⚠ This model is only applicable when the SmartNIC is operating ECPF ownership mode.

BlueField SmartNIC uses netdev representors to map each one of the host side physical and virtual functions:

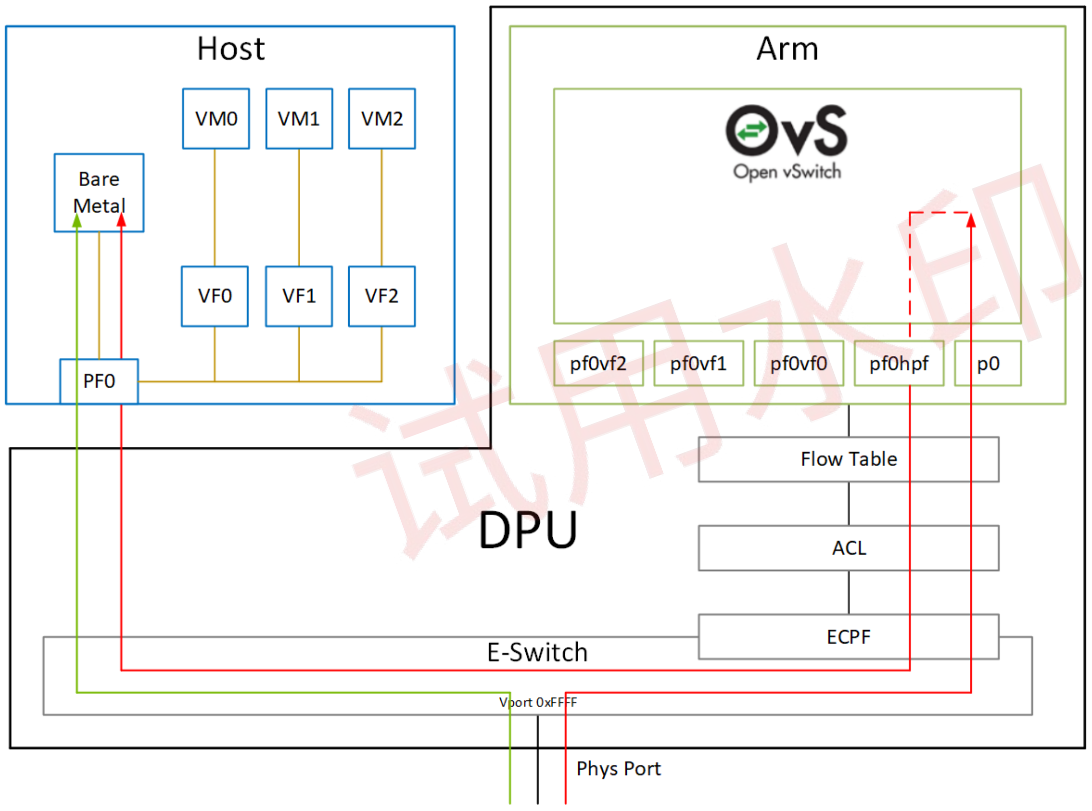
1. Serve as the tunnel to pass traffic for the virtual switch or application running on the Arm cores to the relevant PF or VF on the Arm side.
2. Serve as the channel to configure the embedded switch with rules to the corresponding represented function.

Those representors are used as the virtual ports being connected to OVS or any other virtual switch running on the Arm cores.

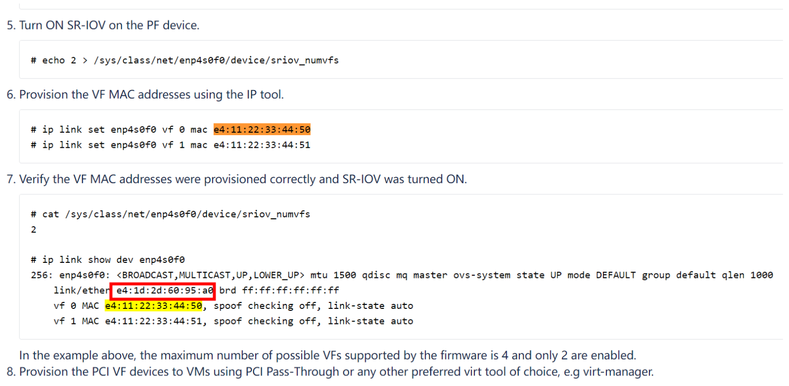
When in ECPF ownership mode, we see 2 representors for each one of the SmartNIC's network ports: one for the uplink, and another one for the host side PF (the PF representor created even if the PF is not probed on the host side). For each one of the VFs created on the host side a corresponding representor would be created on the Arm side. The naming convention for the representors is as follows:

- Uplink representors: p<port_number>
- PF representors: pf<port_number>hpf
- VF representors: pf<port_number>vf<function_number>

The diagram below shows the mapping of between the PCI functions exposed on the host side and the representors. For the sake of simplicity, we show a single port model (duplicated for the second port).



The red arrow demonstrates a packet flow through the representors, while the green arrow demonstrates the packet flow when steering rules are offloaded to the embedded switch. More details on that are available in the switch offload section.



5. Turn ON SR-IOV on the PF device.

```
# echo 2 > /sys/class/net/enp4s0f0/device/sriov_numvfs
```

6. Provision the VF MAC addresses using the IP tool.

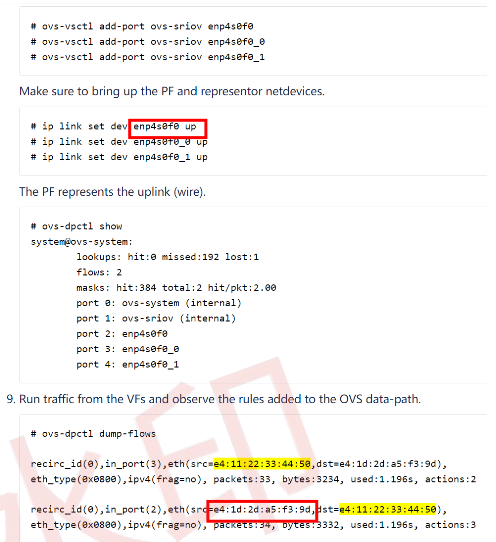
```
# ip link set enp4s0f0 vf 0 mac e4:11:22:33:44:50
# ip link set enp4s0f0 vf 1 mac e4:11:22:33:44:51
```

7. Verify the VF MAC addresses were provisioned correctly and SR-IOV was turned ON.

```
# cat /sys/class/net/enp4s0f0/device/sriov_numvfs
2

# ip link show dev enp4s0f0
256: enp4s0f0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq master ovs-system state UP mode DEFAULT group default qlen 1000
    link/ether e4:1d:2d:60:95:a0 brd ff:ff:ff:ff:ff:ff
    vf 0 MAC e4:11:22:33:44:50, spoof checking off, link-state auto
    vf 1 MAC e4:11:22:33:44:51, spoof checking off, link-state auto
```

In the example above, the maximum number of possible VFs supported by the firmware is 4 and only 2 are enabled.

8. Provision the PCI VF devices to VMs using PCI Pass-Through or any other preferred virt tool of choice, e.g virt-manager.



```
# ovs-vsctl add-port ovs-sriov enp4s0f0
# ovs-vsctl add-port ovs-sriov enp4s0f0_0
# ovs-vsctl add-port ovs-sriov enp4s0f0_1
```

Make sure to bring up the PF and representor netdevices.

```
# ip link set dev enp4s0f0 up
# ip link set dev enp4s0f0_0 up
# ip link set dev enp4s0f0_1 up
```

The PF represents the uplink (wire).

```
# ovs-dpctl show
system@ovs-system:
        lookups: hit:0 missed:192 lost:1
        flows: 2
        masks: hit:384 total:2 hit/pkt:2.00
        port 0: ovs-system (internal)
        port 1: ovs-sriov (internal)
        port 2: enp4s0f0
        port 3: enp4s0f0_0
        port 4: enp4s0f0_1
```

9. Run traffic from the VFs and observe the rules added to the OVS data-path.

```
# ovs-dpctl dump-flows

recirc_id(0),in_port(3),eth(src=e4:11:22:33:44:50,dst=e4:1d:2d:a5:f3:9d),
eth_type(0x0800),ipv4(frag=no), packets:33, bytes:3234, used:1.196s, actions:2

recirc_id(0),in_port(2),eth(src=e4:1d:2d:a5:f3:9d,dst=e4:11:22:33:44:50),
eth_type(0x0800),ipv4(frag=no), packets:34, bytes:3332, used:1.196s, actions:3
```

offload TC SW datapath to e-switch with VF reps

- enable SRIOV, assign VFs to VMs, set e-switch to switchdev mode [..]
- use TC HW offloads to program ingress rules to VF rep → e-switch HW
- typical rule format: <ingress port, matching, action>

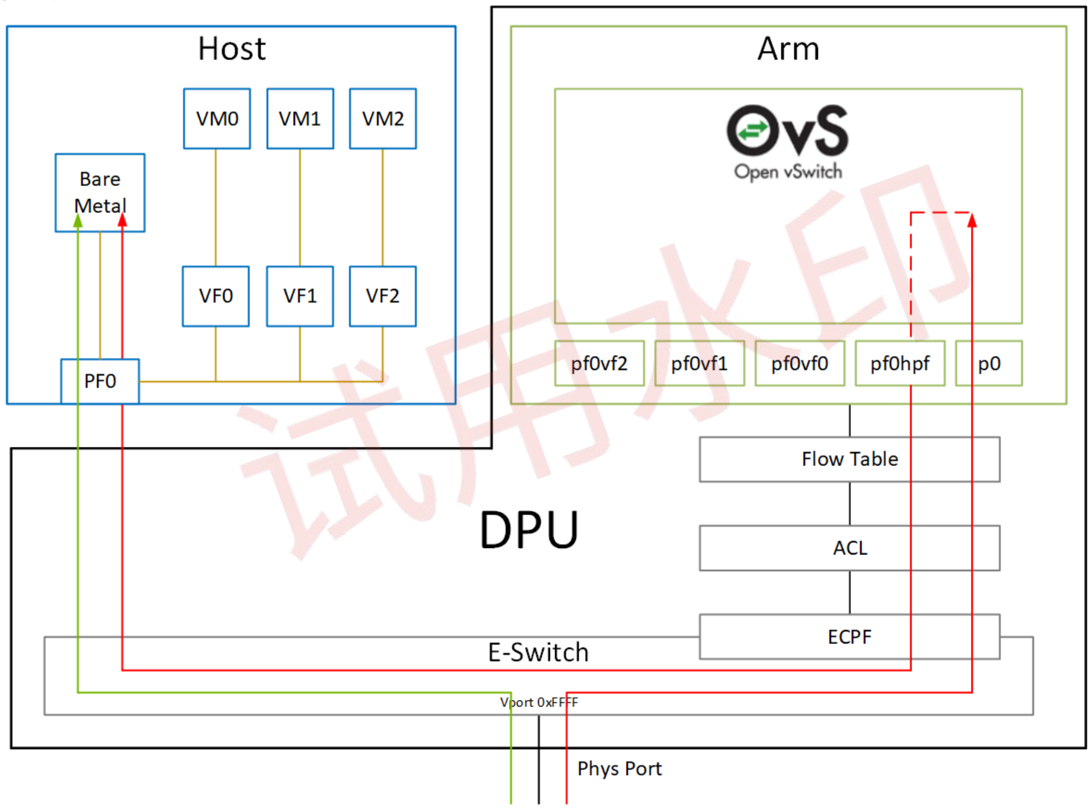
```
# ethtool -K enp4s0f1_0 hw-tc-offload on
# tc qdisc add dev enp4s0f1_0 ingress

# tc filter add dev enp4s0f1_0 protocol ip parent ffff:
flower skip_sw src_mac e4:11:22:33:44:50 dst_mac
e4:1d:2d:a5:f3:9d action mirred egress redirect dev
enp4s0f1
```

2. Serve as the channel to configure the embedded switch with rules to the corresponding represented function.

Those representors are used as the virtual ports being connected to OVS or any other virtual switch running on the Arm cores.
When in ECPF ownership mode, we see 2 representors for each one of the SmartNIC’s network ports: one for the uplink, and another one for the host side PF (the PF representor created even if the PF is not probed on the host side). For each one of the VFs created on the host side a corresponding representor would be created on the Arm side. The naming convention for the representors is as follows:

- Uplink representors: p<port_number>
- PF representors: pf<port_number>hpf
- VF representors: pf<port_number>vf<function_number>

The diagram below shows the mapping of between the PCI functions exposed on the host side and the representors. For the sake of simplicity, we show a single port model (duplicated for the second port).



The red arrow demonstrates a packet flow through the representors, while the green arrow demonstrates the packet flow when steering rules are offloaded to the embedded switch. More details on that are available in the switch offload section.

Virtual Switch on BlueField SmartNIC

ASAP^2 is only supported in ECPF ownership mode.

BlueField supports ASAP^2 technology. It utilizes the representors mentioned in the previous section. BlueField SW package includes OVS installation which already supports ASAP^2. The virtual switch

running on the Arm cores allows us to pass all the traffic to and from the host functions through the Arm cores while performing all the operations supported by OVS. ASAP^2 allows us to offload the datapath by programming the NIC embedded switch and avoiding the need to pass every packet through the Arm cores. The control plain remains the same as working with standard OVS.
The examples will follow a scenario where there were 3 VFs created on the PF of port 0 of the host side.
To enable 3 virtual functions, run the following command on the host side:

```
# echo 3 > /sys/class/net/ens1f0/device/sriov_numvfs
```

This results in the following interfaces appearing on the Arm side:

```
# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: tmfifo_net0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT group default
qlen 1000
    link/ether 00:1a:ca:ff:ff:01 brd ff:ff:ff:ff:ff:ff
5: p0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN mode DEFAULT group default qlen 1000
    link/ether 50:6b:4b:34:a5:0e brd ff:ff:ff:ff:ff:ff
6: pf0hpf: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000
    link/ether 36:ef:20:b3:53:eb brd ff:ff:ff:ff:ff:ff
7: p1: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc mq state DOWN mode DEFAULT group default qlen 1000
    link/ether 50:6b:4b:34:a5:0f brd ff:ff:ff:ff:ff:ff
8: pf1hpf: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000
    link/ether c6:9b:16:6a:1c:7e brd ff:ff:ff:ff:ff:ff
9: pf0vf0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 7a:01:80:55:93:56 brd ff:ff:ff:ff:ff:ff
10: pf0vf1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether e2:ab:51:0c:01:ee brd ff:ff:ff:ff:ff:ff
11: pf0vf2: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 0a:58:2d:d2:f6:73 brd ff:ff:ff:ff:ff:ff
```

The tmfifo_net0 interface is the RShim network interface and it is not part of the datapath between the host and the network.
For this example ignore the interfaces “p1” and “p1hpf” also, which are representors for the second port. This leaves 5 representors:

- p0 – uplink representor
- pf0hpf – host PF representor
- pf0vf<0-2> – host VF representors for virtual functions 0 to 2

The next step is starting the OpenvSwitch and creating a new OVS bridge, we’ll call it “armbr1”:

```
# service openvswitch start
# ovs-vsctl add-br armbr1
```

Add each one of the representors (including the uplink representor) to the bridge:

```
# ovs-vsctl add-port armbr1 p0
# ovs-vsctl add-port armbr1 pf0hpf
# ovs-vsctl add-port armbr1 pf0vf0
# ovs-vsctl add-port armbr1 pf0vf1
# ovs-vsctl add-port armbr1 pf0vf2
```

To verify successful bridging:

```
# ovs-vsctl show
4c1cd894-2bd2-431c-a808-a50705c69798
    Bridge "armbr1"
        Port "pf0vf0"
            Interface "pf0vf0"
        Port "pf0vf1"
            Interface "pf0vf1"
        Port "pf0vf2"
            Interface "pf0vf2"
        Port "pf0hpf"
            Interface "pf0hpf"
        Port "p0"
            Interface "p0"
        Port "armbr1"
            Interface "armbr1"
                type: internal
    ovs_version: "2.11.1
```

The host is now connected to the network.
To enable the connection next time the SmartNIC boots, add the "openvswitch" service to the list of services to be started at boot time. For example, with "systemctl", run "systemctl enable openvswitch".

Verifying Host Connection

When the SmartNIC is connected to another SmartNIC on another machine, manually assign IP addresses with the same subnet to both ends of the connection.

1. Assuming the link is connected to p3p1 on the other host, run:

```
ifconfig p3p1 192.168.200.1/24 up
```

2. On the host which the SmartNIC is connected to, run:

```
ifconfig p4p2 192.168.200.2/24 up
```

3. Have one ping the other. This is an example of the SmartNIC pinging the host:

```
ping 192.168.200.1
```

Enabling OVS HW Offloading

To enable the HW offload run the following commands (restarting OVS is required after enabling the HW offload):

```
# ovs-vsctl set Open_vSwitch . Other_config:hw-offload=true
# systemctl restart openvswitch
```

To show OVS configuration:

```
# ovs-dpctl show
system@ovs-system:
  lookups: hit:0 missed:0 lost:0
  flows: 0
  masks: hit:0 total:0 hit/pkt:0.00
  port 0: ovs-system (internal)
  port 1: armbr1 (internal)
  port 2: p0
  port 3: pf0hpf
  port 4: pf0vf0
  port 5: pf0vf1
  port 6: pf0vf2
```

At this point OVS would automatically try to offload all the rules.
To see all the rules that are added to the OVS datapath: