernet Controller SR-IOV Support

This section will provide an overview of how 82599 handles such tasks as sorting packets for specific Virtual Functions, how resources are allocated for each VF, and what resources the PF receives.

3.1 Queues

The 82599 has 128 Transmit and 128 Receive queues. They are generally referred to/thought of as queue pairs (1 Transmit and 1 Receive queue). This gives the 82599 128 queue pairs.

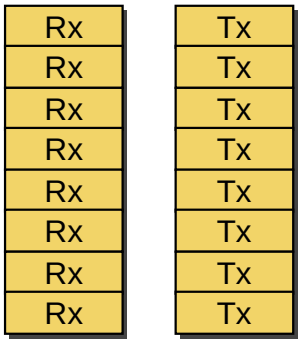


Figure 4. Queue Pair

3.2 Pools

A pool is a group of queue pairs assigned to the same VF, used for transmit and receive.

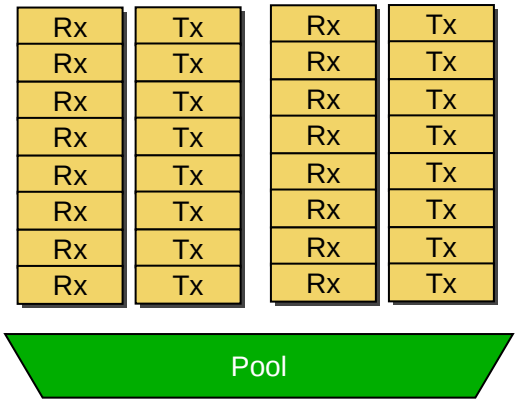


Figure 5 A 82599 Pool

The 82599 has up to 64 pools, with each pool having two queue pairs in it. That is two Transmit and Receive queue assigned to each VF. For simplificatio, in the remainder of this document, pools are represented like the representation in Figure 6.

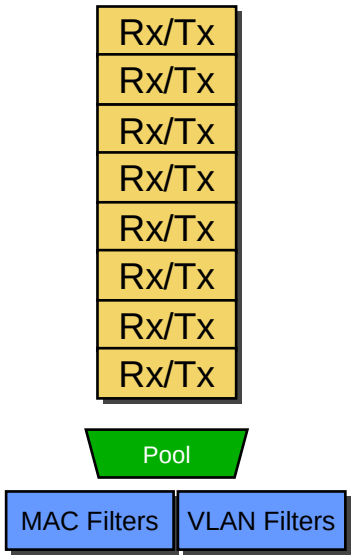


Figure 6. Pool Filters

The 82599 allows for a variable configuration for the number of pools. In SR-IOV mode, there can be 16, 32 or 64 pools. The number of queue pairs associated with a pool differs depending upon the configuration.

Table 1. Queue/Pool Assignment

Number of Pools	Queue Pairs per Pool	Total Queues per Pool
16	8	16
32	4	8
64	2	4

Refer to Section 6.2.1 for further information regarding the configuration of the number of pools.

Data is placed into the pool based upon L2 filters – MAC Address and VLAN Tag filters. Each pool can be configured for multiple MAC address and 64 VLAN Tags shared amongst all pools (both MAC addresses and VLAN tags).

3.3 Layer 2 Classifier/Sorter

The 82599 has within it what can be thought of as a simple Layer-2 (L2) switch that is responsible for sorting packets based upon the destination MAC address or VLAN Tag.

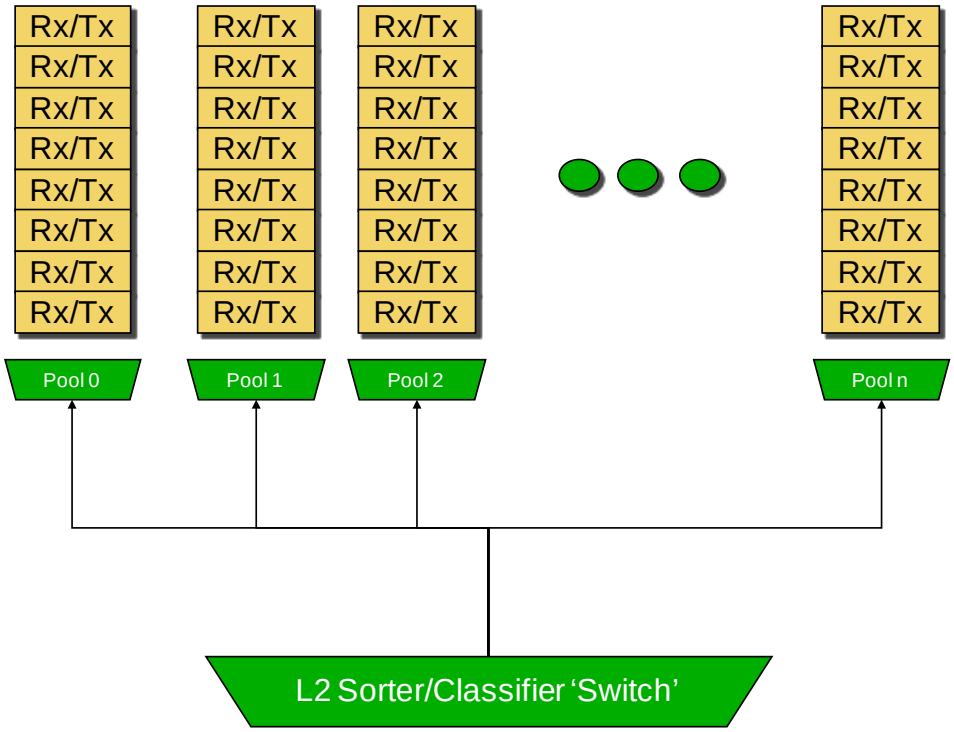


Figure 7. Layer 2 Sorter/Classifier/Switch within the Intel® 82599

When a packet comes into the Ethernet Controller, it is examined to see if it matches one of the possible L2 filters, such as MAC and VLAN Tag filters. It is then forwarded to the appropriate pool based upon a filter match.

3.3.1 Virtual Function and Resources

Each Virtual Function in 82599 has dedicated resources allocated to it, including a Pool which is composed of Queue's as discussed above. Each VF also contains its own set of registers used by the VF driver to configure actions for transmit and receive, mailbox messages (refer to Section 4) etc.

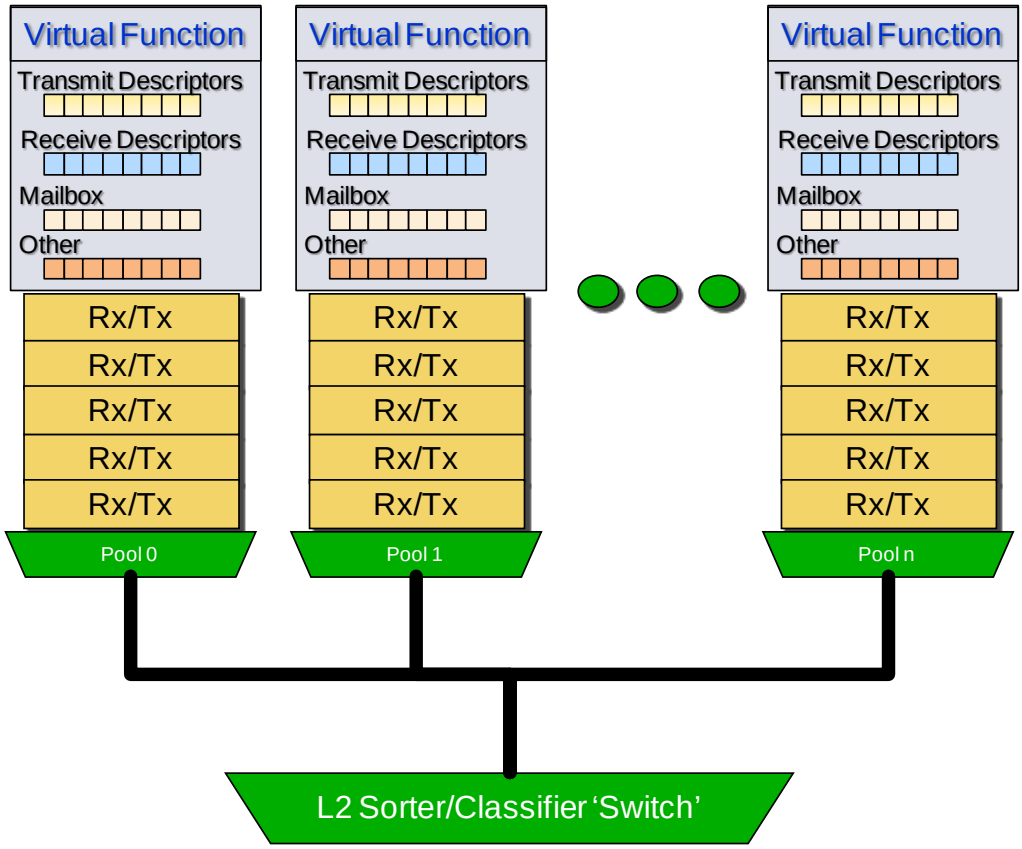


Figure 8 Virtual Functions

3.3.2 Additional Capabilities

3.3.2.1 Pool to Pool (VF to VF) Bridging

In a classic virtualized environment where a software based virtual switch exists in the Hypervisor, packets being transmitted from a VM are examined to see if the destination MAC address matches that of another VM running on the system.

If there is a match, the Hypervisor does not physically transmit the packet using the Ethernet Controller, but rather it sends it to the destination VM via the software switch.

One of the goals of SR-IOV is to bypass the Hypervisor from the steps required to transmit and receive I/O. If the Hypervisor were to be required to examine all packets being transmitted from a VF to see if it was destined for another VF, this goal of no Hypervisor involvement in data movement would not be achieved.

The 'switch' within the 82599 is capable of performing this bridging from one VF to another VF itself, in hardware, without the involvement of the Hypervisor.

3.3.2.2 Anti-spoofing

There are two types of anti-spoofing mechanisms supported by the 82599.

3.3.2.2.1 MAC Address

The source MAC address of each outgoing packet can be compared to the MAC address the sending VM uses for packets reception. A packet with a non matching Source MAC address is dropped - thus preventing spoofing of the MAC address.

This feature is can be enabled on a per VF basis from the PF Driver (refer to Section 7.2).

3.3.2.2.2 VLAN Tag

If VLAN anti spoofing is enabled, a check is done to validate that sender VF is a member of the VLAN set in the packet.

This feature is can be enabled on a per VF basis from the PF Driver (refer to Section 7.3).

3.4 Example Receive Flow

This section provides a high level flow overview of how a packet may be received and sent to a VM.

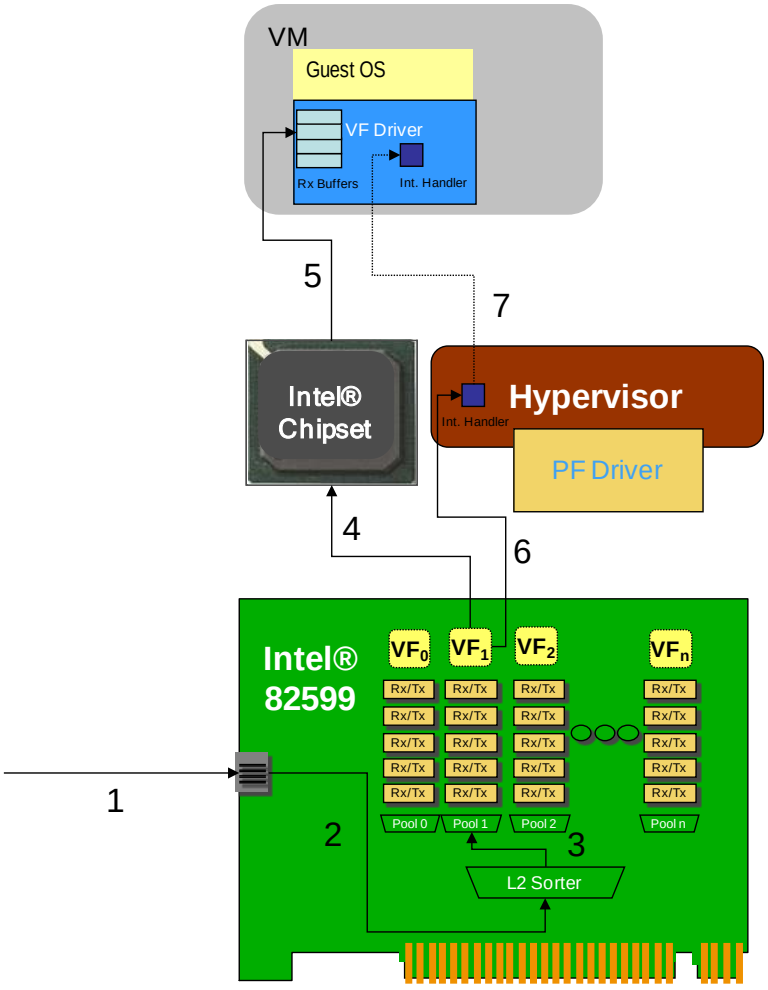


Figure 9. Packet Sent to a VM

Step 1 & 2: Packet Arrives, sent to the L2 Sorter/Switch.

Step 3: Packet is sorted based upon destination MAC address; in this case, it matches Pool/VF 1.

Step 4: NIC initiates DMA action to move packet to VM

Step 5: DMA action hits the Intel Chipset, where VT-d (configured by the Hypervisor) performs the required Address Translation, for the DMA operation; resulting in the packet being DMA'd into the VM's VF Driver buffers.

Step 6: NIC posts MSI-X interrupt indicating a Rx transaction has been completed. This interrupt is received by the Hypervisor.

Step 7: The Hypervisor injects a virtual interrupt to the VM indicating a Rx transaction has been completed, the VM's VF Driver then processes the packet.

4 Mailbox Communication System

There are times when the VF Driver must communicate with the PF Driver in order to accomplish something not provided within the PCI resources exposed via the Virtual Function. An example is when the VF Driver wants to define a VLAN filter. This capability is not exposed in the Virtual Function in any way, as such the VF Driver cannot directly configure a VLAN filter.

The VF Driver can however make a request to the PF Driver to do this type of configuration on behalf of the VF.

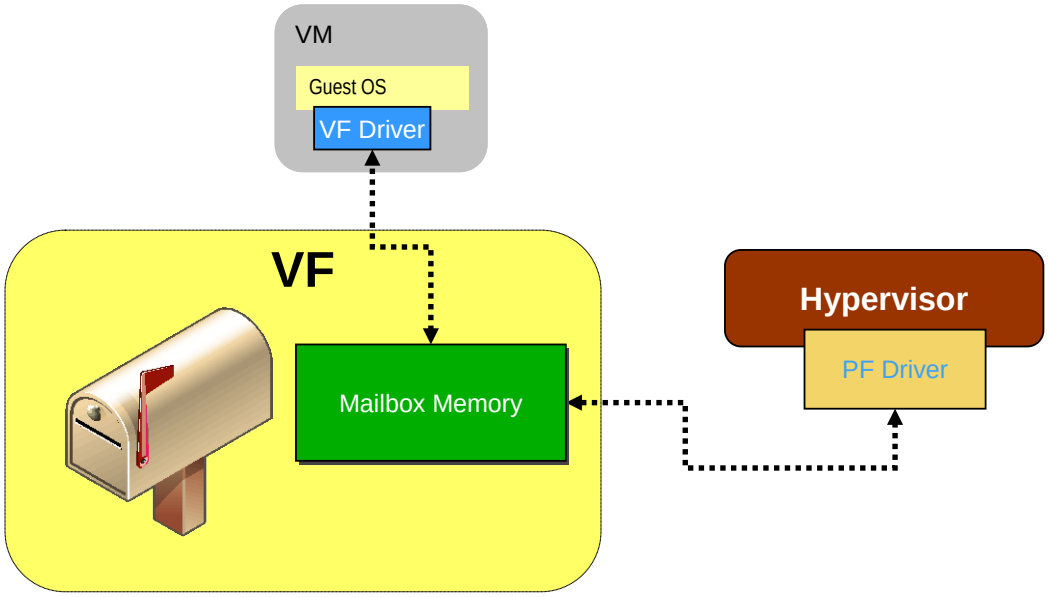


Figure 10. Mailbox Communication

4.1 Virtual Function Mailbox

When a VM is given access to a VF, part of the VF resources exposed is the Mailbox functionality. It is a fairly simple capability, composed of a buffer into which messages are written to or read from and a register (VFMailbox) providing a synchronization/semaphore between the PF and the VF when using the Mailbox.

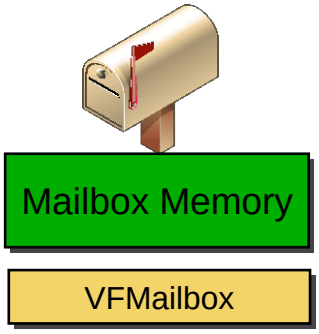


Figure 11. Mailbox from VF Perspective

The buffer is accessible by both the VF and the PF, in this way messages can be passed back and forth. Since the mailbox buffer and VFMailbox register are part of the VF resources, they are exclusive to a given VF. This means that a VF Driver for a given VF cannot write to any mailbox resources other than the one associated with the VF assigned to the VM (where the VF Driver is running).

More information about the mailbox buffer (Virtualization Mailbox Memory [VMBMEM]) and the VFMailbox register is in the *Intel® 82599 10 Gigabit Ethernet Controller Datasheet*.

4.1.1 Example Code

Within the Intel ixgbevf sample driver code, see the ixgbe_mbx.h and ixgbe_mbx.c files. These create a set of APIs used by the ixgbevf Driver for mailbox communication.

4.2 Physical Function Mailbox

The Physical Function Driver has access to all of the VF mailboxes via the VFMBMEM (VF Mailbox Mailbox Memory) and PFMailbox (Physical Function Mailbox) register arrays.

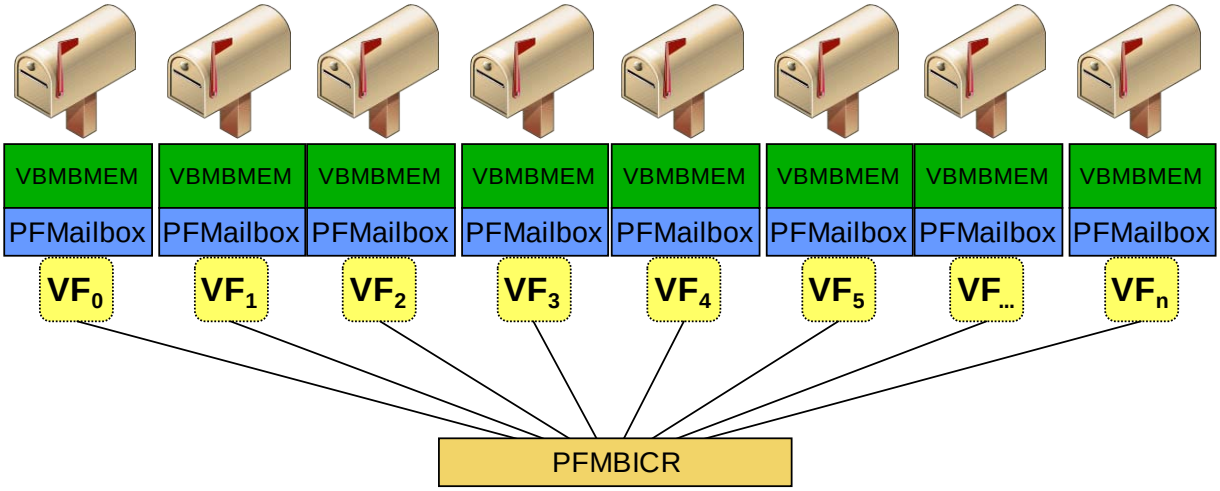


Figure 12. Mailbox from PF Perspective

When a VF Driver writes a message to the VFMBMEM buffer and sets the appropriate bits within its VFMailbox register, an interrupt is generated for the PF. The PF Driver then fetches the message and may, depending upon the message, acknowledge it.

At this time, the PF Driver sends one asynchronous message to a VF using the Mailbox; this is the IXGBE_PF_CONTROL_MSG message detailed in Section 5.4.1.1.

More information on the VFMBMEM and PFMailbox registers is in the *Intel® 82599 10 Gigabit Ethernet Controller Datasheet.*

5 Virtual Function Driver

The sample Intel VF Driver is a modified version of standard Intel ixgbe 10 Gigabit Ethernet driver. It is loaded based upon the Device ID presented to the VM as part of the PCI Configuration information from the Hypervisor. The Intel Virtual Functions have a Device ID that identifies them as Intel Virtual Functions, so that the appropriate driver can be loaded.

The Intel VF Driver can be split in three parts:

- The OS interface – that API's exposed to the VM Operating System
- I/O Operations – utilizing the SR-IOV capabilities for I/O without Hypervisor intervention
- Configuration Tasks – actions such as configure VLAN filtering that require communication with the PF Driver.

5.1 The Operating System Interface

As far as the OS in which the Intel VF Driver is running is concerned, the VF Driver is a standard driver, with all of the expected functionality. There is nothing specific to SR-IOV required.

Standard tools and API's work just fine, including Ethtool and Netdev- though the output from these tools is differ from standard drivers.

Examples of output from Ethtool running in a VM are below:

ethtool – i

```
driver: ixgbevf
version: 1.04-NAPI
firmware-version: N/A
bus-info: 0000:00:05.0
```

ethtool – t

```
The test result is PASS
The test extra info:
Register test  (offline)    0
Link test  (on/offline) 0
```

ethtool – k

```
Offload parameters for eth1:
rx-checksumming: on
tx-checksumming: on
scatter-gather: on
tcp segmentation offload: on
udp fragmentation offload: off
generic segmentation offload: off
generic-receive-offload: off
```

5.2 I/O Operations and Activities

The fundamental reason for SR-IOV is to enable a driver within a VM for direct access to PCI for I/O operations and to share a device among VMs. The Intel VF Driver understands that it is running in a virtualized environment and has limited PCI resources available.

It has standard I/O operations for an Intel Ethernet Controller. The only difference is the limited PCI resources available. The resources available include the required ability to Transmit and Receive Ethernet packets. Additional capabilities provided by Intel® Virtual Function hardware include:

- Status Information, including:
  - Link Speed
  - Link Status
  - Duplex Mode
- Statistics include:
  - Good Packets Received Count
  - Good Packets Transmitted Count
  - Good Octets Received Count
  - Good Octets Transmitted Count
  - Multicast Packets Received Count
  - Function Level Reset (FLR)
  - VLAN Tag Insertion
  - Checksum Insertion

For more information on VF hardware, refer to the Device Registers – VF in the Intel® 82599 10 Gigabit Ethernet Controller Datasheet.

5.3 Actions taken via Mailbox system – VF to PF

The PCI resources exposed by the VF do not have provisions for all required activities the VF Driver may need to do, such as configuring a VLAN tag or Multicast address.

In such cases, the VF Driver utilizes the Mailbox system to pass a message to the PF Driver so that it in turn will perform the desired operation.

Currently defined actions performed utilizing this Mailbox mechanism includes:

- Resetting the VF
- Configuring the VF MAC Address
- Setting Multicast Address
- Setting VLAN Filter
- Setting Long Packet Maximum Length
- The mailbox message IDs are defined within the ixgbe_mbx.h file for both the ixgbe PF

and ixgbevf VF Drivers.

5.3.1 Resetting the Virtual Function

```
Message ID: IXGBE_VF_RESET
```

The VF Driver sends this message to the PF Driver after it (the VF Driver) has performed a Function Level Reset (FLR) on the VF resources.

An example of this can be found in ixgbevf source file ixgbe_vf.c, in the ixgbe_reset_hw_vf() function:

File: `ixgbe_vf.c`

Function: `ixgbe_reset_hw_vf`

```
msgbuf[0] = IXGBE_VF_RESET;
mbx->ops.write_posted(hw, msgbuf, 1, 0);
```

When the PF Driver receives this message, part of its acknowledgement is to send back the MAC address for the VF.

See the PF Driver Handler (Section 6.3.2.1) for more information.

5.3.2 Configuring a MAC Address

```
Message ID: IXGBE_VF_SET_MAC_ADDR
```

The VF Driver sends this message when it wishes to define its own MAC address (rather than use the default one defined by the PF during initialization of the VF).

An example of this can be found in ixgbevf source file ixgbe_vf.c, in the ixgbe_set_rar_vf() function:

File: `ixgbe_vf.c`

Function: `ixgbe_set_rar_vf`

```
msgbuf[0] = IXGBE_VF_SET_MAC_ADDR;
memcpy(msg_addr, addr, 6);
ret_val = mbx->ops.write_posted(hw, msgbuf, 3);
```

See the PF Driver Handler (Section 6.3.2.2) for more information.

5.3.3 Setting Multicast Address

```
Message ID: IXGBE_VF_SET_MULTICAST
```

The VF Driver sends this message when it wishes to add a Multicast Address for filtering incoming packets on.

An example of this can be found in ixgbevf source file ixgbe_vf.c, in the ixgbe_update_mc_addr_list_vf() function:

File: `ixgbe_vf.c`

Function: `ixgbe_update_mc_addr_list_vf`

```
cnt = (mc_addr_count > 30) ? 30 : mc_addr_count;
```

```
msgbuf[0] = IXGBE_VF_SET_MULTICAST;
msgbuf[0] |= cnt << IXGBE_VT_MSGINFO_SHIFT;

for (i = 0; i < cnt; i++) {
   vector = ixgbe_mta_vector(hw, next(hw, &mc_addr_list, &vmdq));
   hw_dbg(hw, "Hash value = 0x%03X\n", vector);
   vector_list[i] = (u16)vector;
}

mbx->ops.write_posted(hw, msgbuf, IXGBE_VFMAILBOX_SIZE, 0);
```

See the PF Driver Handler (Section 6.3.2.3) for more information.

5.3.4 Setting VLAN Filter

```
Message ID: IXGBE_VF_SET_VLAN
```

The VF Driver sends this message when it wishes to set a VLAN ID Tag for filtering incoming packets on.

An example of this can be found in ixgbevf source file ixgbe_vf.c, in the ixgbe_set_vfta_vf() function:

File: `ixgbe_vf.c`

Function: `ixgbe_set_vfta_vf`

```
msgbuf[0] = IXGBE_VF_SET_VLAN;
msgbuf[1] = vlan;

/* Setting the 8 bit field MSG INFO to TRUE indicates "add" */
msgbuf[0] |= vlan_on << IXGBE_VT_MSGINFO_SHIFT;

mbx->ops.write_posted(hw, msgbuf, 2, 0);
```

See the PF Driver Handler (Section 6.3.2.4) for more information. Note that when the PF Driver configures the appropriate VLAN filtering for the VF, it will also, by default, configure VLAN Tag stripping.

The VF Driver configures the insertion of the VLAN Tag on transmit when it creates the packet for transmit.

5.4 PF to VF Mailbox Messages

Most of the communication between the VF and PF Drivers is the VF Driver sending messages to the PF. The PF Driver is also capable of sending messages to the VF Driver. At this time there is only a single message that is passed in this direction.

5.4.1.1 IXGBE_PF_CONTROL_MSG

```
Message Handler Function: ixgbevf_msix_mbx() : ixgbevf_main.c
```

When the VF Driver receives this message from the PF Driver, it reschedules the watchdog task so that it will execute in on the next jiffie. The watchdog task will detect that the PF has reset and will set things up for the VF so that it will go into a wait loop until the PF comes up again.

File: `ixgbevf_main.c`

Function: `ixgbevf_msix_mbx`

```
hw->mbx.ops.read(hw, &msg, 1, 0);

if ((msg & IXGBE_MBVFICR_VFREQ_MASK) == IXGBE_PF_CONTROL_MSG)
   mod_timer(&adapter->watchdog_timer,
         round_jiffies(jiffies + 1));
```

See Section 6.3.1.1 for details on how the PF Driver sends this message.

6 Physical Function Driver

The Physical Function driver written for the 82599 for the Open Source Hypervisors is an extension of the legacy/standard Intel® ixgbe driver. This driver is responsible for the physical function resources as well as handling some of the configuration of VF specific settings.

6.1 Initialization

When the driver is first probed (the __devinit ixgbe_probe function found in ixgbe_main.c), among the many tasks the driver does during initialization is to make a call to register itself as an SR-IOV device:

File: `ixgbe_main.c`

Function: `__devinit ixgbe_probe_vf`

```
err = pci_enable_sriov(adapter->pdev, adapter->num_vfs);
```

This call registers the 82599 as a SR-IOV device, indicating support for the specified number of VFs. There are sufficient resources for 64 VF's, however if you were to allocate all resources for VFs, the PF could not send and receive any Ethernet packets, as there would be no resources available.

Recall that support for SR-IOV must be available from the Hypervisor itself. The Linux kernel has been enhanced to support such requirements. The pci_enable_sriov() function is one of the enhancements that has been added to the kernel.

6.2 Default Configuration

There are a number of defaults configured during the initialization phase of the driver. These defaults include the number of VF's to be used, the VF Offload configuration, and the VF MAC address assignments.

Intel investigated adding the ability to manipulate many of these settings by extending the capability of standard kernel utilities and getting this work approved by the open source community. This mechanism has not yet been accepted by the community; as a result, it is not available at this time.

At this time, the only default that can be overridden is the number of VFs to use. This can be overridden using a command-line switch when the PF Driver is loaded. The command-line parameter is:

max_vfs=n

Where n can be a number up to 63.

NOTE: The 82599 has sufficient resources for up to 64 pools and Virtual Functions. However, only 63 can be created and assigned to Virtual Functions. The Physical Function also requires resources in order to operate.

Details regarding many of the defaults for the PF Driver follow below.

6.2.1 Assignment of Queue's to Pools

The 82599 PF Driver configures, by default, enough pools for up to 63 VFs. Each pool has 2 queue pairs associated with it, for a total of 4 queues. See Section 3.2 for information regarding pools and queues.

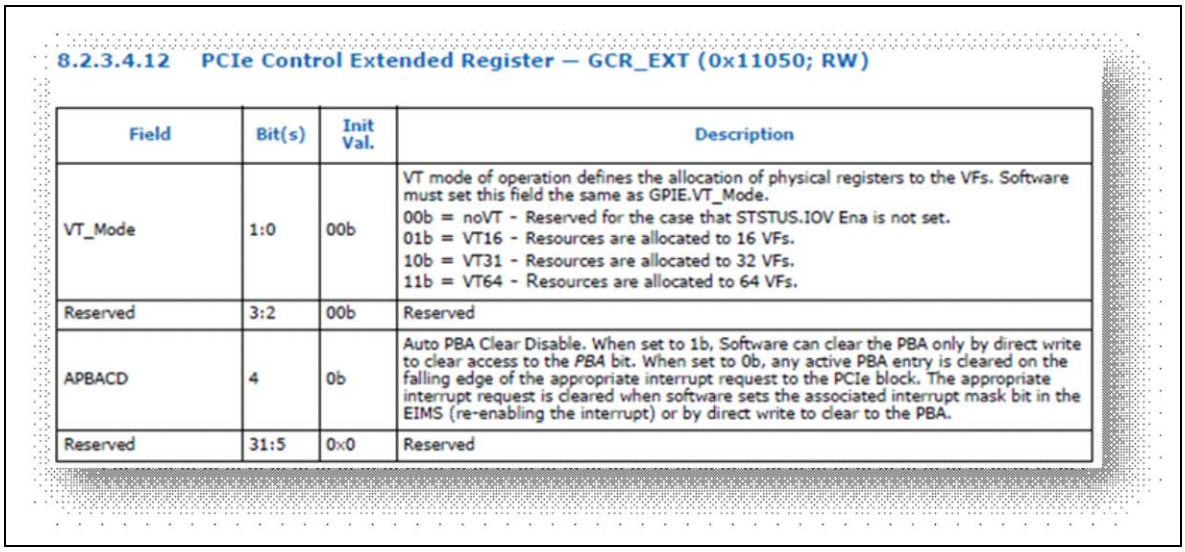
File: `ixgbe_main.c`

Function: `ixgbe_up_complete`

```
        if (adapter->flags & IXGBE_FLAG_SRIOV_ENABLED) {
            gpie &= ~IXGBE_GPIE_VTMODE_MASK;
            gpie |= IXGBE_GPIE_VTMODE_64;
        }

        IXGBE_WRITE_REG(hw, IXGBE_GPIE, gpie);
```

This piece of code configures the PCIe Control Extended Register (GCR_EXT- 0x11050) to enable 64 VF's by default.



8.2.3.4.12 PCIe Control Extended Register — GCR_EXT (0x11050; RW)

Field	Bit(s)	Init Val.	Description
VT_Mode	1:0	00b	VT mode of operation defines the allocation of physical registers to the VFs. Software must set this field the same as GPIE.VT_Mode. 00b = noVT - Reserved for the case that STSTUS.IOV Ena is not set. 01b = VT16 - Resources are allocated to 16 VFs. 10b = VT31 - Resources are allocated to 32 VFs. 11b = VT64 - Resources are allocated to 64 VFs.
Reserved	3:2	00b	Reserved
APBACD	4	0b	Auto PBA Clear Disable. When set to 1b, Software can clear the PBA only by direct write to clear access to the *PBA* bit. When set to 0b, any active PBA entry is cleared on the falling edge of the appropriate interrupt request to the PCIe block. The appropriate interrupt request is cleared when software sets the associated interrupt mask bit in the EIMS (re-enabling the interrupt) or by direct write to clear to the PBA.
Reserved	31:5	0x0	Reserved

Figure 13. PCIE Control Extended Register definition from the Datasheet

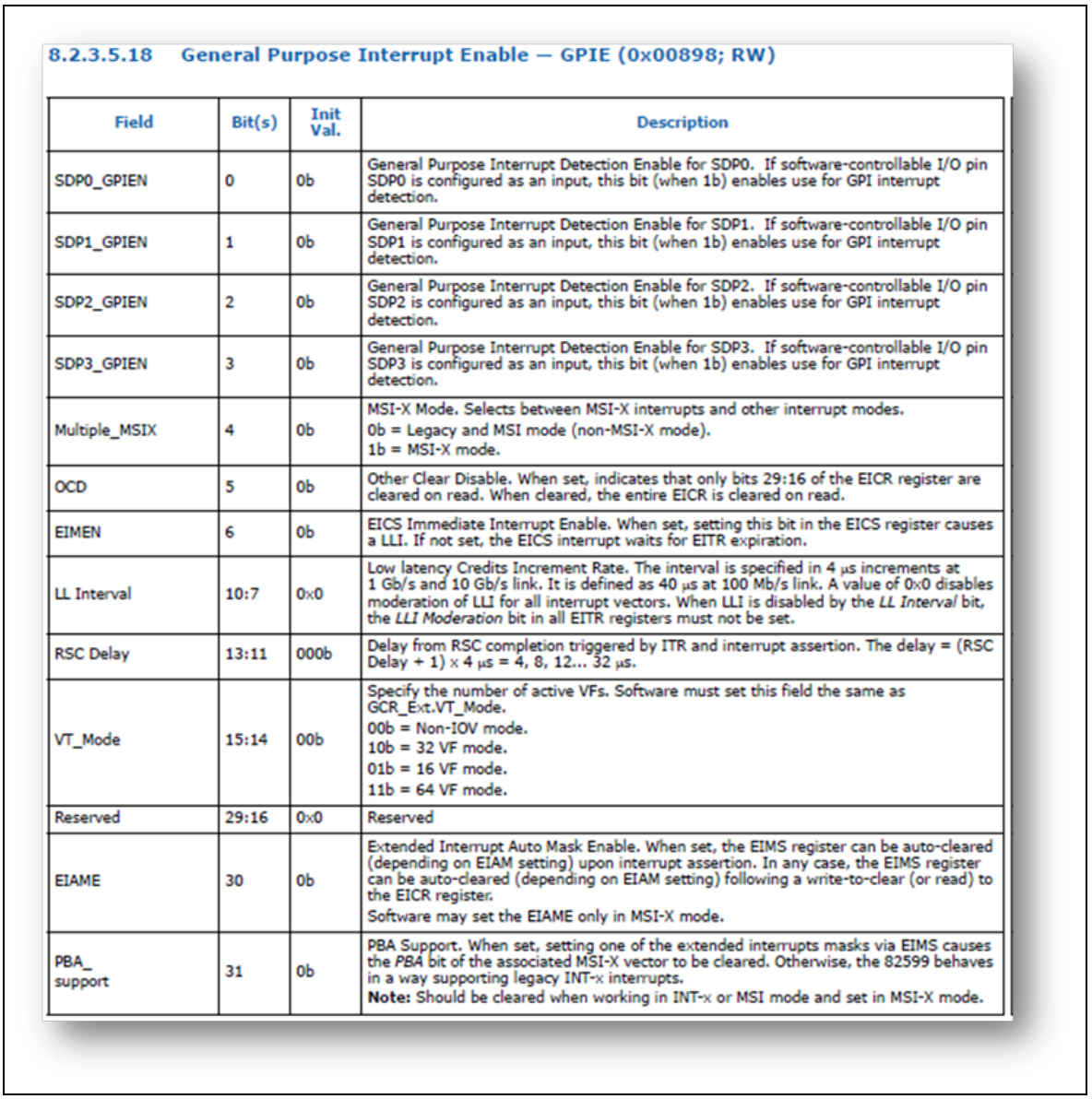
File: `ixgbe_main.c`

Function: `ixgbe_up_complete`

```
if (adapter->flags & IXGBE_FLAG_SRIOV_ENABLED) {
    gpie &= ~IXGBE_GPIE_VTMODE_MASK;
    gpie |= IXGBE_GPIE_VTMODE_64;
}

IXGBE_WRITE_REG(hw, IXGBE_GPIE, gpie);
```

This code snippet configures VT_Mode bits (15:14) within General Purpose Interrupt Enable (GPIE -0x00898) Register for 64 VFs.



8.2.3.5.18 General Purpose Interrupt Enable — GPIE (0x00898; RW)

Field	Bit(s)	Init Val.	Description
SDP0_GPIEN	0	0b	General Purpose Interrupt Detection Enable for SDP0. If software-controllable I/O pin SDP0 is configured as an input, this bit (when 1b) enables use for GPI interrupt detection.
SDP1_GPIEN	1	0b	General Purpose Interrupt Detection Enable for SDP1. If software-controllable I/O pin SDP1 is configured as an input, this bit (when 1b) enables use for GPI interrupt detection.
SDP2_GPIEN	2	0b	General Purpose Interrupt Detection Enable for SDP2. If software-controllable I/O pin SDP2 is configured as an input, this bit (when 1b) enables use for GPI interrupt detection.
SDP3_GPIEN	3	0b	General Purpose Interrupt Detection Enable for SDP3. If software-controllable I/O pin SDP3 is configured as an input, this bit (when 1b) enables use for GPI interrupt detection.
Multiple_MSIX	4	0b	MSI-X Mode. Selects between MSI-X interrupts and other interrupt modes. 0b = Legacy and MSI mode (non-MSI-X mode). 1b = MSI-X mode.
OCD	5	0b	Other Clear Disable. When set, indicates that only bits 29:16 of the EICR register are cleared on read. When cleared, the entire EICR is cleared on read.
EIMEN	6	0b	EICS Immediate Interrupt Enable. When set, setting this bit in the EICS register causes a LLI. If not set, the EICS interrupt waits for EITR expiration.
LL Interval	10:7	0x0	Low latency Credits Increment Rate. The interval is specified in 4 µs increments at 1 Gb/s and 10 Gb/s link. It is defined as 40 µs at 100 Mb/s link. A value of 0x0 disables moderation of LLI for all interrupt vectors. When LLI is disabled by the *LL Interval* bit, the *LLI Moderation* bit in all EITR registers must not be set.
RSC Delay	13:11	000b	Delay from RSC completion triggered by ITR and interrupt assertion. The delay = (RSC Delay + 1) x 4 µs = 4, 8, 12... 32 µs.
VT_Mode	15:14	00b	Specify the number of active VFs. Software must set this field the same as GCR_Ext.VT_Mode. 00b = Non-IOV mode. 10b = 32 VF mode. 01b = 16 VF mode. 11b = 64 VF mode.
Reserved	29:16	0x0	Reserved
EIAME	30	0b	Extended Interrupt Auto Mask Enable. When set, the EIMS register can be auto-cleared (depending on EIAM setting) upon interrupt assertion. In any case, the EIMS register can be auto-cleared (depending on EIAM setting) following a write-to-clear (or read) to the EICR register. Software may set the EIAME only in MSI-X mode.
PBA_ support	31	0b	PBA Support. When set, setting one of the extended interrupts masks via EIMS causes the *PBA* bit of the associated MSI-X vector to be cleared. Otherwise, the 82599 behaves in a way supporting legacy INT-x interrupts. **Note:** Should be cleared when working in INT-x or MSI mode and set in MSI-X mode.

Figure 14. General Purpose Interrupt Enable Register Definition from the Datasheet

6.2.2 Enabling VF to VF Bridging

The enabling of the VF to VF bridge is done within the ixgbe_configure_rx() function in ixgbe_main.c. This function enables the Loopback Enable bit (LBE) of the PF DMA Tx General Switch Control (PFDTXGSWC) register.

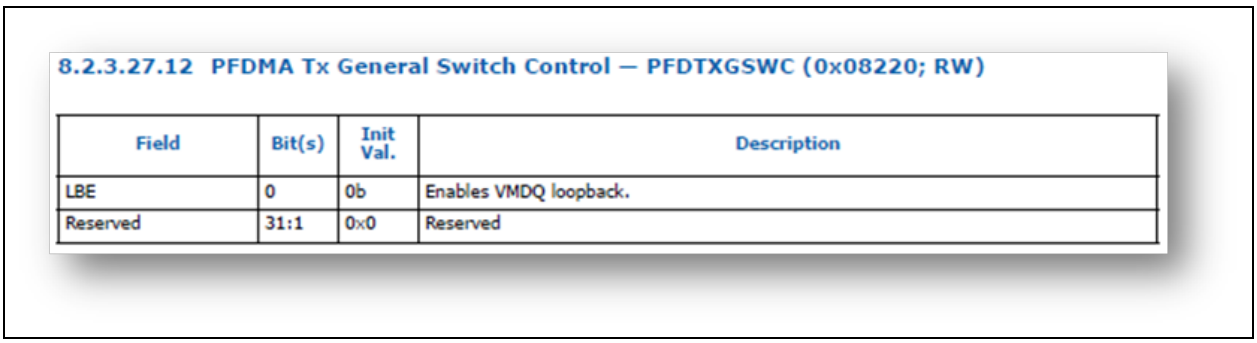
File: ixgbe_main.c

Function: ixgbe_configure_rx

```
#ifdef CONFIG_PCI_IOV
       if (adapter->flags & IXGBE_FLAG_SRIOV_ENABLED) {
          IXGBE_WRITE_REG(hw, IXGBE_PFDTXGSWC,
                     IXGBE_PFDTXGSWC_VT_LBEN);

          ixgbe_set_vmolr(hw, adapter->num_vfs);
       }
#endif
```

The snippet of code also enables VF offloads via the ixgbe_set_vmolr function call. The offloads are discussed in detail later in this document.



8.2.3.27.12 PFDMA Tx General Switch Control — PFDTXGSWC (0x08220; RW)

Field	Bit(s)	Init Val.	Description
LBE	0	0b	Enables VMDQ loopback.
Reserved	31:1	0x0	Reserved

Figure 15. PFDTXGSWC Register definition from the Datasheet

6.2.3 Default Pool

The default pool is the place where packets not sent to a specific VF (based upon VLAN or MAC address filtering) are sent for processing. This is the pool resource reserved for the PF.

The default pool is configured to be the first available pool not assigned to a VF. So, if there are 32 VFs configured, pool number 33 is configured to be the default.

Files: ixgbe_main.c

Function: ixgbe_configure_rx

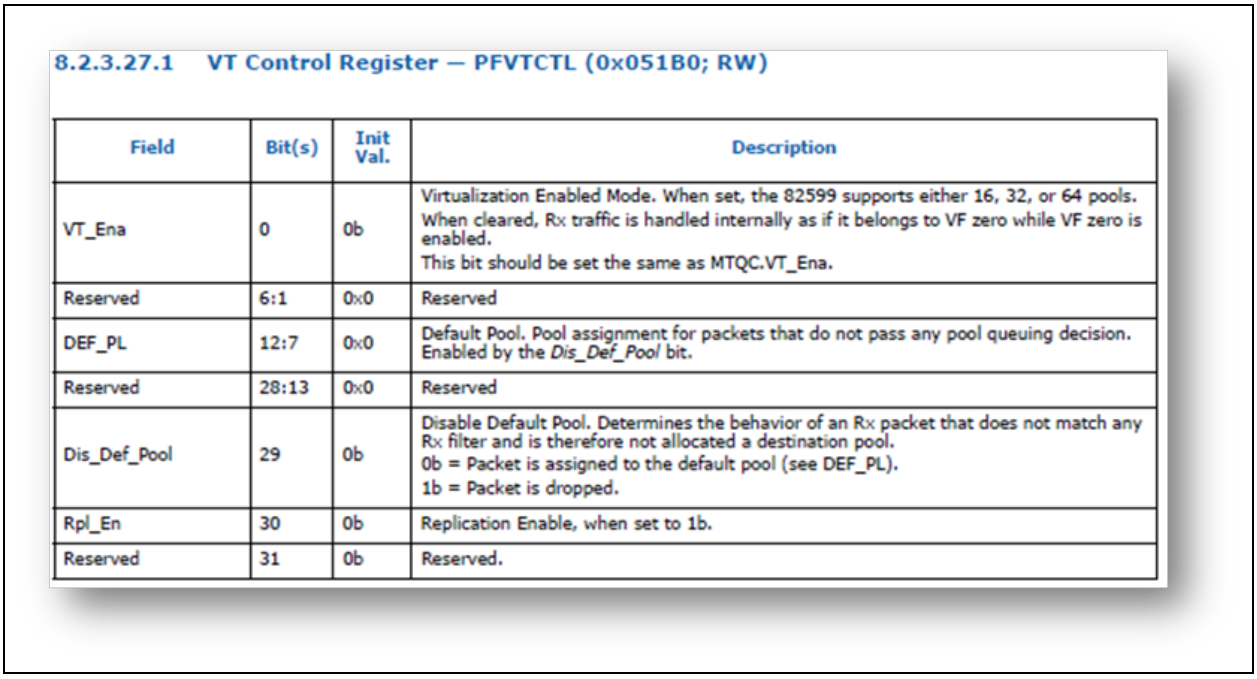
```
if (adapter->num_vfs) {
   vt_reg_bits &= ~IXGBE_VT_CTL_POOL_MASK;
   vt_reg_bits |= (adapter->num_vfs <<
          IXGBE_VT_CTL_POOL_SHIFT);
}

u32 vt_reg;
u32 vt_reg_bits;
if (hw->mac.type == ixgbe_mac_82599EB) {
   vt_reg = IXGBE_VT_CTL;
   vt_reg_bits = IXGBE_VMD_CTL_VMDQ_EN
          | IXGBE_VT_CTL_REPLEN;
   if (adapter->num_vfs) {
      vt_reg_bits &= ~IXGBE_VT_CTL_POOL_MASK;
      vt_reg_bits |= (adapter->num_vfs <<
             IXGBE_VT_CTL_POOL_SHIFT);
```

```
    }
} else {
    vt_reg = IXGBE_VMD_CTL;
    vt_reg_bits = IXGBE_VMD_CTL_VMDQ_EN;
}
vmdctl = IXGBE_READ_REG(hw, vt_reg);
IXGBE_WRITE_REG(hw, vt_reg, vmdctl | vt_reg_bits);
```

The driver manipulates the PFVTCTL (0x051B0) register to configure the default pool; specifically the DEF_PL bits (12:7).

NOTE: This code uses the name of IXGBE_VT_CTL as the name of the PFVTCTL register.



8.2.3.27.1 VT Control Register — PFVTCTL (0x051B0; RW)

Field	Bit(s)	Init Val.	Description
VT_Ena	0	0b	Virtualization Enabled Mode. When set, the 82599 supports either 16, 32, or 64 pools. When cleared, Rx traffic is handled internally as if it belongs to VF zero while VF zero is enabled. This bit should be set the same as MTQC.VT_Ena.
Reserved	6:1	0x0	Reserved
DEF_PL	12:7	0x0	Default Pool. Pool assignment for packets that do not pass any pool queuing decision. Enabled by the *Dis_Def_Pool* bit.
Reserved	28:13	0x0	Reserved
Dis_Def_Pool	29	0b	Disable Default Pool. Determines the behavior of an Rx packet that does not match any Rx filter and is therefore not allocated a destination pool. 0b = Packet is assigned to the default pool (see DEF_PL). 1b = Packet is dropped.
Rpl_En	30	0b	Replication Enable, when set to 1b.
Reserved	31	0b	Reserved.

Figure 16. VT Control Register definition from the Datasheet

6.2.4 Replication Enable

This is the Broadcast and Multicast replication capability. It replicates incoming Broadcast and Multicast to all pools/VFs.

Files: `ixgbe_main.c`

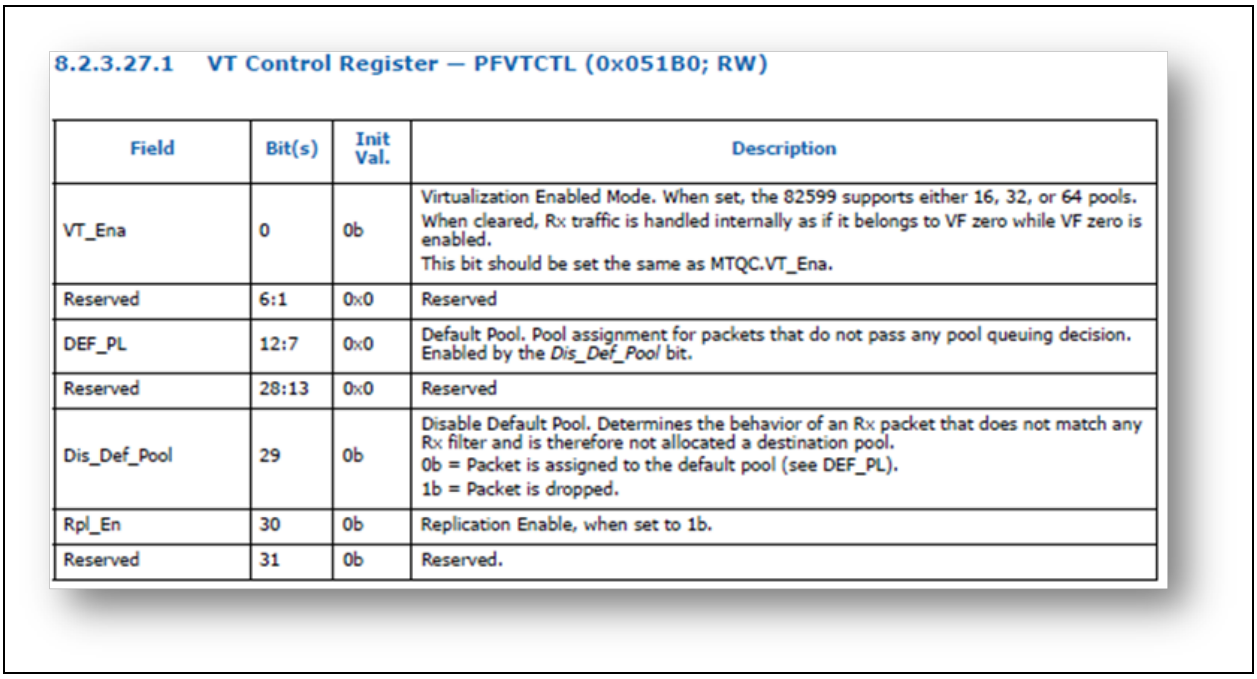
Function: `ixgbe_configure_rx`

```
vt_reg = IXGBE_VT_CTL;
vt_reg_bits = IXGBE_VMD_CTL_VMDQ_EN
      | IXGBE_VT_CTL_REPLEN;

IXGBE_WRITE_REG(hw, vt_reg, vmdctl | vt_reg_bits);
```

The driver manipulates the VT Control Register (PFVTCTL -0x051B0) register to configure the default pool; specifically the Rpl_En bit (30).

NOTE: This code uses the name of IXGBE_VT_CTL as the name of the PFVTCTL register.



8.2.3.27.1 VT Control Register — PFVTCTL (0x051B0; RW)

Field	Bit(s)	Init Val.	Description
VT_Ena	0	0b	Virtualization Enabled Mode. When set, the 82599 supports either 16, 32, or 64 pools. When cleared, Rx traffic is handled internally as if it belongs to VF zero while VF zero is enabled. This bit should be set the same as MTQC.VT_Ena.
Reserved	6:1	0x0	Reserved
DEF_PL	12:7	0x0	Default Pool. Pool assignment for packets that do not pass any pool queuing decision. Enabled by the *Dis_Def_Pool* bit.
Reserved	28:13	0x0	Reserved
Dis_Def_Pool	29	0b	Disable Default Pool. Determines the behavior of an Rx packet that does not match any Rx filter and is therefore not allocated a destination pool. 0b = Packet is assigned to the default pool (see DEF_PL). 1b = Packet is dropped.
Rpl_En	30	0b	Replication Enable, when set to 1b.
Reserved	31	0b	Reserved.

Figure 17. VT Control Register definition from the Datasheet

6.2.5 Broadcast Accept Mode

This allows a VF to accept Broadcast Packets.

By default, Broadcast Accept Mode is enabled for all VFs.

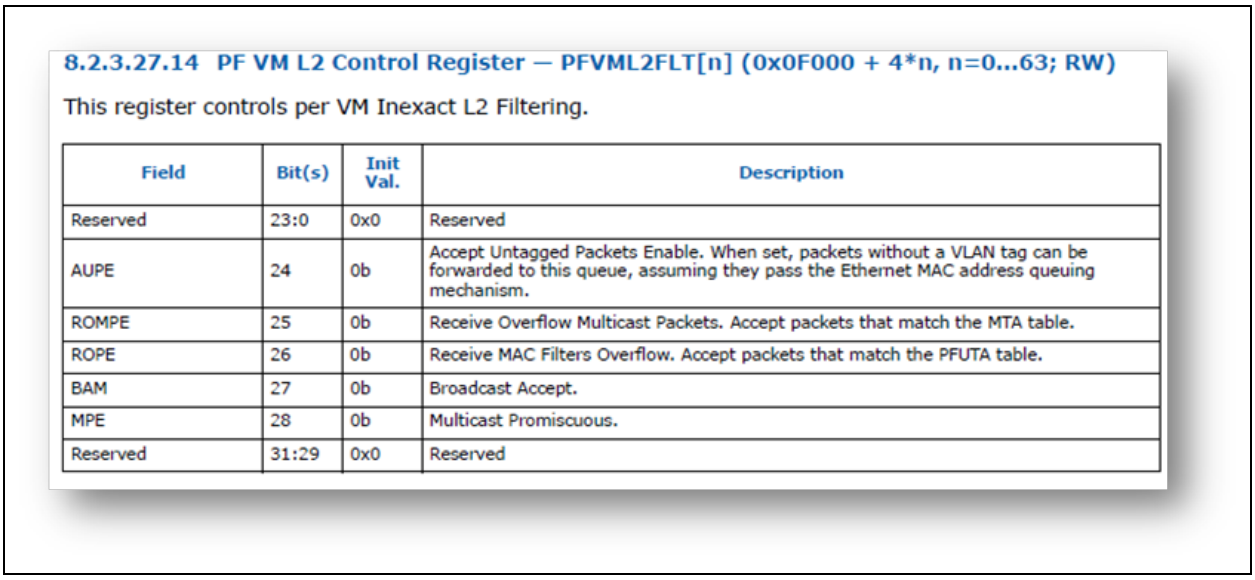
File: `ixgbe_sriov.c`

Function: `ixgbe_set_vmolr`

```
u32 vmolr = IXGBE_READ_REG(hw, IXGBE_VMOLR(vf));
vmolr |= (IXGBE_VMOLR_AUPE |
      IXGBE_VMOLR_ROMPE |
      IXGBE_VMOLR_ROPE |
      IXGBE_VMOLR_BAM);
IXGBE_WRITE_REG(hw, IXGBE_VMOLR(vf), vmolr);
```

The driver manipulates the BAM field (bit 27) of the PF VM L2 Control Register (PFVML2FLT 0x0F000 + 4*n [n=0..63]) register to enable/disable Broadcast Accept Mode for each VF.

NOTE: This code uses the name of IXGBE_VMOLR as the name of the PFVML2FLT register.



8.2.3.27.14 PF VM L2 Control Register — PFVML2FLT[n] (0x0F000 + 4*n, n=0...63; RW)

This register controls per VM Inexact L2 Filtering.

Field	Bit(s)	Init Val.	Description
Reserved	23:0	0x0	Reserved
AUPE	24	0b	Accept Untagged Packets Enable. When set, packets without a VLAN tag can be forwarded to this queue, assuming they pass the Ethernet MAC address queuing mechanism.
ROMPE	25	0b	Receive Overflow Multicast Packets. Accept packets that match the MTA table.
ROPE	26	0b	Receive MAC Filters Overflow. Accept packets that match the PFUTA table.
BAM	27	0b	Broadcast Accept.
MPE	28	0b	Multicast Promiscuous.
Reserved	31:29	0x0	Reserved

Figure 18. PFVML2FLT Register Definition from Datasheet

6.2.6 Accept Packets Matching PFUTA Table

This allows a VF to accept packets that match the PF Unicast address entries in the Unicast Table Array (PFUTA 0x0x0F400).

The default is to accept.

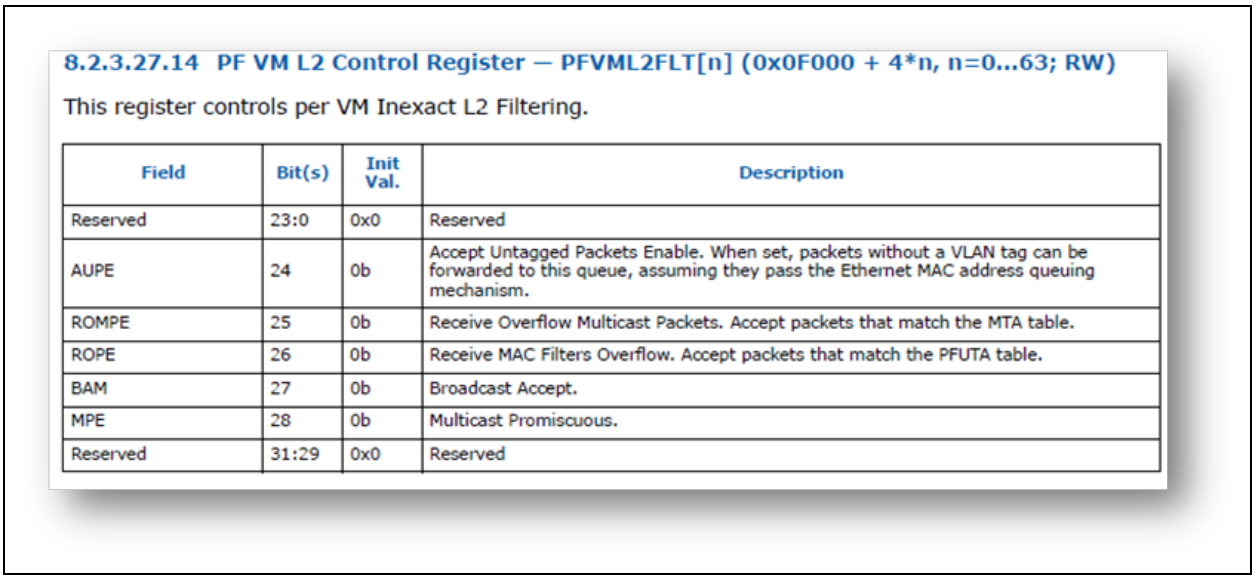
File: `ixgbe_sriov.c`

Function: `ixgbe_set_vmolr`

```
u32 vmolr = IXGBE_READ_REG(hw, IXGBE_VMOLR(vf));
vmolr |= (IXGBE_VMOLR_AUPE |
      IXGBE_VMOLR_ROMPE |
      IXGBE_VMOLR_ROPE |
      IXGBE_VMOLR_BAM);
IXGBE_WRITE_REG(hw, IXGBE_VMOLR(vf), vmolr);
```

The driver manipulates the ROPE field (bit 26) of the PF VM L2 Control Register (PFVML2FLT 0x0F000 + 4*n [n=0..63]) register to enable/disable accepting packets from the PFUTA Table for each VF.

NOTE: This code uses the name of IXGBE_VMOLR as the name of the PFVML2FLT register.



8.2.3.27.14 PF VM L2 Control Register — PFVML2FLT[n] (0x0F000 + 4*n, n=0...63; RW)

This register controls per VM Inexact L2 Filtering.

Field	Bit(s)	Init Val.	Description
Reserved	23:0	0x0	Reserved
AUPE	24	0b	Accept Untagged Packets Enable. When set, packets without a VLAN tag can be forwarded to this queue, assuming they pass the Ethernet MAC address queuing mechanism.
ROMPE	25	0b	Receive Overflow Multicast Packets. Accept packets that match the MTA table.
ROPE	26	0b	Receive MAC Filters Overflow. Accept packets that match the PFUTA table.
BAM	27	0b	Broadcast Accept.
MPE	28	0b	Multicast Promiscuous.
Reserved	31:29	0x0	Reserved

Figure 19. PFVML2FLT Register Definition from Datasheet

6.2.7 Accept Packets Matching MTA Table

This allows a VF to accept packets that match the Multicast address entries in the Multicast Table Array (UTA 0xA000).

The default is to accept.

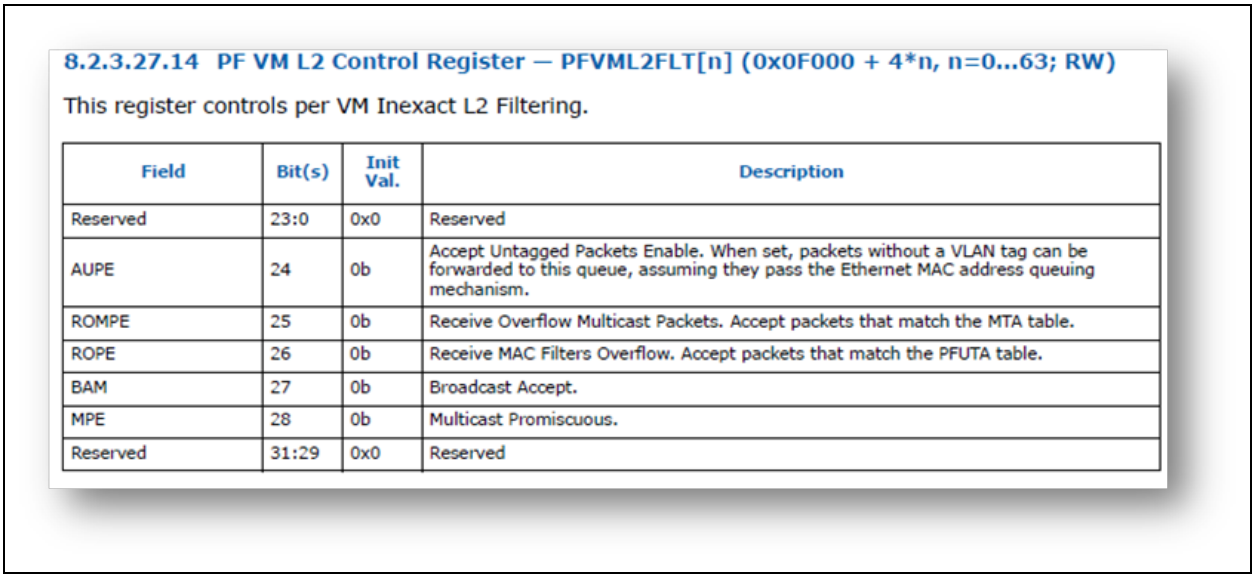
File: `ixgbe_sriov.c`

Function: `ixgbe_set_vmolr`

```
u32 vmolr = IXGBE_READ_REG(hw, IXGBE_VMOLR(vf));
vmolr |= (IXGBE_VMOLR_AUPE |
      IXGBE_VMOLR_ROMPE |
      IXGBE_VMOLR_ROPE |
      IXGBE_VMOLR_BAM);
IXGBE_WRITE_REG(hw, IXGBE_VMOLR(vf), vmolr);
```

The driver manipulates the ROMPE field (bit 25) of the PF VM L2 Control Register (PFVML2FLT 0x0F000 + 4*n [n=0..63]) register to enable/disable accepting packets from the MTA Table for each VF.

NOTE: This code uses the name of IXGBE_VMOLR as the name of the PFVML2FLT register.



8.2.3.27.14 PF VM L2 Control Register — PFVML2FLT[n] (0x0F000 + 4*n, n=0...63; RW)

This register controls per VM Inexact L2 Filtering.

Field	Bit(s)	Init Val.	Description
Reserved	23:0	0x0	Reserved
AUPE	24	0b	Accept Untagged Packets Enable. When set, packets without a VLAN tag can be forwarded to this queue, assuming they pass the Ethernet MAC address queuing mechanism.
ROMPE	25	0b	Receive Overflow Multicast Packets. Accept packets that match the MTA table.
ROPE	26	0b	Receive MAC Filters Overflow. Accept packets that match the PFUTA table.
BAM	27	0b	Broadcast Accept.
MPE	28	0b	Multicast Promiscuous.
Reserved	31:29	0x0	Reserved

Figure 20. PFVML2FLT Register Definition from Datasheet

6.2.8 Accept Untagged Packets Enable

This allows a VF to accept packets without a matching VLAN tag as long as MAC address matches.

By default, Accept Untagged Packets Mode is enabled.

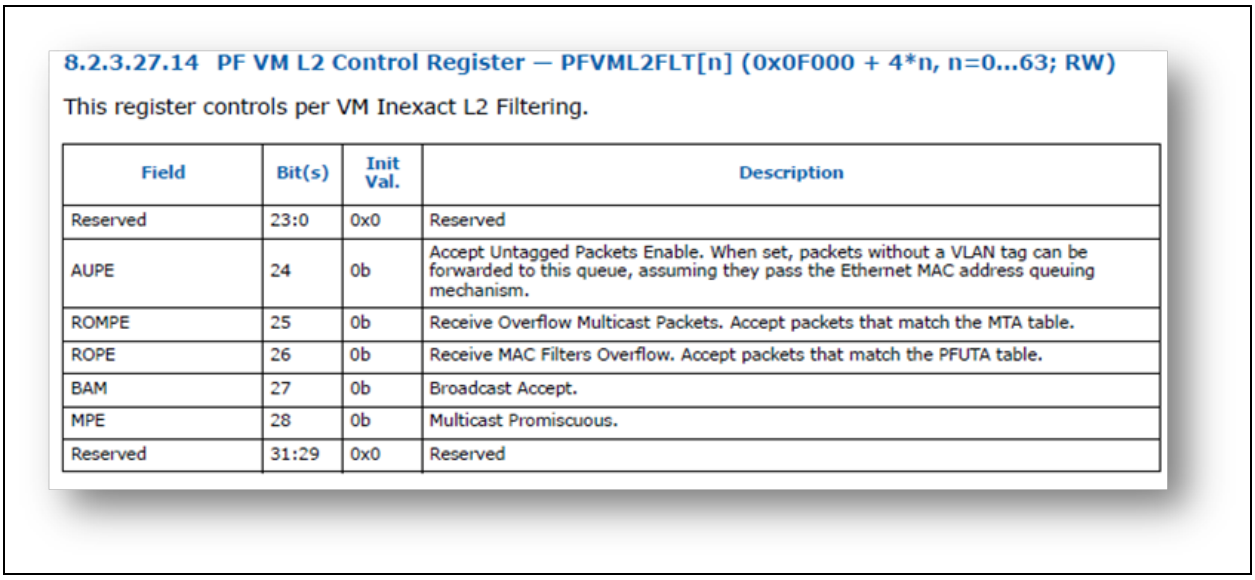
File: `ixgbe_sriov.c`

Function: `ixgbe_set_vmolr`

```
u32 vmolr = IXGBE_READ_REG(hw, IXGBE_VMOLR(vf));
vmolr |= (IXGBE_VMOLR_AUPE |
      IXGBE_VMOLR_ROMPE |
      IXGBE_VMOLR_ROPE |
      IXGBE_VMOLR_BAM);
IXGBE_WRITE_REG(hw, IXGBE_VMOLR(vf), vmolr);
```

The driver manipulates the AUPE field (bit 24) of the PF VM L2 Control Register (PFVML2FLT 0x0F000 + 4*n [n=0..63]) register to enable/disable Accept Untagged Packets Mode for each VF.

NOTE: This code uses the name of IXGBE_VMOLR as the name of the PFVML2FLT register.



8.2.3.27.14 PF VM L2 Control Register — PFVML2FLT[n] (0x0F000 + 4*n, n=0...63; RW)

This register controls per VM Inexact L2 Filtering.

Field	Bit(s)	Init Val.	Description
Reserved	23:0	0x0	Reserved
AUPE	24	0b	Accept Untagged Packets Enable. When set, packets without a VLAN tag can be forwarded to this queue, assuming they pass the Ethernet MAC address queuing mechanism.
ROMPE	25	0b	Receive Overflow Multicast Packets. Accept packets that match the MTA table.
ROPE	26	0b	Receive MAC Filters Overflow. Accept packets that match the PFUTA table.
BAM	27	0b	Broadcast Accept.
MPE	28	0b	Multicast Promiscuous.
Reserved	31:29	0x0	Reserved

Figure 21. PFVML2FLT Register Definition from Datasheet

6.2.9 Strip VLAN Tag for Incoming Packets

This allows a VF to have the VLAN Tag stripped from incoming packets after it has passed the L2 filtering.

By default, this is enabled.

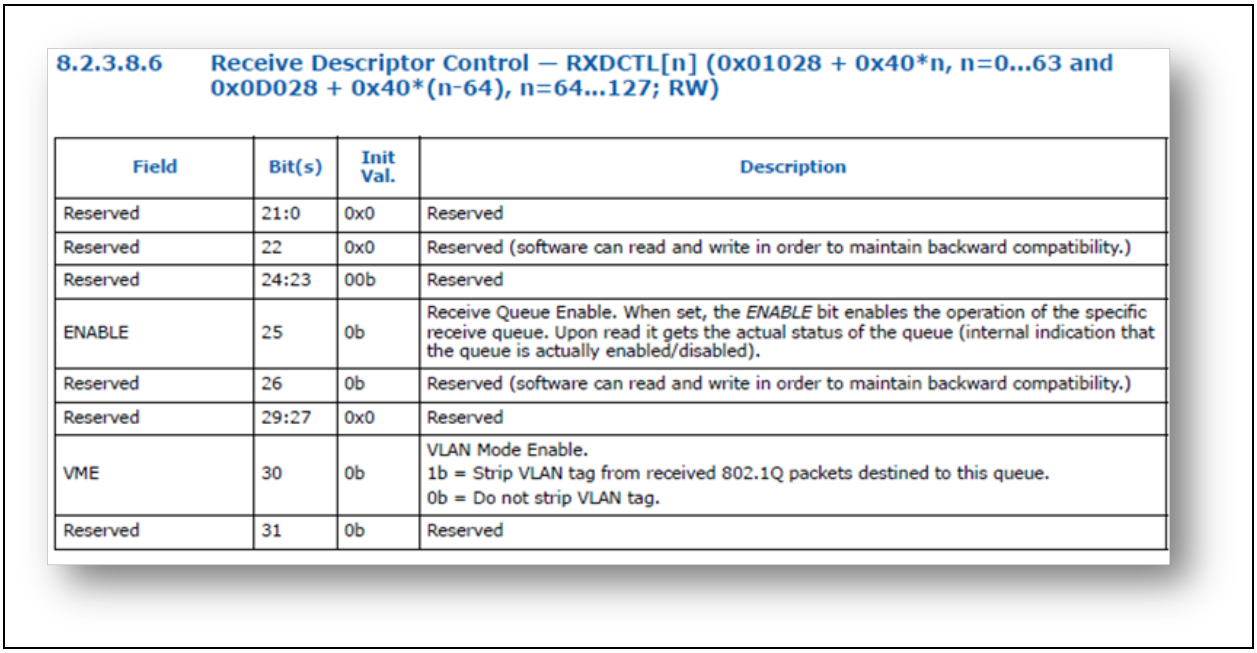
Files: `ixgbe_main.c`

Functions: `ixgbe_vlan_rx_register`

```
for (i = 0; i < adapter->num_rx_queues; i++) {
    j = adapter->rx_ring[i]->reg_idx;
    ctrl = IXGBE_READ_REG(&adapter->hw, IXGBE_RXDCTL(j));
    ctrl |= IXGBE_RXDCTL_VME;
    IXGBE_WRITE_REG(&adapter->hw, IXGBE_RXDCTL(j), ctrl);
}
```

The driver manipulates the VLAN Mode Enable field (bit 30) of the Receive Descriptor Control Register (RXDCTL[n] (0x01028 + 0x40*n, n=0..63 and 0x0D028 + 0x40*(n-64), n=64..127; RW)) register to enable/disable the stripping of VLAN tags from incoming packets.

NOTE: VLAN insertions are configured as part of the transmit descriptor from within the VF Driver.



8.2.3.8.6 Receive Descriptor Control — RXDCTL[n] (0x01028 + 0x40*n, n=0...63 and 0x0D028 + 0x40*(n-64), n=64...127; RW)

Field	Bit(s)	Init Val.	Description
Reserved	21:0	0x0	Reserved
Reserved	22	0x0	Reserved (software can read and write in order to maintain backward compatibility.)
Reserved	24:23	00b	Reserved
ENABLE	25	0b	Receive Queue Enable. When set, the *ENABLE* bit enables the operation of the specific receive queue. Upon read it gets the actual status of the queue (internal indication that the queue is actually enabled/disabled).
Reserved	26	0b	Reserved (software can read and write in order to maintain backward compatibility.)
Reserved	29:27	0x0	Reserved
VME	30	0b	VLAN Mode Enable. 1b = Strip VLAN tag from received 802.1Q packets destined to this queue. 0b = Do not strip VLAN tag.
Reserved	31	0b	Reserved

Figure 22. Receive Descriptor Control Register from the Datasheet

6.2.10 VF MAC Address Assignment

The PF Driver dynamically assigns each VF a MAC address using the random_ether_addr() function provided by the kernel. The following code snippet is an example:

File: `ixgbe_sriov.c`

Function: `ixgbe_vf_configuration`

```
random_ether_addr(vf_mac_addr);
memcpy(adapter->vfinfo[vfn].vf_mac_addresses, vf_mac_addr, 6);
```

The __devinit ixgbe_probe() routine calls the ixgbe_vf_configuration() once for each VF that has been created. The VF gets its MAC address when it performs function level reset and sends the IXGBE_VF_RESET message to the PF via the mailbox. The PF in response sends the VF MAC address as part of the reset ack.

In addition, the VF Driver can specify its own MAC address using the IXGBE_VF_SET_MAC_ADDR message, as described in Section 6.3.2.2.

6.3 Mailbox Messages

Because the PCIe resources exposed to a VF are limited, there are some actions that the VF Driver must request the PF Driver perform on its behalf. This is accomplished using the mailbox system discussed in Section 3 to pass messages back and forth.

6.3.1 PF Driver to VF Driver Messages

Currently the PF Driver only sends a single asynchronous message to the VF Driver – all other messages are in response to a message from the VF Driver.

6.3.1.1 IXGBE_PF_CONTROL_MSG

This message is sent from the PF Driver to each VF Driver at a regular interval as a sort of PING/Watchdog so that the VF Driver knows that the PF Driver is functioning.

File: `ixgbe_vf.c`

Function: `ixgbe_ping_all_vfs`

```
for (i = 0 ; i < adapter->num_vfs; i++) {
   ping = IXGBE_PF_CONTROL_MSG;
   if (adapter->vfinfo[i].clear_to_send)
      ping |= IXGBE_VT_MSGTYPE_CTS;
   ixgbe_write_mbx(hw, &ping, 1, i);
}
```

Section 5.4.1.1 details how the VF Driver handles this message.

6.3.2 VF Driver to PF Driver Messages

There are a number of messages that the VF Driver passes to the PF Driver using the mailbox system. These are discussed from the VF Driver side in Section 5.3. This section will detail the actions taken when receiving a specific message from the VF Driver by the PF Driver.

All message IDs are defined in the ixgbe_mbx.h file as part of both the PF and VF Drivers. When a VF Driver sends a message via the Mailbox, the PF Driver is notified via an interrupt, which it then services and calls the appropriate message handler.

6.3.2.1 IXGBE_VF_RESET

```
Message Handler Function: ixgbe_vf_reset_msg() : ixgbe_sriov.c
```

When this message is processed by the PF Driver, all settings regarding the VF are cleared including any MAC address the VF Driver may have set using the IXGBE_VF_SET_MAC_ADDR message (see Section 6.3.2.2). Any offloads, VLAN Tags or multicast addresses are cleared.

The VF is put into an initialized state, ready for reception and transmission. The MAC default address assigned to the VF is then returned to the VF Driver as part of an acknowledgement.

6.3.2.2 IXGBE_VF_SET_MAC_ADDR

```
Message Handler Function: ixgbe_set_vf_mac () : ixgbe_sriov.c
```

The VF Driver sends this message if it wants to configure its own MAC address – overriding the one generated by the PF Driver (see Section 6.2.10). The VF receives its default MAC Address in response to the IXGBE_VF_RESET message.

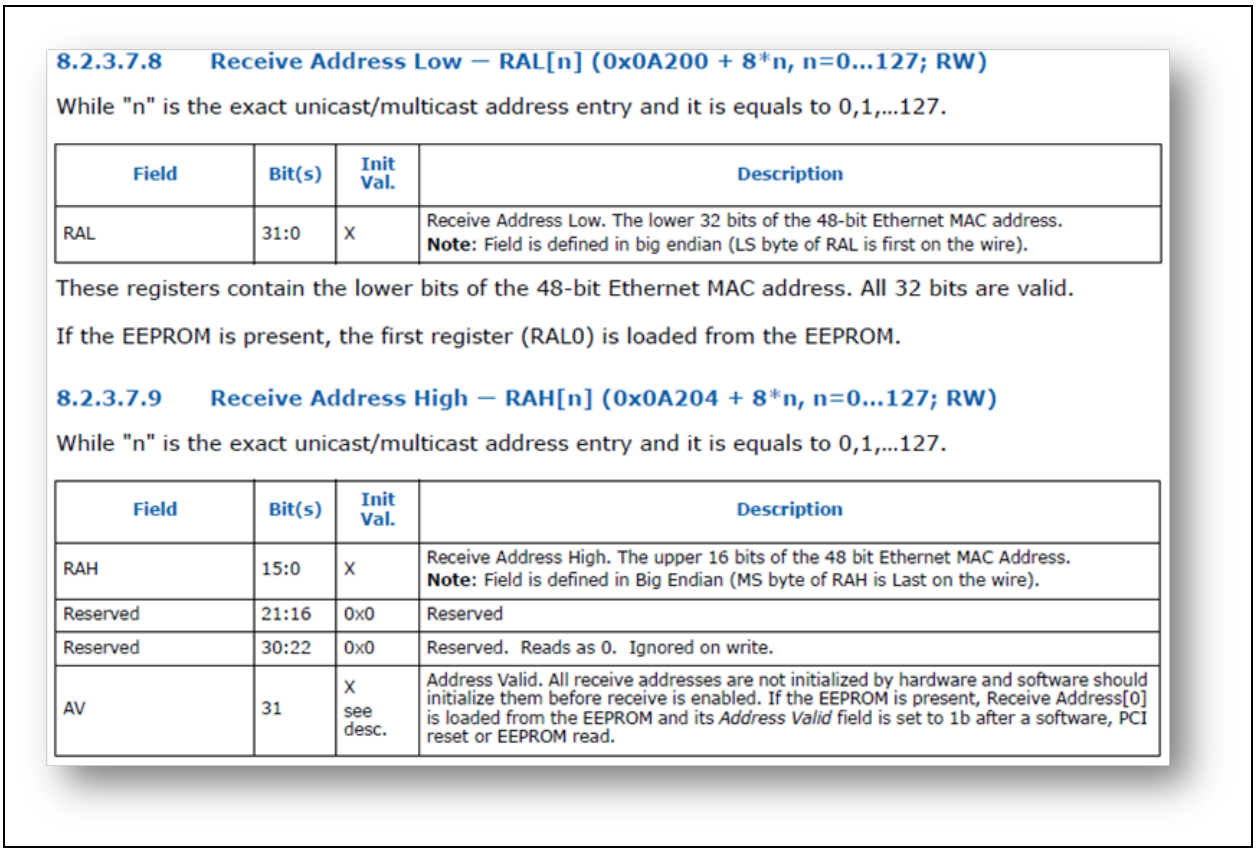
If successful the PF Driver returns an ACK message(IXGBE_VT_MSGTYPE_ACK) with the default VF MAC address, otherwise a NACK message (IXGBE_VT_MSGTYPE_NACK).

The PF Driver inserts the specified VF MAC address into the Receive Address Registers (0x0A200 and 0x0A204). Next the driver indicates which pool (VF) that MAC address gets assigned to by filling in the proper entry into the MAC Pool Select Array (MPSAR – 0x0A600) register an example of this is below, where the variable `vmdq` is the pool (VF) number and `rar` is the entry into the Receive Address Registers array where the MAC address is stored.

File: `ixgbe_common.c`

Function: `ixgbe_set_vmdq_generic`

```
if (vmdq < 32) {
    mpsar = IXGBE_READ_REG(hw, IXGBE_MPSAR_LO(rar));
    mpsar |= 1 << vmdq;
    IXGBE_WRITE_REG(hw, IXGBE_MPSAR_LO(rar), mpsar);
} else {
    mpsar = IXGBE_READ_REG(hw, IXGBE_MPSAR_HI(rar));
    mpsar |= 1 << (vmdq - 32);
    IXGBE_WRITE_REG(hw, IXGBE_MPSAR_HI(rar), mpsar);
}
```



8.2.3.7.8 Receive Address Low — RAL[n] (0x0A200 + 8*n, n=0...127; RW)

While "n" is the exact unicast/multicast address entry and it is equals to 0,1,...127.

Field	Bit(s)	Init Val.	Description
RAL	31:0	X	Receive Address Low. The lower 32 bits of the 48-bit Ethernet MAC address. **Note:** Field is defined in big endian (LS byte of RAL is first on the wire).

These registers contain the lower bits of the 48-bit Ethernet MAC address. All 32 bits are valid.

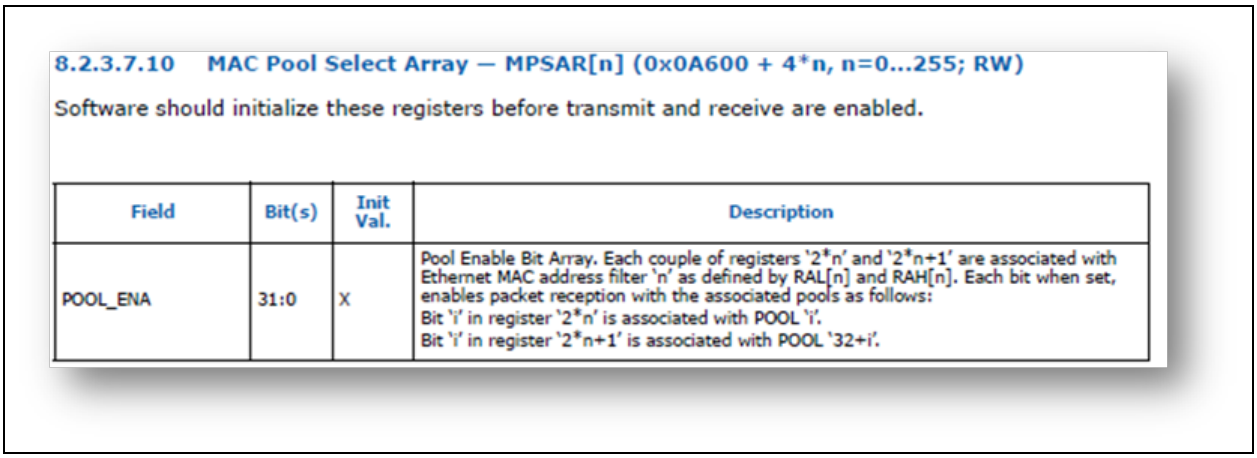
If the EEPROM is present, the first register (RAL0) is loaded from the EEPROM.

8.2.3.7.9 Receive Address High — RAH[n] (0x0A204 + 8*n, n=0...127; RW)

While "n" is the exact unicast/multicast address entry and it is equals to 0,1,...127.

Field	Bit(s)	Init Val.	Description
RAH	15:0	X	Receive Address High. The upper 16 bits of the 48 bit Ethernet MAC Address. **Note:** Field is defined in Big Endian (MS byte of RAH is Last on the wire).
Reserved	21:16	0x0	Reserved
Reserved	30:22	0x0	Reserved. Reads as 0. Ignored on write.
AV	31	X see desc.	Address Valid. All receive addresses are not initialized by hardware and software should initialize them before receive is enabled. If the EEPROM is present, Receive Address[0] is loaded from the EEPROM and its *Address Valid* field is set to 1b after a software, PCI reset or EEPROM read.

Figure 23. RAR Register Definitions from the Datasheet



8.2.3.7.10 MAC Pool Select Array — MPSAR[n] (0x0A600 + 4*n, n=0...255; RW)

Software should initialize these registers before transmit and receive are enabled.

Field	Bit(s)	Init Val.	Description
POOL_ENA	31:0	X	Pool Enable Bit Array. Each couple of registers '2*n' and '2*n+1' are associated with Ethernet MAC address filter 'n' as defined by RAL[n] and RAH[n]. Each bit when set, enables packet reception with the associated pools as follows: Bit 'i' in register '2*n' is associated with POOL 'i'. Bit 'i' in register '2*n+1' is associated with POOL '32+i'.

Figure 24. MAC Pool Select Array definition from the Datasheet

6.3.2.3 IXGBE_VF_SET_MULTICAST

```
Message Handler Function: ixgbe_set_vf_multicasts() : ixgbe_sriov.c
```

The VF Driver sends a hash table of multicast addresses as part of this message. The PF Driver takes this hash table and adds it to the MTA Hash Table. Refer to the *Intel® 82599 10 Gigabit Ethernet Controller Datasheet* for information regarding the MTA Hash Table.

If successful the PF Driver returns an ACK message (IXGBE_VT_MSGTYPE_ACK), otherwise a NACK message (IXGBE_VT_MSGTYPE_NACK).

6.3.2.4 IXGBE_VF_SET_VLAN

```
Message Handler Function: ixgbe_set_vf_vlan() : ixgbe_sriov.c
```

When the PF Driver receives this message it attempts to update the global VLAN Tag table (the Vland Fitler Table Array – VFTA (0xA000). Then associates the VF with the specific Tag.

If successful the PF Driver returns an ACK message (IXGBE_VT_MSGTYPE_ACK), otherwise a NACK message (IXGBE_VT_MSGTYPE_NACK).

The Intel® 82599 10 Gigabit Ethernet Controller supports up to 64 VLANs shared among all VF's.

The PF Driver configures the PF VM VLAN Pool Filter (PFVLVF) to specify a VLAN Tag for a pool and then the PFVM VLAN Pool Filter Bitmap (PFVLVFB) to indicate the pool (VF) to be associated with that VLAN filter.

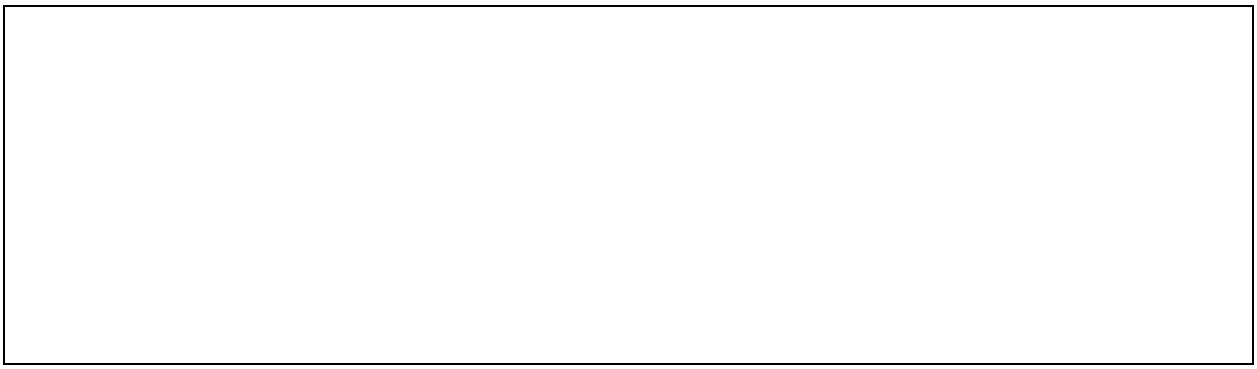
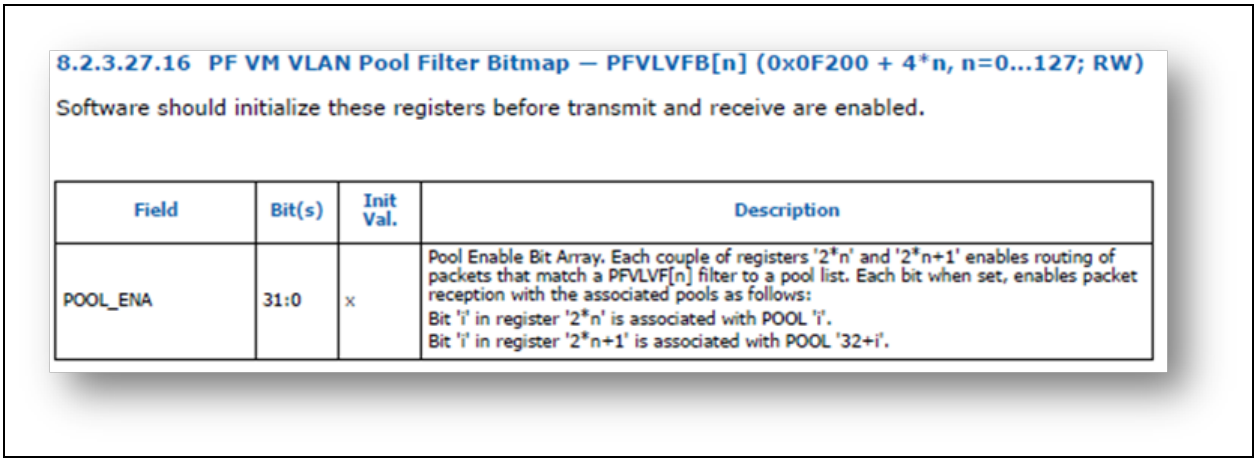


Figure 25. PF VM VLAN Pool Filter Register from the Datasheet



8.2.3.27.16 PF VM VLAN Pool Filter Bitmap — PFVLVFB[n] (0x0F200 + 4*n, n=0...127; RW)

Software should initialize these registers before transmit and receive are enabled.

Field	Bit(s)	Init Val.	Description
POOL_ENA	31:0	x	Pool Enable Bit Array. Each couple of registers '2*n' and '2*n+1' enables routing of packets that match a PFVLVF[n] filter to a pool list. Each bit when set, enables packet reception with the associated pools as follows: Bit 'i' in register '2*n' is associated with POOL 'i'. Bit 'i' in register '2*n+1' is associated with POOL '32+i'.

Figure 26. PF VM VLAN Pool Filter Bitmap Register from the Datasheet

7 Thoughts for Customization

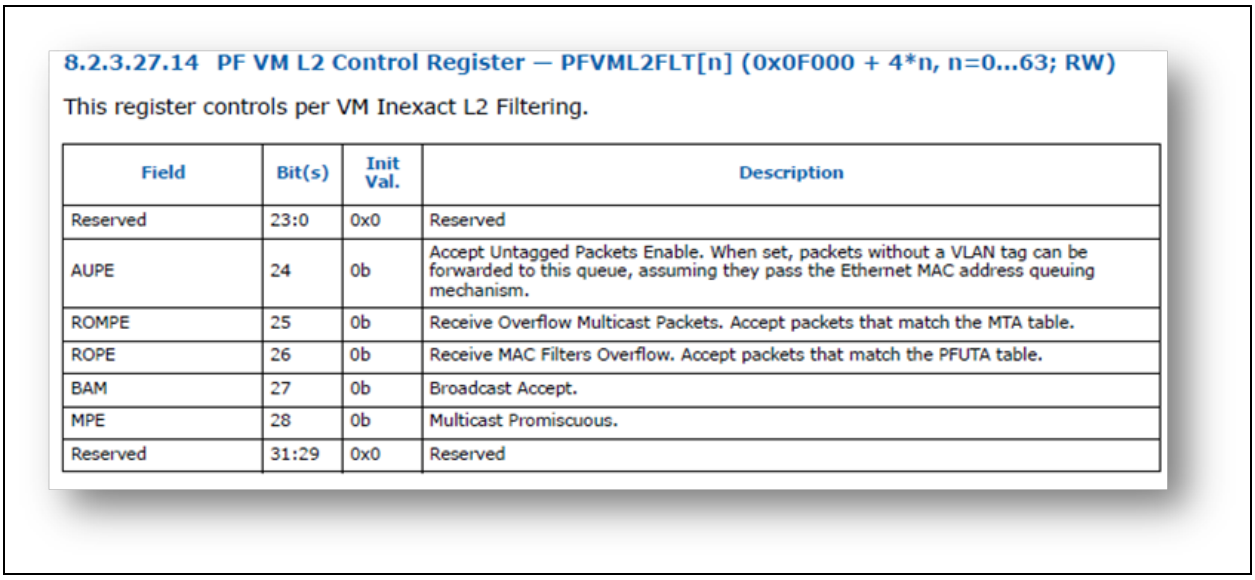
The PF and VF Drivers written by Intel provide a robust set of capabilities, including MAC Address filtering, VLAN Tag Filtering, configuring MAC addresses for the VF etc. The 82599 has a large number of capabilities that are not, at this time, exploited by the reference drivers provided by Intel.

This section highlights features of the 82599 that customers may want to investigate supporting.

7.1 Multicast Promiscuous Enable

The controller has provisions to allow each VF to be put into Multicast Promiscuous mode. The Intel reference driver does not configure this option.

The capability can be enabled/disabled by manipulating the MPE field (bit 28) of the PF VF L2 Control Register (PFVML2FLT – 0x0F000).



8.2.3.27.14 PF VM L2 Control Register — PFVML2FLT[n] (0x0F000 + 4*n, n=0...63; RW)

This register controls per VM Inexact L2 Filtering.

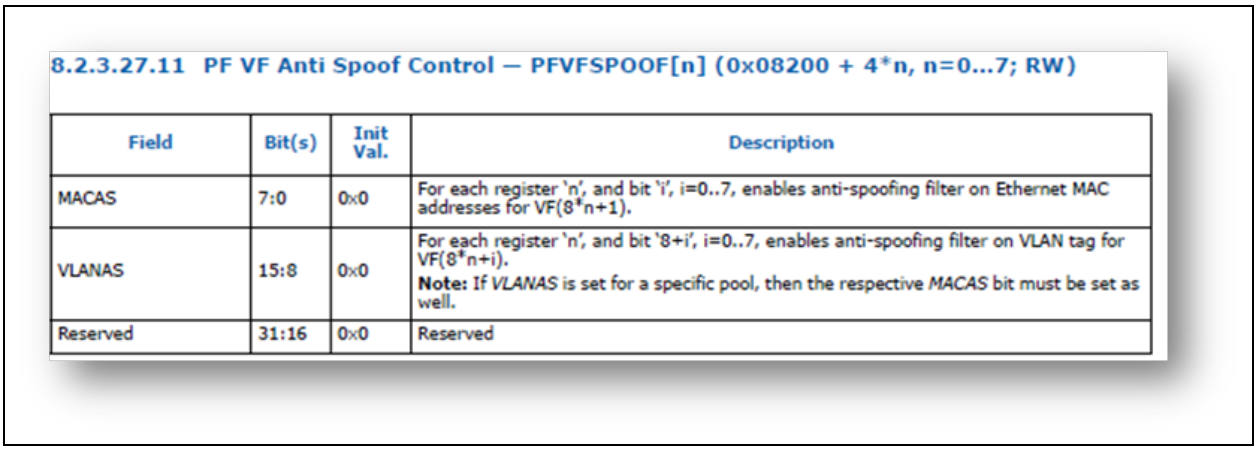
Field	Bit(s)	Init Val.	Description
Reserved	23:0	0x0	Reserved
AUPE	24	0b	Accept Untagged Packets Enable. When set, packets without a VLAN tag can be forwarded to this queue, assuming they pass the Ethernet MAC address queuing mechanism.
ROMPE	25	0b	Receive Overflow Multicast Packets. Accept packets that match the MTA table.
ROPE	26	0b	Receive MAC Filters Overflow. Accept packets that match the PFUTA table.
BAM	27	0b	Broadcast Accept.
MPE	28	0b	Multicast Promiscuous.
Reserved	31:29	0x0	Reserved

Figure 27. PFVML2FLT Register Definition from Datasheet

7.2 MAC Anti Spoofing

This is the Anti-Spoofing filtering based upon MAC address.

The capability can be added by setting the MACAS field (bits 0:7) of the PF VF Anti-Spoof Control (PFVFSPOOF - 0x3500) Register to enable anti-spoofing based upon MAC address for each VF.

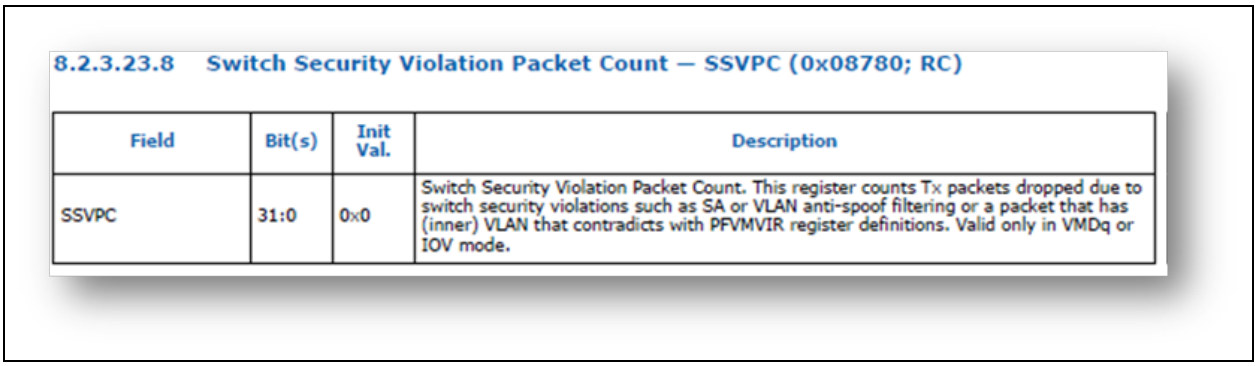


8.2.3.27.11 PF VF Anti Spoof Control — PFVFSPOOF[n] (0x08200 + 4*n, n=0...7; RW)

Field	Bit(s)	Init Val.	Description
MACAS	7:0	0x0	For each register 'n', and bit 'i', i=0..7, enables anti-spoofing filter on Ethernet MAC addresses for VF(8*n+1).
VLANAS	15:8	0x0	For each register 'n', and bit '8+i', i=0..7, enables anti-spoofing filter on VLAN tag for VF(8*n+i). **Note:** If *VLANAS* is set for a specific pool, then the respective *MACAS* bit must be set as well.
Reserved	31:16	0x0	Reserved

Figure 28. PF Anti Spoof Control Register definition from the Datasheet

If a spoof condition is detected, the packet being transmitted is discarded and the Switch Security Violation Packet Count (SSVPC – 0x08780) is incremented.



8.2.3.23.8 Switch Security Violation Packet Count — SSVPC (0x08780; RC)

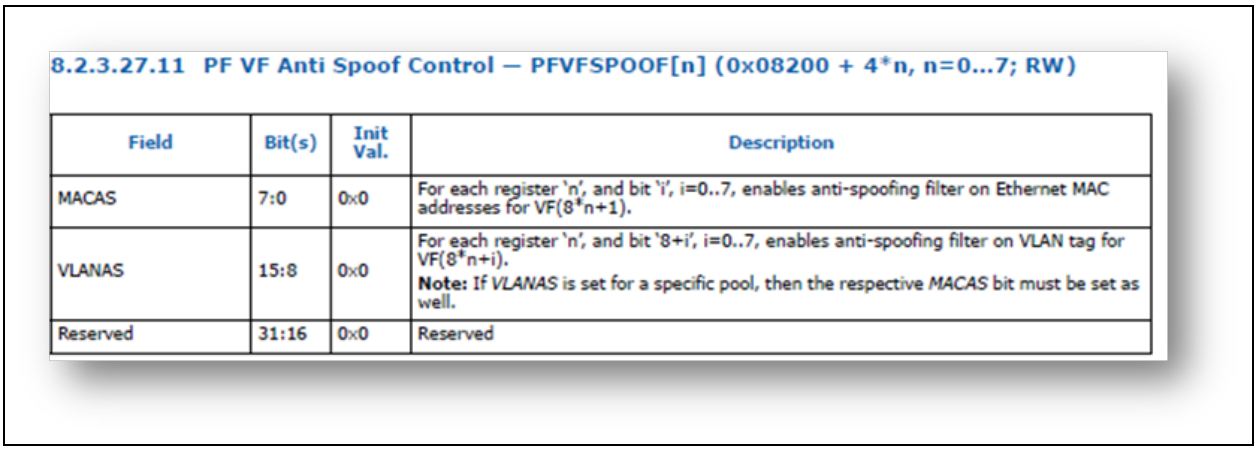
Field	Bit(s)	Init Val.	Description
SSVPC	31:0	0x0	Switch Security Violation Packet Count. This register counts Tx packets dropped due to switch security violations such as SA or VLAN anti-spoof filtering or a packet that has (inner) VLAN that contradicts with PFVMVIR register definitions. Valid only in VMDq or IOV mode.

Figure 29. Switch Security Violation Packet Count definition from the Datasheet

7.3 VLAN Tag Anti Spoofing

This is Anti-Spoof filtering based on the VLAN Tag address.

This capability can be added by setting the VLANAS field (bits 15:8) of the PF VF Anti-Spoof Control (PFVFSPOOF - 0x3500) Register to enable anti-spoofing based on the VLAN Tag for each VF.

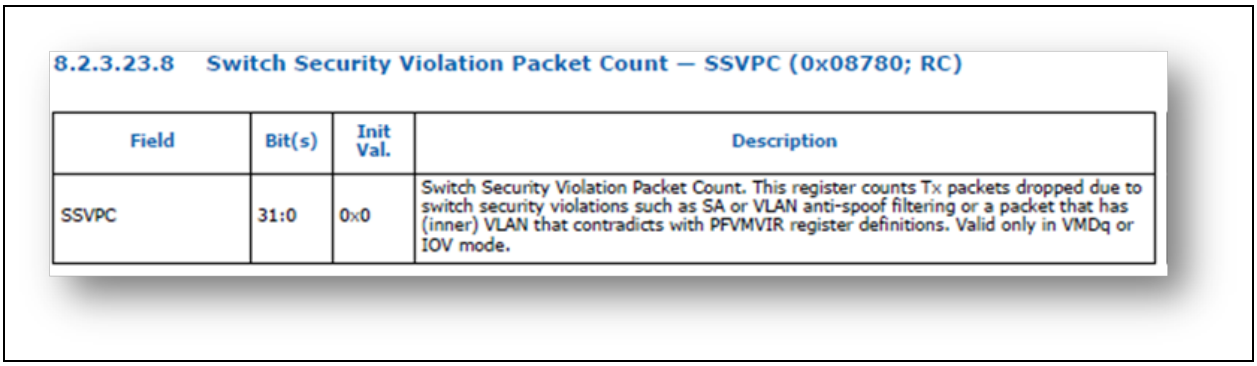


8.2.3.27.11 PF VF Anti Spoof Control — PFVFSPOOF[n] (0x08200 + 4*n, n=0...7; RW)

Field	Bit(s)	Init Val.	Description
MACAS	7:0	0x0	For each register 'n', and bit 'i', i=0..7, enables anti-spoofing filter on Ethernet MAC addresses for VF(8*n+1).
VLANAS	15:8	0x0	For each register 'n', and bit '8+i', i=0..7, enables anti-spoofing filter on VLAN tag for VF(8*n+i). **Note:** If *VLANAS* is set for a specific pool, then the respective *MACAS* bit must be set as well.
Reserved	31:16	0x0	Reserved

Figure 30. PF Anti Spoof Control Register definition from the Datasheet

If a spoof condition is detected, the packet being transmitted is discarded and the Switch Security Violation Packet Count (SSVPC – 0x08780) is incremented.



8.2.3.23.8 Switch Security Violation Packet Count — SSVPC (0x08780; RC)

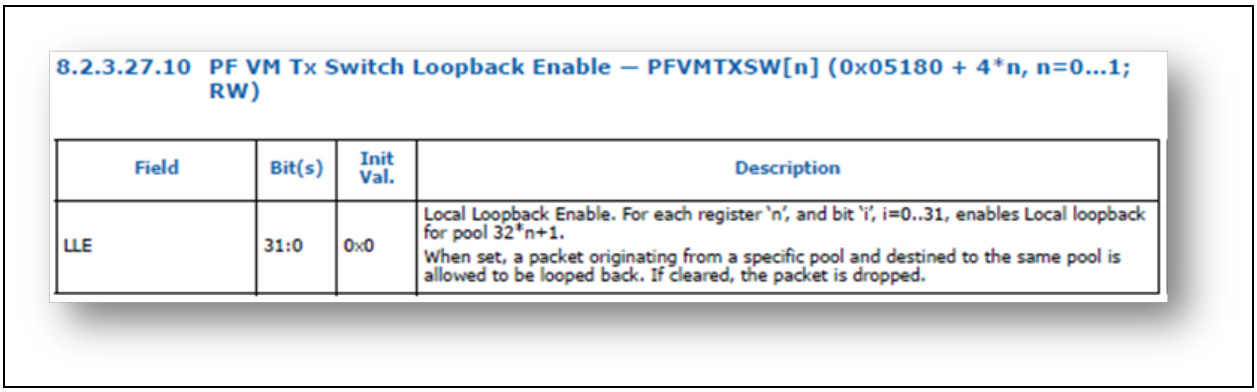
Field	Bit(s)	Init Val.	Description
SSVPC	31:0	0x0	Switch Security Violation Packet Count. This register counts Tx packets dropped due to switch security violations such as SA or VLAN anti-spoof filtering or a packet that has (inner) VLAN that contradicts with PFVMVIR register definitions. Valid only in VMDq or IOV mode.

Figure 31. Switch Security Violation Packet Count definition from the Datasheet

7.4 Local Loopback

This is provided the capability for an individual VF to be able to send traffic and have it loop-backed to itself. This capability might be useful for debug.

To enable Local Loopback for a VF, enable the appropriate bit within the PF VM Tx Switch Loopback Enable (PFVMTXSW – 0x05180) Register.



8.2.3.27.10 PF VM Tx Switch Loopback Enable — PFVMTXSW[n] (0x05180 + 4*n, n=0...1; RW)

Field	Bit(s)	Init Val.	Description
LLE	31:0	0x0	Local Loopback Enable. For each register 'n', and bit 'i', i=0..31, enables Local loopback for pool 32*n+1. When set, a packet originating from a specific pool and destined to the same pool is allowed to be looped back. If cleared, the packet is dropped.

Figure 32. PF VM Tx Switch Loopback Enable definition from the Datasheet

8 Additional Materials

There are documents and materials available to someone investigating SR-IOV. For a complete list, see the *Intel® Ethernet SR-IOV Toolkit Overview* at http://developer.intel.com/products/ethernet/index.htm?iid=nc+ethernet.[2]

Intel® VT-d Information:

http://www.intel.com/technology/itj/2006/v10i3/2-io/7-conclusion.htm

Intel Virtualization for Connectivity:

http://www.intel.com/network/connectivity/solutions/virtualization.htm

[2] On the Developer picklist: select the device, then look for the datasheet.