VFD

VFD 的全称是Virtual Function Daemon。在网络虚拟化（NFV）环境下，VFD被用来配置网卡上的虚拟端口。 VFD 由AT &T 公司开发，具有开源软件许可Apache License, Version 2.0， 2016年4月在GitHub上发布。（https://github.com/att/vfd）

VFD的相关背景

要说明VFD的由来，得先回顾一下网络功能虚拟化（NFV）的背景。

在经典的NFV架构中，虚拟交换机(vSwitch)是整个架构的核心，它同时和虚拟机上的虚拟网卡以及主机上的物理网卡相连，成为数据面（data plane）的中间枢纽，同时接受来自控制面（control plane）对包流表的配置。

经典架构的优势是，它非常的灵活，vSwitch是纯软件实现，不依赖于具体硬件，可以随意创建虚拟端口，轻松实现类似OpenFlow的标准协议用来配置流表，同时基于virtio的VM可以随意的在不同平台上迁移。但是经典架构的最大问题则是性能，网络包从服务器外部进入虚拟机需要经过主机端的网卡驱动，vSwitch，然后通过共享内存进入虚拟机virtio驱动，最终到达应用程序，其间需要经过多次的内存copy，而性能问题已经是目前商用NFV不可忽视的一个问题。

针对NFV的性能提升，可以分为主机端和虚拟机端两部分。在虚拟机中，可以用DPDK的用户态的virtio PMD来做加速，内核协议栈被绕过，并减少了数据从内核空间到用户空间的复制。

而在主机端，目前比较流行的是两种方法:

第一种方法是通过DPDK对vSwitch 进行加速（如：OVS-DPDK是最为典型的开源实现），其原理与以上虚拟机中的方法一致。这种方法的意义是，即提升了整体性能又不失灵活性。但是这也无法避免在将数据包传入虚拟机中时，vSwitch需要将整个包的内容copy到Guest Memory这一过程，在需要大吞吐量的情况下，这仍然会是个问题。

第二种方法则是基于SR-IOV的硬件加速，虚拟机中的应用可以通过SR-IOV直接访问网卡上的数据，从而达到极致的传输性能。基于SR-IOV的NFV架构如下图：

由于vSwitch被绕过，包的传输路由完全依赖于对网卡本身所支持的分流和过滤功能的配置，因此路由功能简单是这种方法无法避免的缺点之一。另外，由于目前SR-IOV在虚拟机端不像virtio有通用的驱动，每种网卡都需要有各自的专有VF驱动，这也使得VM的迁移成为问题，不过相关标准正在制定之中，相信在不久的将来，这一问题可以得到解决。

为什么需要VFD?

VFD主要为第二种方法服务。

部署基于SR-IOV的NFV时会遇到的一个比较现实的问题：如何高效的配置这些虚拟端口。（例如，添加MAC地址, 设置VLAN过滤, 设置QoS带宽，MPLS插入/剥离等等）? 自然而然我们会想到应该有这样的一个工具，它一方面能够提供标准的接口接受用户发送的命令，同时又能够引导不同网卡的驱动程序来完成这些任务。

与配置vSwitch上的纯软件实现的虚拟端口不同，配置基于SR-IOV的虚拟端口需要通过网卡驱动的帮助。支持SR-IOV的网卡通常会有PF驱动和VF驱动。而VF驱动运行在虚拟机里通常是被限制直接对硬件直接进行设置的，所以需要由PF驱动来完成，基本上所有支持SR-IOV的网卡都会有Linux内核PF驱动，同时Linux提供了一些网络接口管理工具 (如iproute2，net-tools等等)，这些工具定义了配置VF的标准接口，只要内核网卡驱动实现这些接口，就能够通过简单的命令行来完成配置任务。然而随着网卡虚拟端口所提供的功能日益增多，可配置的内容也越来越多，越来越复杂，这就对配置工具提出了更易于扩展，更易于更新的要求。

基于内核驱动来配置虚拟端口的方式受限于Linux内核的网络软件框架，更新成本相对较高，更新周期也相对较长。而随着DPDK功能不断完善，将DPDK作为PF端管理入口来配置VF端口已变得可行。由于DPDK是用户态以及发布周期短，基于DPDK的配置工具会变得更为灵活，更能够适应需求不断变化的环境, 而VFD便是其中的一次尝试。

VFD的应用场景

 下图描述了VFD的应用场景。

在一台服务器上，某物理网卡PF由DPDK来管理。

由PF分出VF通过SR-IOV给VM1和VM2，在VM1和VM2上，可以根据所部署的应用特征，选择使用内核驱动（通常在需要用到内核协议栈的情况下）还是直接由DPDK来驱动VF。在这里，VFD是作为一个DPDK的应用部署在主机端，VFD可以同过网卡的DPDK PF驱动来对虚拟机上的VF进行配置。

当然，我们前面也提到了SR-IOV的局限性，有时候vSwtich并不能被完全绕开。好在使用SR-IOV并不限制vSwitch的部署，我们可以将两者结合在一起，将由SR-IOV分出的另一部分VF可以作为vSwitch的外部接口使用，我们看到一块物理网卡可以做到同时支持基于SR-IOV的NFV和基于vSwitch的NFV，而对于所有VF的管理, VFD在这里起到了关键的作用。

VFD的软件结构

VFD由一个后台程序(Daemon)和若干前台命令行工具组成，其中最重要的工具是iplex，对于VF的配置主要通过它来实现。

iplex是一个由python实现的小程序，它能够解析命令行然后转换成JSON格式的命令包,并通过Linux有名管道发送给后台程序，具体实现可参见源码：

https://github.com/att/vfd/blob/master/src/system/iplex

 后台程序由C语言编写，它真正实现了VF的管理和配置，当然没有DPDK的支持，这一切也无法完成，所有对DPDK的封装在sriov.c这个文件中实现。

https://github.com/att/vfd/blob/master/src/vfd/sriov.c

在接收到来自有名管道的命令后，后台程序会根据要求去调用相应的DPDK API。

VFD还有一个重要的特性是对OpenStack的支持，VFD可以作为VF Agent来兼容nova, neutron, heat。具体可参照文档

https://github.com/att/vfd/tree/master/virt/openstack-kilo-mos

VFD的具体功能

在GitHub上已发布的VFD所提供的功能相对比较简单，而且只支持Intel 82599网卡，而在最新发布的DPDK17.02中，我们看到不仅加入了Intel i40e 网卡对VFD的支持，而且比原来82599还扩充了更多的功能，以下列出了一些功能（具体请查看DPDK 17.02的release notes）

· MAC 地址添加/修改/删除

·  单播/多播/广播的使能

·  VLAN 过滤的设置

·  VLAN RX 剥除/ TX插入

·  Anti-spoofing

·  TX 环回使能

·  收发包数据统计/清除

·  Rate Limit 设置

另外在正在卡发的DPDK17.05中，我们还会看到更多新的功能（QoS，MPLS，QinQ等等），而这些功能都会在VFD的后续版本中被集成进去。

VFD的Roadmap

VFD是由AT &T 发起和维护的项目，目前并没有公开的Roadmap，但我们仍然可以在GitHub上看到一些关于VFD2.0的计划。

VFD2.0最大的亮点是引入了对QoS的支持，用户可以创建最多4个TC(Traffic Class)，对于每个TC，可以设置其优先级和带宽，并可将VF上RX/TX队列分配给它。

同时VFD2.0也会提供更为直观的系统管理工具和更为细化的系统配置方法，具体可参考以下文档：

https://github.com/att/vfd/blob/master/doc/qos/vfd\_features\_1.md

VFD对于DPDK的意义

VFD作一个配置工具，其本身还是相对比较简单的，但是他对DPDK的意义却非同一般，传统意义上，DPDK是用来为数据面加速的，而VFD则是将DPDK作为控制面的后台引擎部署在NFV中，这是前所未有的。应该说，在NFV这个大舞台中， VFD给DPDK带来了一个新的角色。

Intel 82599 ixgbe & ixgbevf CNA 卡驱动分析02——VF/PF/MailBox

Physical Function Driver：

     PF 驱动是一个专门管理SR-IOV设备全局功能驱动，而且还要配置相关共享资源。PF 驱动 随着Hypervisor 的不同而不同，一般需要具有比普通虚拟机更高的权限才能对其进行操作。PF驱动包含了所有传统驱动的功能，使得Hypervisor能够访问设备I/O资源。也可以通过调用PF驱动执行相关操作从而影响整个设备。PF驱动必须在VF驱动之前加载，而且需要等VF驱动卸载之后才能卸载。

     Intel SR-IOV 驱动包含了所有 Intel 以太网卡的功能，并且还有下面使用SR-IOV时拥有的特殊功能：

     ·给每个VF生成一个MAC 地址

     ·通过信箱系统与VF驱动通信：

          ·通过VF驱动配置VLAN过滤器

          ·通过VF驱动配置多播地址

          ·通过VF驱动配置最大包长

          ·处理VF驱动资源复位请求

Virutal Function Driver

     标准设备驱动（驱动不会意识到自己所处的虚拟化环境）期望获知如何控制设备和设备是如何工作的。在虚拟化环境下，一个标准的驱动一般与一个软件间接层进行交互，这个软件间接层模拟了底层的物理硬件设备。大多数情况下，该驱动不会意识到这个间接层的存在。

     通过直接赋值，我们期望改变本地共享设备的功能。VF 接口并没有包含完整的PCIe控制功能，而且一般不能直接控制共享设备和资源，比如设置以太网连接速率。这时候有必要告知VF 驱动，使其意识到自己所处的虚拟环境。当VF驱动意识到自己所处的虚拟化环境之后，就能够直接与硬件进行数据交互。同时这些VF驱动也能够知道这些设备都是共享设备，依赖于PF驱动提供的服务来进行操作，这些操作都具有全局效果，比如初始化和下一级结构的控制。

PF Driver - VF Driver Communication

     设备共享需要一个功能就是VF驱动与PF驱动通信，这样VF驱动就能够请求具有全局功能的操作。这个信道需要具有传递信息和产生中断的能力。

     SR-IOV标准并没有定义这条通信线路的实现机制。Intel 选择了通过使用一系列硬件信箱和门铃寄存器来为每个VF实现这个通信信道。

     Intel选择使用一件信箱的根本原因是为了确保实现一个独立于Hypervisor且持久可靠的通信机制。然而， 基于软件的信道也能够用来实现该通信功能，这个功能需要由Hypervisor子系统提供。这些信道的实现各个生产商并不相同，需要客户自己启动这些支持。硬件信道则不同，总是能够工作，除非被Hypervisor禁用。现在看来，并不是所用Hypervisor生产商都提供基于软件的消息机制，因此Intel 相关度去哦那个提供了利用硬件信道的API。

Example Receive Flow

     一个包是如何被一个虚拟机接收和发送的：

第一、二步：包到达，被送往L2 进行分类和交换

第三步：       根据目的MAC地址进行分类，这时候，改包与 缓冲池1匹配

第四步：       网卡发起DMA操作，将一个包传递给一个VM

第五步：       DMA操作到达Intel 芯片集，在这里VT-d（由Hypervisor 配置）进行DMA操作所需的地址翻译；结果使得该包

                    被直接送入到虚拟机的VF驱动缓冲里面

第六步：       网卡发起了MSI-X中断，表明该接收操作已经完成。该中断由Hypervisor接收。

第七步：       Hypervisor向虚拟机注入一个虚拟中断表明传输已经结束，这时候虚拟机的VF驱动就开始处理该包。

Mailbox Communication System

     有时候VF驱动必须与PF驱动通信，以便完成一些工作，这些工作在VF提供的PCI资源基础上是无法完成的。

     例如：当VF驱动想要定义一个VLAN过滤器。这个功能在VF中并没有暴露，因此VF驱动也不能直接配置VLAN过滤器。

     VF驱动能够代替VF 向PF驱动发起这种类型的配置请求。

Virtual Function Mailbox

     当一个虚拟机获得访问VF的权限后，VF 资源中的信箱功能就会暴露给该虚拟机。这个功能相当简单，这是一系列的缓冲区，可以从这些缓冲区中读取或者写入信息，并增加一个寄存器（VFMailbox）用于提供PF和VF之间的同步功能。

     VF和PF都可以访问该缓冲区，可以通过这个缓冲区来传递信息。因为信箱缓冲和VFMailbox寄存器是VF资源的一部分，因此这些资源对于特定的VF来说是独占的。这意味着针对特定VF的VF驱动不能对分配给其他虚拟机的信箱进行写入。

     更多的关于信箱缓冲的信息可以参见：Intel ®82559 10Gigabit Ethernet Controller Datasheet。

Physical Function Mailbox

     PF 驱动能够访问所有的VF信箱，通过VF信箱的信箱内（VFMBMEM：VF Mailbox Mailbox Memory）存和PF信箱(Physical Function Mailbox)寄存器阵列.

     当一个VF驱动向VFMBMEM缓冲写入信息并在VFMailbox寄存器中设置适当的比特位之后，就向给PF产生一个中断。PF驱动获取这些信息，并给予这条消息进行回复。

     这时候，PF驱动会使用信箱发送一个异步消息给VF，这就是IXGBE\_PF\_CONTORL\_MSG 消息。

Virtual Function Driver

     Intel VF驱动实例代码是标准 Intel ixgbe 10 Gigabit Ethernet 驱动的一个修改后的版本。通过设备ID来加载。Intel VF有一个设备ID表明它们是一个VF， 这样祥光的驱动就可以被加载。

     Intel VF 驱动可以被分割为三个部分：

     ·操作系统界面——虚拟机操作系统可以通过该界面调用各种API

     ·I/O操作——使用SR-IOV 功能来进行I/O操作，而避免Hypervisor的干预

     ·配置任务——配置像VLAN过滤器等需要与PF驱动进行通信的任务

注：PF & VF 有时候只实际的PCIe 物理功能模块，有时候就是泛指，注意区别

make it simple, make it happen

**VFD as SRIOV Policy Manager Tests**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#vfd-as-sriov-policy-manager-tests)

**VFD** is SRIOV Policy Manager (daemon) running on the host allowing configuration not supported by kernel NIC driver, supports ixgbe and i40e drivers’ NIC. Run on the host for policy decisions w.r.t. what a VF can and cannot do to the PF. Only the DPDK PF would provide a callback to implement these features, the normal kernel drivers would not have the callback so would not support the features. Allow passing information to application controlling PF when VF message box event received such as those listed below, so action could be taken based on host policy. Stop VM1 from asking for something that compromises VM2.

Multiple purposes:

1. set VF MAC anti-spoofing
2. set VF VLAN anti-spoofing
3. set TX loopback
4. set VF unicast promiscuous mode
5. set VF multicast promiscuous mode
6. set VF MTU
7. get/reset VF stats
8. set VF MAC address
9. set VF VLAN stripping
10. VF VLAN insertion
11. set VF broadcast mode
12. set VF VLAN tag
13. set VF VLAN filter
14. Set/reset the queue drop enable bit for all pools(only ixgbe support)
15. **Set/reset the enable drop bit in the split receive control register**

(only ixgbe support)

**VFD** also includes VF to PF mailbox message management by APP. When PF receives mailbox messages from VF, PF should call the callback provided by APP to know if they’re permitted to be processed.

**53.1. Prerequisites**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#prerequisites)

1. Host PF in DPDK driver. Create 2 VFs from 1 PF with dpdk driver,take Niantic for example:
2. ./tools/dpdk-devbind.py -b igb\_uio 81:00.0
3. echo 2 >/sys/bus/pci/devices/0000:81:00.0/max\_vfs
4. Detach VFs from the host:
5. rmmod ixgbevf
6. Passthrough VF 81:10.0 to vm0 and passthrough VF 81:10.2 to vm1, start vm0 and vm1
7. Login vm0 and vm1, then bind VF0 device to igb\_uio driver.
8. Start testpmd on host and vm0 in chained port topology:
9. ./testpmd -c f -n 4 -- -i --port-topology=chained --txqflags=0

**53.2. Test Case 1: Set VLAN insert for VF from PF**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#test-case-1-set-vlan-insert-for-vf-from-pf)

1. Disable vlan insert for VF0 from PF:
2. testpmd> set vf vlan insert 0 0 0
3. Start VF0 testpmd, set it in mac forwarding mode and enable verbose output
4. Send packet from tester to VF0 without vlan id
5. Stop VF0 testpmd and check VF0 can receive packet without any vlan id
6. Enable vlan insert and insert random vlan id (1~4095) for VF0 from PF:
7. testpmd> set vf vlan insert 0 0 id
8. Start VF0 testpmd
9. Send packet from tester to VF0 without vlan id
10. Stop VF0 testpmd and check VF0 can receive packet with configured vlan id

**53.3. Test Case 2: Set VLAN strip for VF from PF**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#test-case-2-set-vlan-strip-for-vf-from-pf)

1. Disable VLAN strip for all queues for VF0 from PF:
2. testpmd> set vf vlan stripq 0 0 off
3. Start VF0 testpmd, add rx vlan id as random 1~4095, set it in mac forwarding mode and enable verbose output:
4. testpmd> rx\_vlan add id 0
5. Send packet from tester to VF0 with configured vlan id
6. Stop VF0 testpmd and check VF0 can receive packet with configured vlan id
7. Enable VLAN strip for all queues for VF0 from PF:
8. testpmd> set vf vlan stripq 0 0 on
9. Start VF0 testpmd
10. Send packet from tester to VF0 with configured vlan id
11. Stop VF0 testpmd and check VF0 can receive packet without any vlan id
12. Remove vlan id on VF0

**53.4. Test Case 3: Set VLAN antispoof for VF from PF**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#test-case-3-set-vlan-antispoof-for-vf-from-pf)

1. Disable vlan filter and strip from PF:
2. testpmd> vlan set filter off 0
3. testpmd> vlan set strip off 0
4. Add a random 1~4095 vlan id to set filter from PF for VF:
5. testpmd> rx\_vlan add id port 0 vf 1
6. Disable vlan antispoof for VF from PF:
7. testpmd> set vf vlan antispoof 0 0 off
8. Disable vlan filter and strip on VF0
9. Start testpmd on VF0, set it in mac forwarding mode and enable:
10. verbose output
11. Send packets with matching/non-matching/no vlan id on tester port
12. Stop VF0 testpmd and check VF0 can receive and transmit packets with matching/non-matching/no vlan id
13. Enable mac antispoof and vlan antispoof for vf from PF:
14. testpmd> set vf mac antispoof 0 0 on
15. testpmd> set vf vlan antispoof 0 0 on
16. Start VF0 testpmd
17. Send packets with matching/non-matching/no vlan id on tester port
18. Stop VF0 testpmd and check VF0 can receive all but only transmit packet with matching vlan id

**53.5. Test Case 4: Set mac antispoof for VF from PF**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#test-case-4-set-mac-antispoof-for-vf-from-pf)

1. Add fake mac and use fake mac instead of transmitted mac in the macswap mode, so default is non-matching SA:
2. .addr\_bytes = {0x00, 0x11, 0x22, 0x33, 0x44, 0x55}
3. Disable VF0 mac antispoof from PF:
4. testpmd> set vf mac antispoof 0 0 off
5. Start testpmd on VF0, set it in macswap forwarding mode and enable verbose output:
6. testpmd> set fwd macswap
7. Send packet from tester to VF0 with correct SA, but code has changed to use fake SA
8. Stop VF0 testpmd and check VF0 can receive then transmit packet
9. Enable VF0 mac antispoof from PF:
10. testpmd> set vf mac antispoof 0 0 on
11. Start VF0 testpmd
12. Send packet from tester to VF0 with correct SA, but code has changed to use fake SA
13. Stop VF0 testpmd and check VF0 can receive packet but can’t transmit packet
14. Recover original code

**53.6. Test Case 5: Set the MAC address for VF from PF**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#test-case-5-set-the-mac-address-for-vf-from-pf)

1. Set VF0 different MAC address from PF, such as A2:22:33:44:55:66
2. testpmd> set vf mac addr 0 0 A2:22:33:44:55:66
3. Stop VF0 testpmd and restart VF0 testpmd, check VF0 address is configured address A2:22:33:44:55:66
4. Set testpmd in mac forwarding mode and enable verbose output
5. Send packet from tester to VF0 configured address
6. Stop VF0 testpmd and check VF0 can receive packet

**53.7. Test Case 6: Enable/disable tx loopback**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#test-case-6-enable-disable-tx-loopback)

1. Disable tx loopback for VF0 from PF:
2. testpmd> set tx loopback 0 off
3. Set VF0 in rxonly forwarding mode and start testpmd
4. tcpdump on the tester port
5. Send 10 packets from VF1 to VF0
6. Stop VF0 testpmd, check VF0 can’t receive any packet but tester port could capture packet
7. Enable tx loopback for VF0 from PF:
8. testpmd> set tx loopback 0 on
9. Start VF0 testpmd
10. Send packet from VF1 to VF0
11. Stop VF0 testpmd, check VF0 can receive packet,but tester port can’t capture packet

**53.8. Test Case 7: Set drop enable bit for all queues**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#test-case-7-set-drop-enable-bit-for-all-queues)

1. Bind VF1 device to igb\_uio driver and start testpmd in chained port topology
2. Disable drop enable bit for all queues from PF:
3. testpmd> set all queues drop 0 off
4. Only start VF1 to capture packet, set it in rxonly forwarding mode and enable verbose output
5. Send 200 packets to VF0, make VF0 queue full of packets
6. Send 20 packets to VF1
7. Stop VF1 testpmd and check VF1 can’t receive packet
8. Enable drop enable bit for all queues from PF:
9. testpmd> set all queues drop 0 on
10. Start VF1 testpmd
11. Stop VF1 testpmd and check VF1 can receive original queue buffer 20 packets
12. Start VF1 testpmd
13. Send 20 packets to VF1
14. Stop VF1 testpmd and check VF1 can receive 20 packets

**53.9. Test Case 8: Set split drop enable bit for VF from PF**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#test-case-8-set-split-drop-enable-bit-for-vf-from-pf)

1. Disable split drop enable bit for VF0 from PF:
2. testpmd> set vf split drop 0 0 off
3. Set VF0 and host in rxonly forwarding mode and start testpmd
4. Send a burst of 20000 packets to VF0 and check PF and VF0 can receive all packets
5. Enable split drop enable bit for VF0 from PF:
6. testpmd> set vf split drop 0 0 on
7. Send a burst of 20000 packets to VF0 and check some packets dropped on PF and VF0

**53.10. Test Case 9: Get/Reset stats for VF from PF**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#test-case-9-get-reset-stats-for-vf-from-pf)

1. Add testpmd and some print code in the rte\_pmd\_i40e\_set\_vf\_vlan\_filter() function(drivers/net/i40e/i40e\_ethdev.c) to start test, rebuild the code
2. Get stats output for VF0 from PF, and check RX/TX packets is 0:
3. testpmd> get vf stats 0 0
4. Set VF0 in mac forwarding mode and start testpmd
5. Send 10 packets to VF0 and check VF0 can receive 10 packets
6. Get stats for VF0 from PF, and check RX/TX packets is 10
7. Reset stats for VF0 from PF, and check PF and VF0 RX/TX packets is 0:
8. testpmd> reset vf stats 0 0
9. testpmd> get vf stats 0 0

**53.11. Test Case 10: enhancement to identify VF MTU change**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#test-case-10-enhancement-to-identify-vf-mtu-change)

1. Set VF0 in mac forwarding mode and start testpmd
2. Default mtu size is 1500, send one packet with length bigger than default mtu size, such as 2000 from tester,check VF0 can receive but can’t transmit packet
3. Set VF0 mtu size as 3000, but need to stop then restart port to active mtu:
4. testpmd> port stop all
5. testpmd> port config mtu 0 3000
6. testpmd> port start all
7. testpmd> start
8. Send one packet with length 2000 from tester,check VF0 can receive and transmit packet
9. Send one packet with length bigger than configured mtu size, such as 5000 from tester, check VF0 can receive but can’t transmit packet

**53.12. Test Case 11: Enable/disable vlan tag forwarding to VSIs**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#test-case-11-enable-disable-vlan-tag-forwarding-to-vsis)

1. Disable VLAN tag for VF0 from PF:
2. testpmd> set vf vlan tag 0 0 off
3. Start VF0 testpmd, add rx vlan id as random 1~4095, set it in mac forwarding mode and enable verbose output
4. Send packet from tester to VF0 with vlan tag(vlan id should same as rx\_vlan)
5. Stop VF0 testpmd and check VF0 can’t receive vlan tag packet
6. Enable VLAN tag for VF0 from PF:
7. testpmd> set vf vlan tag 0 0 on
8. Start VF0 testpmd
9. Send packet from tester to VF0 with vlan tag(vlan id should same as rx\_vlan)
10. Stop VF0 testpmd and check VF0 can receive vlan tag packet
11. Remove vlan id on VF0

**53.13. Test Case 12: Broadcast mode**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#test-case-12-broadcast-mode)

1. Start testpmd on VF0, set it in rxonly mode and enable verbose output
2. Disable broadcast mode for VF0 from PF:
3. testpmd>set vf broadcast 0 0 off
4. Send packets from tester with broadcast address,ff:ff:ff:ff:ff:ff, and check VF0 can not receive the packet
5. Enable broadcast mode for VF0 from PF:
6. testpmd>set vf broadcast 0 0 on
7. Send packets from tester with broadcast address,ff:ff:ff:ff:ff:ff, and check VF0 can receive the packet

**53.14. Test Case 13: Multicast mode**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#test-case-13-multicast-mode)

1. Start testpmd on VF0, set it in rxonly mode and enable verbose output
2. Disable promisc and multicast mode for VF0 from PF:
3. testpmd>set vf promisc 0 0 off
4. testpmd>set vf allmulti 0 0 off
5. Send packet from tester to VF0 with multicast MAC, and check VF0 can not receive the packet
6. Enable multicast mode for VF0 from PF:
7. testpmd>set vf allmulti 0 0 on
8. Send packet from tester to VF0 with multicast MAC, and check VF0 can receive the packet

**53.15. Test Case 14: Promisc mode**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#test-case-14-promisc-mode)

1. Start testpmd on VF0, set it in rxonly mode and enable verbose output
2. Disable promisc mode for VF from PF:
3. testpmd>set vf promisc 0 0 off
4. Send packet from tester to VF0 with random MAC, and check VF0 can not receive the packet
5. Send packet from tester to VF0 with correct MAC, and check VF0 can receive the packet
6. Enable promisc mode for VF from PF:
7. testpmd>set vf promisc 0 0 on
8. Send packet from tester to VF0 with random MAC, and the packet can be received by VF0
9. Send packet from tester to VF0 with correct MAC, and the packet can be received by VF0

**53.16. Test Case 14: Set Vlan filter for VF from PF**[**¶**](http://dpdk.org/doc/dts/test_plans/vf_daemon_test_plan.html?highlight=vfd#test-case-14-set-vlan-filter-for-vf-from-pf)

1. Start VF0 testpmd, set it in rxonly forwarding mode, enable verbose output
2. Send packet without vlan id to random MAC, check VF0 can receive packet
3. Add vlan filter id as random 1~4095 for VF0 from PF:
4. testpmd> rx\_vlan add id port 0 vf 1
5. Send packet from tester to VF0 with wrong vlan id to random MAC, check VF0 can’t receive packet
6. Send packet from tester to VF0 with configured vlan id to random MAC, check VF0 can receive packet
7. Remove vlan filter id for VF0 from PF:
8. testpmd> rx\_vlan rm id port 0 vf 1
9. Send packet from tester to VF0 with wrong vlan id to random MAC, check VF0 can receive packet
10. Send packet from tester to VF0 with configured vlan id to random MAC, check VF0 can receive packet
11. Send packet without vlan id to random MAC, check VF0 can receive packet