Intel® Ethernet Controller E810
Dynamic Device Personalization (DDP) for Telecommunications

Technology Guide

Ethernet Products Group (EPG)

December 2020

Revision 2.2
618651-003

Revision History

Revision	Date	Comments
2.2	December 18, 2020	Updates include the following: • Update OS-Default Package version from 1.3.16.0 to 1.3.20.0. • Updated DDP Comms Package version from 1.3.20.0 to 1.3.24.0. • Updated E810 firmware version from 1.5.1.5/1.5.1.9 to 1.5.3.7. • Updated E810 NVM version from 2.10/2.12 to 2.30/2.32. • Updated DPDK version from 20.08 to 20.11. • Updated Section 1.2, "Software/Firmware Requirements". • Updated Table 2, "Supported Protocols". • Updated Table 6, "LTE/5G Capabilities Summary". • Updated Section 3.1, "Comms Package Download". • Updated Section 3.2.1, "Option 1: *ice Linux Base Driver*". • Updated Section 3.2.2, "Option 2: DPDK Driver Only". • Updated Section 3.3, "Loading Comms Package to a Specific 800 Series Device". • Updated Section 4.1, "5G UPF". • Updated Section 4.1.1, "RSS for Packet Steering Based on UE IP Address". • Added Section 4.1.2, "RSS for Packet Steering Based on Flow". • Updated Section 4.2, "Limitations". • Updated Section 4.4.4, "Actions". • Updated Section 4.6, "RSS on Inner Source IPv4 Address". • Added Section 4.7, "RSS on Outer Flow". • Added Section 4.8, "RSS on Inner Flow". • Updated Table 4.10, "Route to Queue Group Based on QFI". • Added Section 4.12, "IPv6 Prefixes". • Updated Section 5.3, "Programming Switch Filters on DCF". • Added Section 6.0, "Intel® Ethernet 800 Series Features"
2.1	September 24, 2020	Updates include the following: • Update OS-Default Package version from 1.3.11.0 to 1.3.16.0. • Updated DDP Comms Package version from 1.3.17.0 to 1.3.20.0. • Updated E810 firmware version from 1.4.1.1 to 1.5.1.5/1.5.1.9. • Updated E810 NVM version from 1.40 to 2.10/2.12. • Updated DPDK version from 20.05 to 20.08. • Updated Section 1.0, "Introduction". • Updated Table 2, "Supported Protocols", • Updated Section 2.1.1, "GTP". • Added Section 2.1.6, "MPLS". • Updated Table 3, "Supported Protocols in the Comms DDP Package", • Updated Table 6, "LTE/5G Capabilities Summary", • Updated Section 3.1, "Comms Package Download". • Updated Section 3.2.1, "Option 1: *ice Linux Base Driver*". • Updated Step 2 in Section 3.3, "Loading Comms Package to a Specific 800 Series Device". • Updated Section 4.1, "5G UPF". • Added Section 4.11, "MPLS Protocol Extraction". • Updated Section 5.3, "Programming Switch Filters on DCF".
2.0[1]	August 25, 2020	Initial public release.

1. There are no previous publicly-available versions of this document.

Contents

1.0 Introduction

The Intel® Ethernet 800 Series (800 Series) is the next generation of Intel® Ethernet Controllers and Network Adapters. The 800 Series is designed with an enhanced programmable pipeline, allowing deeper and more diverse protocol header processing. This on-chip capability is called Dynamic Device Personalization (DDP). In the 800 Series, a DDP profile is loaded dynamically on driver load per device.

A general purpose DDP package is automatically installed with all supported 800 Series drivers on Windows, ESX, FreeBSD, and Linux operating systems, including those provided by the Data Plane Development Kit (DPDK). This general-purpose DDP package is known as the OS-default package.

For more information on DDP technology in the 800 Series products and the OS-default package, refer to the *Intel® Ethernet Controller E810 Dynamic Device Personalization (DDP) Technology Guide*, available here: https://cdrdv2.intel.com/v1/dl/getContent/617015.

This document describes an optional DDP package targeted towards the needs of telecommunication (Comms) customers. In addition to the protocols in the OS-default package, the Comms DDP package (v1.3.24.0) provides support for GTP, PPPoE, IPSEC, L2TPv3, PFCP, and MPLS protocols. The Comms DDP package is supported by DPDK 20.11 drivers and the 800 Series *ice* driver on Linux operating systems. The Comms DDP Package can be loaded on all 800 Series devices, or different packages can be selected via serial number per device.

1.1 Terminology

Table 1. Acronyms and Definitions

Acronym	Definition
FD	Intel® Flow Director
GTP	GPRS Tunneling Protocol An IP-based communications protocol.
OEM	Original Equipment Manufacturer
OOT	Out-of-Tree
PPPoE	Point-to-Point Protocol over Ethernet A network protocol for encapsulating PPP inside Ethernet frames.
PTYPE	Packet Type
QFI	Quality of Service Flow Indicator
RSS	Receive Side Scaling
UPF	5G User Plane Function
URLLC	Ultra-Reliable Low-Latency Communication

1.2 Software/Firmware Requirements

The specific DDP package requires certain firmware and DPDK versions and 800 Series firmware/NVM versions. The required DPDK version contains the support of loading the specific Comms DDP package.

- E810 DDP Comms Package: 1.3.24.0
- E810 OS Package file: 1.3.20.0
- E810 *ice* driver: 1.3.2
- E810 *iavf* driver: 4.0.2
- DPDK version: 20.11
- E810 NVM version: 2.30/2.32
- E810 Firmware version: 1.5.3.7

2.0 Comms DDP Package Description

The Comms DDP package is built on top of the existing OS-default package, adding specific support of protocols typically utilized in telecommunications in addition to those provided by the OS-default package.

2.1 New in Comms DDP Package

The new protocols supported in the Comms DDP package are listed in Table 2.

Table 2. Supported Protocols

DDP Package Version	Corresponding Supported DPDK Version	Supported Protocols
1.3.10.0	19.11, 20.02	GTP, PPPoE
1.3.17.0	20.05	GTP, PPPoE, IPSEC, L2TPv3, PFCP
1.3.20.0	20.08	GTP, PPPoE, IPSEC, L2TPv3, PFCP, MPLS
1.3.20.0/1.3.24.0	20.11	GTP, PPPoE, IPSEC, L2TPv3, PFCP, MPLS Added Separate Package Type Group for outer IPv4 and IPv6 tunnel packet

2.1.1 GTP

General Packet Radio System (GPRS) Tunneling Protocol (or GTP) is a main tunneling protocol used on LTE networks. GTP uses UDP as a transport protocol, and the DDP parses GTP packets similar to the way other UDP-based tunnels are parsed (for example, VXLAN and Geneve). The DDP does not report a separate Protocol ID for GTP. Instead the outer UDP protocol includes the UDP header and the variable length GTP header.

To differentiate between the types of GTP headers, the DDP reports separate PTYPEs to allow for custom extraction sequences in the downstream blocks, such as switch, Receive Side Scaling, and Intel® Flow Director (downstream blocks) of the 800 Series device. Two bits packet flags are set to identify if the PDU Session Container Extension Header is present with DL/UL type.

2.1.2 PPPoE

Point-to-Point over Ethernet (PPPoE) is the main protocol for Broadband Network Gateways (BNG). There are two different types of PPPoE headers that are supported by the package.

- PPPoE Discovery stage (PPPoED) supports control type only with Ethernet type of 0x8863.
- PPPoE Session (PPPoES) supports control and data with Ethernet type of 0x8864.

Different packet types are reported for different types of PPPoE to allow for custom extraction sequences in the downstream blocks. The DDP only supports detecting the PPPoE header on the outer Ethernet type field. Tunneled packets are not supported.

2.1.3 IPSEC - ESP/AH

IP Security (IPsec) is an Internet Engineering Task Force (IETF) standard suite of protocols between two communication points across the IP network, providing data authentication, integrity, and confidentiality.

For IPsec packets, the parser (one of the hardware logics in the on-chip processing pipeline — see the *Intel® Ethernet Controller E810 Dynamic Device Personalization (DDP) Technology Guide* for more information) reports unique PTYPES to differentiate the types of security headers and unique protocol IDs for IP Encapsulating Security Payload (ESP) and IP Authentication Header (AH) headers.

2.1.4 L2TPv3

Layer 2 Tunneling Protocol Version 3 (L2TPv3) is an IETF standard related to L2TP that is used for encapsulating multi-protocol Layer 2 communications traffic over IP network.

The parser supports detecting the presence of L2TPv3 Session and Control headers over IP. For both IPv4 and IPv6, a value of 115 identifies the L2TPv3 header. The start of the L2TPv3 header is identified by the L2TPv3 protocol ID. The parser only parses the first four bytes of *Session ID* field.

2.1.5 PFCP

Packet Forwarding Control Protocol (PFPC) is the main control plane — user plane communication protocol for 5G networks.

PFCP packets are similar to GTP Control (GTP-C, which is used for 4G/LTE networks) packets and use UDP over IPv4/IPv6 as transport protocol. PFCP packets are considered as non-tunneled packets. The parser supports parsing two different types of PFCP headers: Node and Session headers.

2.1.6 MPLS

Multi-Protocol Label Switching (MPLS) is a routing technique in Comms networks that forwards the packet from one router to the next based on a short fixed length value known as a "label" rather than long network addresses, avoiding complex lookups in a routing table and speeding traffic flows. When a packet is forwarded to its next hop, the label is sent along with it. In other words, the packets are "labeled" before they are forwarded. MPLS is "multi-protocol" because its techniques are applicable to any network layer protocol.

2.2 Comms DDP Package Protocols and Packet Types

Once the Comms DDP package is successfully loaded, the protocols shown in Table 3 are supported. Shading in green indicates Comms DDP Package-specific, while no color indicates also supported by the OS-default package.

Table 3. Supported Protocols in the Comms DDP Package

Protocols		
MAC	ICMP	NVGRE
ETYPE	ICMPv6	RoCEv2
VLAN	CTRL	GTPv1-C, GTPv1-U + GTP extension headers
IPv4	LLDP	GTPv2-C
IPv6	ARP	PPPoE
TCP	VXLAN-GPE	ESP/AH
UDP	VXLAN (non-GPE)	L2TPv3
SCTP	GRE	PFCP
		MPLS

Packet type (PTYPE) is one of the inputs needed to determine a packet parsing profile for filtering and other processing. The Comms DDP package supports all of the PTYPES supported by the OS-default package as well as additional PTYPES listed in Table 4. For a complete list of OS-default DDP package supported PTYPES, refer to the *Intel® Ethernet Controller E810 Dynamic Device Personalization (DDP) Technology Guide*.

Table 4. Additional Supported Packet Types in the Comms DDP Package

PTYPE	PTYPE Description	PTYPE	PTYPE Description
160	MAC_IPV4_ESP	329	MAC_IPV4_GTPU
161	MAC_IPV6_ESP	330	MAC_IPV6_GTPU
162	MAC_IPV4_AH	331	MAC_IPV4_GTPU_IPV4_FRAG
163	MAC_IPV6_AH	332	MAC_IPV4_GTPU_IPV4_PAY
164	MAC_IPV4_NAT-T-ESP	333	MAC_IP4_GTPU_IPV4_UDP_PAY
165	MAC_IPV6_NAT-T-ESP	334	MAC_IPV4_GTPU_IPV4_TCP
166	MAC_IPV4_NAT-T-IKE	335	MAC_IPV4_GTPU_IPV4_ICMP
167	MAC_IPV6_NAT-T-IKE	336	MAC_IPV6_GTPU_IPV4_FRAG
168	MAC_IPV4-NAT-T-KEEP	337	MAC_IPV6_GTPU_IPV4_PAY
169	MAC_IPV6_NAT-T-KEEP	338	MAC_IPV6_GTPU_IPV4_UDP_PAY
300	MAC_PPPOD_PAY	339	MAC_IPV6_GTPU_IPV4_TCP
301	MAC_PPPoE_PAY	340	MAC_IPV6_GTPU_IPV4_ICMP
302	MAC_PPPoE_IPV4_FRAG	341	MAC_IPV4_GTPU_IPV6_FRAG
303	MAC_PPPoE_IPV4_PAY	342	MAC_IPV4_GTPU_IPV6_PAY
304	MAC_PPPoE_IPV4_UDP_PAY	343	MAC_IPV4_GTPU_IPV6_UDP_PAY
305	MAC_PPPoE_IPV4_TCP	345	MAC_IPV4_GTPU_IPV6_ICMPV6
306	MAC_PPPoE_IPV4_SCTP	346	MAC_IPV4_GTPU_IPV6_FRAG
307	MAC_PPPoE_IPV4_ICMP	347	MAC_IPV6_GTPU_IPV6_PAY
308	MAC_PPPoE_IPV6_FRAG	348	MAC_IPV6_GTPU_IPV6_UDP_PAY
309	MAC_PPPoE_IPV6_PAY	349	MAC_IPV6_GTPU_IPV6_TCP
310	MAC_PPPoE_IPV6_UPD_PAY	350	MAC_IPV6_GTPU_IPV6_ICMPV6
311	MAC_PPPoE_IPV6_TCP	351	MAC_IPV4_PFCP_NODE
312	MAC_PPPoE_IPV6_SCTP	352	MAC_IPV4_PFCP_SESSION
313	MAC_PPPoE_IPV6_ICMPV6	353	MAC_IPV6_PFCP_NODE
325	MAC_IPV4_GTPC_TEID	354	MAC_IPV6_PFCP_SESSION
326	MAC_IPV6_GTPC_TEID	360	MAC_IPV4_L2TPV3
327	MAC_IPV4_GTPC	361	MAC_IPV6_L2TPV3
328	MAC_IPV6_GTPC		

2.3 DDP Package Filtering Capabilities

Refer to the *Intel® Ethernet Controller E810 Dynamic Device Personalization (DDP) Technology Guide* for an example of enhanced Comms DDP package parsing specific Comms protocol.

After Ethernet packets are parsed and PTYPES are determined, certain packets' fields are extracted. Each PTYPE associates with a profile that has its own extraction sequence.

The extraction sequences for RSS and FD are dynamically programmed by the base driver at runtime based on input from the user to redirect packets on FD and RSS blocks.

The following example shows a FD filter to match on a GTP TEID and route such packets to queue 1.

```
testpmd> flow create 0 ingress pattern eth / ipv4 / udp / gtpu teid is 0x12345678 /
gtp_psc / ipv4 / end actions queue index 1 / end
```

For switch filters, certain packet fields extracted for packet matching are set by the user's switch rules.

Table 5 shows the packet fields extraction at Switch block. Shading in green indicates Comms DDP Package-specific support (which requires support from the DPDK driver), while no color indicates fields that are also supported by the OS-default package.

Table 5. Packet Fields Extraction for Switch Filters

Content	Notes
Switch ID	
Packet Direction, Type	
VLAN Flags	
Tunnel Type	
Source VSI	
Outer Destination Address	Outer MAC header.
Outer Source Address	
VLAN ID	First outer VLAN on the wire (if present, usually S-tag).
VLAN ID	Second outer VLAN on the wire (if present, usually C-tag).
Outer Last EtherType	
VNI	For profiles with a UDP Tunnel (VXLAN, VXLAN-GPE, Geneve).
Virtual Subnet ID (VSID) and Flow ID	For profiles with a IP Tunnel (GRE, NVGRE).
VNI	Inner MAC header (tunnel profiles only).
Virtual Subnet ID (VSID) and Flow ID	
Destination Address	
Source Address	
Inner Last EtherType	
Source Port	
Source Port	
Destination Port	
Destination Port	
Version, Traffic Class	
Version, Traffic Class	Outer or Inner IPv6 header (depends on profile).
TTL, Protocol	

Table 5. Packet Fields Extraction for Switch Filters [continued]

Content	Notes
Flow Label	
Source Address	
Source Address	
Destination Address	
Version, Traffic Class	
TTL, Protocol	
Source Address	
Source Address	
Destination Address	
Destination Address	
PPPoE Session Type	
GTP Message Type	
GTP TEID	
GTP PDU	
GTP QoS Flow Identifier	
L2TPv3 Session ID	
ESP SPI (over IP or UDP)	ESP Security Parameters Index
AH SPI	AH Security Parameters Index
PFCP Session Packet SEID	First 4-byte of SEID field

Note: Not all combinations of fields extraction for switch filters are supported. For example, if a tunnel packet contains IPv4 on both inner and outer, both inner and outer IPv4 are extracted. However, if either inner or outer is of IPv6, only inner IP are extracted.

2.4 LTE/5G Capabilities Summary

Table 6. LTE/5G Capabilities Summary

Category	Capability	800 Series NIC (2x100 GbE)
LTE	GTP-U based steering on S1-U interface (based on Inner Src-IP UE Address)	Supported
	GTP-U based steering on S4 & S5/S8 interface (based on Inner Dst-IP UE Address)	Supported
	Concurrent S1-U, S4, & S5/S8 interface specific steering configurations	In plan. Separate configuration per VF. GTP-U header does not provide any delineating information up to Rel-14. Different configuration across VFs enables UE based steering to cores.
LTE/5G	GTP-U based steering on SGi, or N9 interface (based on outer Dst-IP Address)	Supported
	PFCP based steering on N4/Sx interface	Supported
	Inner IP Packet Type Detection	TCP, UDP, ICMP, everything else (“other”)
	Inner IP's TCP header Parsing and Flag Extraction.	Reported via FlexiRx Descriptor.

Table 6. LTE/5G Capabilities Summary [continued]

Category	Capability	800 Series NIC (2x100 GbE)
5G	GTP-U based steering on N3 (based on Inner Src-IP Address)	Supported. Software programmable of outer/inner IP packet fields for steering decisions.
	GTP-U based steering on N9 interface (based on outer Dst-IP Address)	Supported
	Concurrent N3 & N9 interface specific steering configurations	Supported
	Packet priority (QFI, DSCP) based Queue Group	Supported. Load distribution within queue group (for example, 'n' queues per queue group).
Config	RSS configuration granularity	At VF level.
	Number of Rx Queues in PF mode	256
	Number of Rx Queues in VF mode	16 (DPDK 20.05, 20.08), 256 (DPDK 20.11)
	Max number of VFs per device	256
Checksums	Outer L3+L4, Inner L3+L4	Supported (i.e., 4 checksums per packet)

3.0 Comms DDP Package Setup

The 800 Series Comms DDP package supports only Linux-based operating systems at this time.

Currently, the Comms DDP package is fully supported by DPDK versions 19.11, 20.02, 20.05, 20.08, and 20.11. It can be loaded either by DPDK or the 800 Series Linux base driver.

3.1 Comms Package Download

For details on how to set up DPDK, refer to *Intel® Ethernet Controller E810 Data Plane Development Kit (DPDK) Configuration Guide* (Doc ID: 610231).

There are two methods where Comms DDP package can be loaded and used under DPDK (see Section 3.2.1 and Section 3.2.2). For both methods, the user must to obtain the latest DDP Comms package from https://downloadcenter.intel.com/download/29889/Intel-Ethernet-800-Series-Telecommunication-Comms-Dynamic-Device-Personalization-DDP-Package and extract with the Linux **unzip** utility as shown, assuming the zip file is copied to */usr/tmp/*.

```
# cd /usr/tmp
# unzip ice-1.3.24.0.zip -d ice-1.3.24.0
# ls -al
ls -al
total 724
drwxr-xr-x  2 root    root      4096 Sep 20 17:48 .
drwxr-xr-x 19 paeuser paeuser   4096 Sep 20 17:48 ..
-rwxr-xr-x  1 root    root    688388 Jul 22 15:08 ice_comms-1.3.24.0.pkg
lrwxrwxrwx  1 root    root        22 Sep 20 17:48 ice.pkg -> ice_comms-1.3.24.0.pkg
-rwxr-xr-x  1 root    root     22224 Jul 22 15:08
Intel_800_series_market_segment_DDP_license.txt
-rwxr-xr-x  1 root    root     12682 Jul 22 15:08 readme.txt
```

3.2 DDP Package Load

3.2.1 Option 1: *ice* Linux Base Driver

The first option is to have the *ice* Linux base driver load the package.

The *ice* Linux base driver looks for the symbolic link *intel/ice/ddp/ice.pkg* under the default firmware search path, checking the following folders in order:

- */lib/firmware/updates/*
- */lib/firmware/*

1. To install the Comms package, copy the extracted *.pkg* file and its symbolic link to */lib/firmware/updates/intel/ice/ddp* as follows, and reload the *ice* driver:

   ```
   # cd /usr/tmp
   # unzip ice-1.3.24.0.zip -d ice-1.3.24.0

   # cp /usr/tmp/ice-1.3.20/ice-1.3.24.0.pkg /lib/firmware/updates/intel/ice/ddp/
   # cp /usr/tmp/ ice-1.3.20/ice.pkg /lib/firmware/updates/intel/ice/ddp/
   ```

   Or, cerate a symbolic link:

   ```
   # ln -sf /lib/firmware/updates/intel/ice/ddp/ice_comms-1.3.24.0.pkg ice.pkg
   ```

2. Unload *ice*.

```
rmmod ice
```

Note: **rmmod** brings down the interface. To avoid this, execute the following steps to unbind the interface from a kernel driver and binding it to DPDK to reload the Comms DDP package.

3. Check softlink of *ice.pkg*.

```
ll ice.pkg
lrwxrwxrwx. 1 root root 58 Sep 11 07:59 ice.pkg -> /lib/firmware/updates/intel/
ice/ddp/ice_comms-1.3.24.0.pkg
```

4. Load *ice*.

```
modprobe ice
```

Following is an example of a *dmesg* indicating successful loading of the Comms DDP package on a 2-port device. The package is loaded by the first Physical Function (PF), and remaining PFs use the loaded DDP package.

```
[root@bdcped02 ddp]# dmesg | grep 0000:04:00.0
[109935.903489] ice 0000:04:00.0: The DDP package was successfully loaded: ICE COMMS
Package version 1.3.24.0
```

Once the driver loads the package, the user can unbind the *ice* driver from a desired port on the device so that DPDK can utilize the port.

The following example unbinds Port 0 and Port 1 of device on Bus 6, Device 0. Then, the port is bound to either igb_uio or vfio-pci.

```
# ifdown <interface>
# dpdk-devbind -u 06:00.0
# dpdk-devbind -u 06:00.1
# dpdk-devbind -b igb_uio 06:00.0 06:00.1
```

3.2.2 Option 2: DPDK Driver Only

The second method is if the system does not have the *ice* driver installed. In this case, the user can download the DDP package from the Intel download center and extract the zip file to obtain the package (*.pkg*) file. Similar to the Linux base driver, the DPDK driver looks for the *intel/ddp/ice.pkg* symbolic link in the kernel default firmware search path */lib/firmware/updates and /lib/firmware/*.

Copy the extracted DDP *.pkg* file and its symbolic link to */lib/firmware/intel/ice/ddp*, as follows.

```
# cp /usr/tmp/ice-1.3.24.0/ice-1.3.24.0.pkg /lib/firmware/intel/ice/ddp/
# cp /usr/tmp/ice-1.3.24.0/ice.pkg /lib/firmware/intel/ice/ddp/
```

When DPDK driver loads, it looks for *ice.pkg* to load. If the file exists, the driver downloads it into the device. If not, the driver transitions into safe mode.

DPDK's **testpmd** application also indicates the status and version of the loaded DDP package. The example shows the **testpmd** output of a successful Comms package loading.

```
EAL: PCI device 0000:3b:00.1 on NUMA socket 0
EAL: probe driver: 8086:1592 net_ice
ice_load_pkg_type(): Active package is: 1.3.24.0, ICE COMMS Package
```

3.3 Loading Comms Package to a Specific 800 Series Device

On a host system running with multiple 800 Series devices, there is sometimes a need to load a specific DDP package on a selected device while loading a different package on the remaining devices.

The 800 Series Linux base driver and DPDK driver can both load a specific DDP package to a selected adapter based on the device's serial number. The driver does this by looking for a specific symbolic link package filename containing the selected device's serial number.

The following example illustrates how a user can load a specific package (e.g., *ice-1.3.24.0.pkg*) on the device of Bus 6.

1. Find device serial number.

   To view bus, device, and function of all 800 Series Network Adapters in the system:

   ```
   # lspci | grep -i Ethernet | grep -i Intel
   06:00.0 Ethernet controller: Intel Corporation Ethernet Controller E810-C for QSFP
   (rev 01)
   06:00.1 Ethernet controller: Intel Corporation Ethernet Controller E810-C for QSFP
   (rev 01)
   82:00.0 Ethernet controller: Intel Corporation Ethernet Controller E810-C for SFP
   (rev 01)
   82:00.1 Ethernet controller: Intel Corporation Ethernet Controller E810-C for SFP
   (rev 01)
   82:00.2 Ethernet controller: Intel Corporation Ethernet Controller E810-C for SFP
   (rev 01)
   82:00.3 Ethernet controller: Intel Corporation Ethernet Controller E810-C for SFP
   (rev 01)
   ```

   Use the **lspci** command to obtain the selected device serial number:

   ```
   # lspci -vv -s 06:00.0 | grep -i Serial
   Capabilities: [150 v1] Device Serial Number 35-11-a0-ff-ff-ca-05-68
   ```

   Or, fully parsed without punctuation:

   ```
   # lspci -vv -s 06:00.0 |grep Serial |awk '{print $7}'|sed s/-//g
   3511a0ffffca0568
   ```

2. Rename the package file with the device serial number in the name.

   Copy the specific package over to */lib/firmware/updates/intel/ice/ddp* (or */lib/firmware/intel/ice/ddp*) and create a symbolic link with the serial number linking to the package, as shown. The specific symbolic link filename starts with “ice-” followed by the device serial in lower case without dash ('-').

   ```
   # ln -s /lib/firmware/updates/intel/ice/ddp/ice-1.3.24.0.pkg /lib/firmware/
   updates/intel/ice/ddp/ice-3511a0ffffca0568.pkg
   ```

   Or:

   ```
   ln -sf /lib/firmware/updates/intel/ice/ddp/ice_comms-1.3.24.0.pkg ice-
   e0680bffffb7a640.pkg
   ```

   Check softlink:

   ```
   ll ice.pkg
   lrwxrwxrwx. 1 root root 58 Sep 10 01:24 ice.pkg -> /lib/firmware/updates/intel/
   ice/ddp/ice_comms-1.3.24.0.pkg
   ```

3. If using Linux kernel driver (*ice*), reload the base driver (not required if using only DPDK driver).

```
# rmmod ice
# modprobe ice
```

   The driver loads the specific package to the selected device and the OS-default package to the remaining 800 Series devices in the system.

4. Verify.

   For kernel driver:

   Following is an example of successful loading of the specific DDP package on the selected device of Bus 6 and OS-default package on the other device of Bus 82:

```
# dmesg | grep -i "ddp \| safe"
ice 0000:06:00.0: The DDP package was successfully loaded: ICE COMMS Package
version 1.3.24.0
ice 0000:06:00.1: DDP package already present on device: ICE COMMS Package version
1.3.24.0
ice 0000:82:00.0: The DDP package was successfully loaded: ICE OS Default Package
version 1.3.20.0
ice 0000:82:00.1: DDP package already present on device: ICE OS Default Package
version 1.3.20.0
ice 0000:82:00.2: DDP package already present on device: ICE OS Default Package
version 1.3.20.0
ice 0000:82:00.3: DDP package already present on device: ICE OS Default Package
version 1.3.20.0
```

   If using only DPDK driver:

   Verify using DPDK's **testpmd** application to indicate the status and version of the loaded DDP package.

4.0 Comms DDP Package Utilization

4.1 5G UPF

800 Series DDP capabilities support a number of functionalities important for 5G UPF.

- RSS for packet steering based on UE IP Address.
- Queue Group Mapping based on QFI.
- Queue Group Mapping based on DSCP.

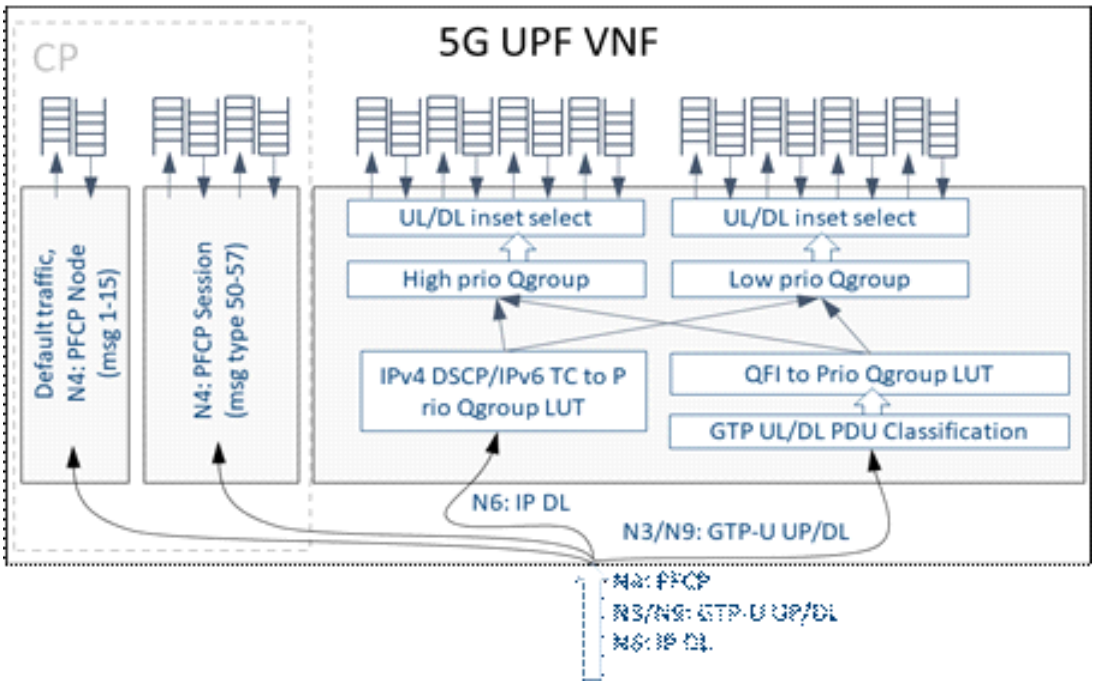

Figure 1. 5G UPF VNF

Table 7 through Table 9 show the pattern types and input sets available to be programmed via RTE FLOW API for RSS. Shading in green indicates Comms DDP Package-specific support (which requires support from the DPDK driver), while no color indicates fields that are also supported by the OS-default package.

Table 7. Patterns and Input Sets for RSS (DPDK 20.05)

Pattern	Input Set
ETH / IPv4(6)	D-IP | S-IP
ETH / IPv4(6) / UDP	D-IP | S-IP | D-PORT | S-PORT
ETH / IPv4(6) / TCP	D-IP | S-IP | D-PORT | S-PORT
ETH / IPv4(6) / SCTP	D-IP | S-IP | D-PORT | S-PORT
ETH / IPv4 / UDP / GTPU / PSC / IPv4	Inner D-IP | Inner S-IP
ETH / IPv4 / UDP / GTPU / PSC / IPv4 / UDP	Inner D-IP | Inner S-IP | Inner S-PORT | Inner D-PORT
ETH / IPv4 / UDP / GTPU / PSC / IPv4 / TCP	Inner D-IP | Inner S-IP | Inner S-PORT | Inner D-PORT

Table 7. Patterns and Input Sets for RSS (DPDK 20.05) [continued]

Pattern	Input Set
ETH / IPv4(6) / UDP / ESP	D-IP | S-IP | SPI
ETH / IPv4(6) / UDP / AH	D-IP | S-IP | SPI
ETH / IPv4(6) / UDP / PFCP (S_FIELD = 1)	D-IP | S-IP | SEID
ETH / IPv4(6) / L2TP	D-IP | S-IP | SESSION ID

Table 8. Patterns and Input Sets for RSS (DPDK 20.08)

Pattern	Input Set
ETH / IPv4(6)	D-IP | S-IP
ETH / IPv4(6) / UDP	D-IP | S-IP | D-PORT | S-PORT | PROT
ETH / IPv4(6) / TCP	D-IP | S-IP | D-PORT | S-PORT | PROT
ETH / IPv4(6) / SCTP	D-IP | S-IP | D-PORT | S-PORT | PROT
ETH / IPv4(6) / UDP / GTPU / IPv4(6)	Inner D-IP | Inner S-IP
ETH / IPv4(6) / UDP / GTPU / IPv4(6) / UDP	Inner D-IP | Inner S-IP | Inner S-PORT | Inner D-PORT | PROT
ETH / IPv4(6) / UDP / GTPU / IPv4(6) / TCP	Inner D-IP | Inner S-IP | Inner S-PORT | Inner D-PORT | PROT
ETH / IPv4(6) / UDP / GTPU / PSC / IPV4(6)	Inner D-IP | Inner S-IP
ETH / IPv4(6) / UDP / GTPU / PSC / IPV4(6) / UDP	Inner D-IP | Inner S-IP | Inner S-PORT | Inner D-PORT | PROT
ETH / IPv4(6) / UDP / GTPU / PSC / IPV4(6) / TCP	Inner D-IP | Inner S-IP | Inner S-PORT | Inner D-PORT | PROT
ETH / IPv4(6) / UDP / GTPU	D-IP | S-IP
ETH / IPv4(6) / UDP / ESP	D-IP | S-IP | SPI
ETH / IPv4(6) / UDP / AH	D-IP | S-IP | SPI
ETH / IPv4(6) / PFCP (S_FIELD = 1)	D-IP | S-IP | SEID
ETH / IPv4(6) / L2TP	D-IP | S-IP | SESSION ID

Notes:

1. GTP-U UL/DL can have different RSS input set based on pdu type, example:
   - Rule 1: Pattern ETH / IPV4 / UDP / GTPU / PSC (pdu_type = 0) / IPV4, Inputset: src IP
   - Rule 2: Pattern ETH / IPV4 / UDP / GTPU / PSC (pdu_type = 1) / IPV4, Inputset: dst IP
2. Packet with same inner IP but different outer IP cannot have separated input set due to DDP package limitation, for example, below 2 rules cannot co-exist.
   - Rule 1: Pattern ETH / IPV4 / UDP / GTPU / IPV4, Input set: src IP
   - Rule 2: Pattern ETH / IPV6 / UDP / GTPU / IPV4, Input set: dst IP

   This is fixed in DPDK 20.11.
3. Toeplitz Symmetric hash is supported.
4. For Input set with L4, protocol type will be included by default.
5. For IPv6 address, 32bit, 48bit, 64bit prefix RSS support will be added in DPDK 20.11.

Table 9. Patterns and Input Sets for RSS (DPDK 20.11)

Pattern	Input Set
ETH / IPv4(6)	D-IP | S-IP
ETH / IPv4(6) / UDP	D-IP | S-IP | D-PORT | S-PORT | PROT
ETH / IPv4(6) / TCP	D-IP | S-IP | D-PORT | S-PORT | PROT
ETH / IPv4(6) / SCTP	D-IP | S-IP | D-PORT | S-PORT | PROT
ETH / IPv4(6) / UDP / GTPU / IPv4(6)	Inner D-IP | Inner S-IP
ETH / IPv4(6) / UDP / GTPU / IPv4(6) / UDP	Inner D-IP | Inner S-IP | Inner S-PORT | Inner D-PORT | PROT
ETH / IPv4(6) / UDP / GTPU / IPv4(6) / TCP	Inner D-IP | Inner S-IP | Inner S-PORT | Inner D-PORT | PROT
ETH / IPv4(6) / UDP / GTPU / PSC / IPV4(6)	Inner D-IP | Inner S-IP
ETH / IPv4(6) / UDP / GTPU / PSC / IPV4(6) / UDP	Inner D-IP | Inner S-IP | Inner S-PORT | Inner D-PORT | PROT
ETH / IPv4(6) / UDP / GTPU / PSC / IPV4(6) / TCP	Inner D-IP | Inner S-IP | Inner S-PORT | Inner D-PORT | PROT
ETH / IPv4(6) / UDP / GTPU	D-IP | S-IP
ETH / IPv4(6) / UDP / ESP	D-IP | S-IP | SPI
ETH / IPv4(6) / UDP / AH	D-IP | S-IP | SPI
ETH / IPv4(6) / PFCP (S_FIELD = 1)	D-IP | S-IP | SEID
ETH / IPv4(6) / L2TP	D-IP | S-IP | SESSION ID
ETH / IPv4(6) / GTPU (no inner IP)	D-IP / S-IP
ETH / IPv4(6) / GTPC	D-IP / S-IP

Note: For IPv6 address: 32-bit, 48-bit, 64-bit prefix RSS support is added in DPDK 20.11.

Table 10 and Table 11 show the pattern types and input sets available to be programmed via RTE FLOW API for the Intel® Ethernet Flow Director. Shading in green indicates Comms DDP Package-specific support (which requires support from the DPDK driver), while no color indicates fields that are also supported by the OS-default package.

Table 10. Patterns and Input Sets for Intel® Ethernet Flow Director (DPDK 20.05)

Pattern	Input Set
ETH	ETHERTYPE
ETH / IPv4(6)	D-IP | S-IP | PROTO | TOS | TTL
ETH / IPv4(6) / UDP	D-IP | S-IP | PROTO | TOS | TTL | D-PORT | S-PORT
ETH / IPv4(6) / TCP	D-IP | S-IP | PROTO | TOS | TTL | D-PORT | S-PORT
ETH / IPv4(6) / SCTP	D-IP | S-IP | PROTO | TOS | TTL | D-PORT | S-PORT
ETH / IPv4 / GTPU	TEID | D-IP | S-IP
ETH / IPv4 / GTPU / PSC	TEID | QFI | D-IP | S-IP
ETH / IPv4(6) / AH	SPI
ETH / IPv4(6) / ESP	SPI
ETH / IPv4(6) / UDP / ESP	D-IP | S-IP | SPI
ETH / IPv4(6) / UDP / PFCP	S-FIELD
ETH / IPv4(6) / L2TP	SESSION ID

Table 11. Patterns and Input Sets for Intel® Ethernet Flow Director (DPDK 20.08 and 20.11)

Pattern	Input Set
ETH	ETHERTYPE
ETH / IPv4(6)	D-IP | S-IP | PROTO | TOS | TTL
ETH / IPv4(6) / UDP	D-IP | S-IP | PROTO | TOS | TTL | D-PORT | S-PORT
ETH / IPv4(6) / TCP	D-IP | S-IP | PROTO | TOS | TTL | D-PORT | S-PORT
ETH / IPv4(6) / SCTP	D-IP | S-IP | PROTO | TOS | TTL | D-PORT | S-PORT
ETH / IPv4 / GTPU	TEID | D-IP | S-IP
ETH / IPv4 / GTPU / PSC	TEID | QFI | D-IP | S-IP
ETH / IPv4(6) / AH	SPI
ETH / IPv4(6) / ESP	SPI
ETH / IPv4(6) / UDP / ESP	D-IP | S-IP | SPI
ETH / IPv4(6) / UDP / PFCP	S-FIELD
ETH / IPv4(6) / L2TP	SESSION ID
Note: In DPDK 20.08, rules for GTPU with outer IPv6 filter have been added, but rules for GTPU with outer IPv4 and outer IPv6 still cannot co-exist due to DDP package limitation. This limitation has been removed in DPDK 20.11.	

4.1.1 RSS for Packet Steering Based on UE IP Address

Packets received by the UPF on the 5G N3 interface are GTP-U encapsulated IPv4 packets. In this context the Source IP Address of the inner IPv4 packet represents the UE IP.

Packets received by the UPF on the 5G N6 interface are IPv4 packets. In this context the Destination IP Address represents the UE IP.

RSS typically operates on a hash of the Source IP Address/Destination IP Address/Source Port/ Destination Port/Protocol, which on the N3/N6 interfaces might not provide sufficient entropy across the receive queues.

To take advantage of cache locality, the same CPU core should handle all traffic associated with a particular UE.

DDP on 800 Series devices can be configured with appropriate RTE Flow Rules such that:

- GTP-U encapsulated packets received on the N3 interface are identified, and RSS is performed on a hash of the inner Source IP Address only.
- IPv4 packets received on the N6 interface are identified, and RSS is performed on a hash of the Destination IP Address only.

Given a sufficiently wide range of UE IP Addresses, this leads to a better distribution across all the receive queues, and the same receive queue is always used for a given UE IP Address. Receive queues can be associated with particular cores to take advantage of Cache locality.

Note: DPDK 20.08 requires a patch for RSS on inner Source IP Address only for GTPU encapsulated packets. Please contact Intel from further information.

4.1.2 RSS for Packet Steering Based on Flow

RSS operates on a hash of the Source IP Address/Destination IP Address/Source Port/Destination Port/ Protocol.

IPv4 packets received on the N6 interface and GTPU Encapsulated IPv4 packets on the N3 interface with identical flows (Source IP Address/Destination IP Address/Source Port/Destination Port/Protocol) are steered to the same receive queue associated with a particular core/thread. Therefore, all packets in a flow can be handled by the same core/thread.

4.1.3 Queue Group Mapping Based on QFI

Packets received by the UPF on the 5G N3 interface are GTP-U encapsulated IPv4 packets. The GTP-U packet contains the PDU Session Information Extension Header.

For Ultra-Reliable Low-Latency Communication (URLLC), a Quality of Service Flow Indicator (QFI) parameter found within the PDU Session Information is used to determine the priority level of the packet. The UPF can designate a number of receive queue groups with varying priority levels.

DDP on 800 Series devices can be configured with appropriate RTE Flow Rules such that GTP-U encapsulated packets received on the N3 interface are identified and directed to a queue group based on the QFI value. In this manner, a number of RTE Flow Rules can be created each directing packets to a queue group based on the QFI value. Refer to Figure 1, which shows two queue groups (high and low priority).

4.1.4 Queue Group Mapping Based on DSCP

Packets received by the UPF on the 5G N6 interface are IPv4 packets.

For URLLC, the DSCP parameter found within the TOS service field in the IPV4 header is used to determine the priority level of the packet. The UPF may designate a number of receive queue groups with varying priority levels.

DDP on 800 Series devices can be configured with appropriate RTE Flow Rules such that Ipv4 packets received on the N6 interface are identified and directed to a queue group based on the DSCP value. In this manner, a number of RTE Flow Rules can be created each directing packets to a queue group based on the DSCP value. Refer to Figure 1 which shows 2 queue groups (high and low priority).

4.2 Limitations

The maximum number of receive queues per PF is 64 and the maximum number of receive queues per VF is 16. Later releases of DPDK 20.08/20.11 will address these limitations.

For DPDK 19.11 onwards, the following workaround can provide up to 256 receive queues per PF:

In *drivers/net/ice/ice_ethdev.h*, increase `ICE_MAX_Q_PER_TC` from 64 to 256 and rebuild DPDK.

```
#define ICE_MAX_Q_PER_TC         256
```

As previously stated, DPDK RTE Flow Rules can direct packets to queue groups or queue region:

- The queue region size should be any of the following values 2, 4, 8, 16, 32, 64, 128, as long as the total number of queues do not exceed the VSI allocation.
- The queue numbers within the queue region must be contiguous.

4.3 RTE Flow API

The DPDK Generic flow API (*rte_flow*) is used to the configure the 800 Series device to match specific ingress traffic and forward it to specified queues.

The specific ingress traffic is identified by a matching pattern that is composed of one or more Pattern items (represented by struct *rte_flow_item*). Once a match has been determined, one or more associated Actions (represented by struct *rte_flow_action*) is performed.

A number of flow rules can be combined such that one rule directs traffic to a queue group based on QFI/DSCP, etc, and a second rule distributes matching packets within that queue group using RSS.

The following subset of the RTE Flow API functions can be used to validate, create, and destroy RTE Flow Rules. Refer to the DPDK Documentation for more information.

4.3.1 RTE Flow Rule Validation

An RTE Flow Rule is created via a call to the function *rte_flow_validate*. This can be used to check the rule for correctness and whether it would be accepted by the device given sufficient resources.

```
int
    rte_flow_validate(uint16_t port_id,
        const struct rte_flow_attr *attr,
        const struct rte_flow_item pattern[],
        const struct rte_flow_action *actions[]
        struct rte_flow_error *error);
```

where:

port_id	=	Port identifier of Ethernet device.
attr	=	Flow Rule attributes (ingress/egress).
pattern	=	Pattern specification (list terminated by the END pattern item).
action	=	Associated actions (list terminated by the END action).
error	=	Perform verbose error reporting if not NULL.

0 is returned upon success, negative errno otherwise.

4.3.2 RTE Flow Rule Creation

An RTE Flow Rule is created via a call to the function *rte_flow_create*.

```
struct rte_flow *
    rte_flow_create(uint16_t port_id,
        const struct rte_flow_attr *attr,
        const struct rte_flow_item pattern[],
        const struct rte_flow_action *actions[]
        struct rte_flow_error *error);
```

where:

port_id	=	Port identifier of Ethernet device.
attr	=	Flow Rule attributes (ingress/egress).
pattern	=	Pattern specification (list terminated by the END pattern item).

action = Associated actions (list terminated by the END action).

error = Perform verbose error reporting if not NULL.

A valid handle is returned upon success, NULL otherwise.

4.3.3 RTE Flow Rule Destruction

An RTE Flow Rule is destroyed via a call to the function *rte_flow_destroy*.

```
int
    rte_flow_destroy(uint16_t port_id,
        struct rte_flow *flow,
        struct rte_flow_error *error);
```

where:

port_id = Port identifier of Ethernet device.

flow = Flow Rule handle to destroy.

error = Perform verbose error reporting if not NULL.

0 is returned upon success, negative errno otherwise.

4.3.4 RTE Flow Flush

All flow rule handles associated with a port can be released using *rte_flow_flush*. They are released as with successive calls to function *rte_flow_destroy*.

```
int
    rte_flow_flush(uint16_t port_id,
        struct rte_flow_error *error);
```

where:

port_id = Port identifier of Ethernet device.

error = Perform verbose error reporting if not NULL.

0 is returned upon success, negative errno otherwise.

4.3.5 RTE Flow Query

A RTE Flow Rule is queried via a call to the function *rte_flow_query*.

```
int
    rte_flow_query(uint16_t port_id,
        struct rte_flow *flow,
        const struct rte_flow_action *action,
        void *data,
        struct rte_flow_error *error);
```

where:

port_id	=	Port identifier of Ethernet device.
flow	=	Flow Rule handle to query.
action	=	Action to query. This must match prototype from flow rule.
data	=	Pointer to storage for the associated query data type.
error	=	Perform verbose error reporting if not NULL.

0 is returned upon success, negative errno otherwise.

4.4 RTE Flow Rules

A flow rule is the combination of attributes with a matching pattern and a list of actions. Each flow rule consists of:

- **Attributes (represented by struct *rte_flow_attr*)** — Properties of a flow rule such as its direction (ingress or egress) and priority.
- **Pattern Items (represented by struct *rte_flow_item*)** — Part of a matching pattern that matches either specific packet data or traffic properties.
- **Matching pattern** — Traffic properties to look for, a combination of any number of items.
- **Actions (represented by struct *rte_flow_action*)** — Operations to perform whenever a packet is matched by a pattern.

4.4.1 Attributes

Flow Rule patterns apply to inbound and/or outbound traffic. For the purposes described in later sections, the rules apply to ingress only. For further information, refer to Section 11 of the *DPDK Programmers Guide*.

```
struct rte_flow_attr {
    uint32_t group;
    uint32_t priority;
    uint32_t ingress:1;
    uint32_t egress:1;
    uint32_t transfer:1;
    uint32_t reserved:29;
};
```

As of DPDK 19.11/20.05, egress, priority, and group are not supported.

4.4.2 Pattern Items

For the purposes described in later sections, Pattern items are primarily for matching protocol headers and packet data, usually associated with a specification structure. These must be stacked in the same order as the protocol layers to match inside packets, starting from the lowest.

Item specification structures are used to match specific values among protocol fields (or item properties).

Up to three structures of the same type can be set for a given item:

- **spec** — Values to match (e.g. a given IPv4 address).
- **last** — Upper bound for an inclusive range with corresponding fields in spec. Use of last is not supported in DPDK 19.11.
- **mask** — Bit-mask applied to both spec and last, the purpose of which is to distinguish the values to take into account and/or partially mask them out (for example, to match an IPv4 Address prefix). Use of masks are limited in DPDK 19.11.

Table 12 shows all the RTE FLOW item types and associated specification structures supported in DPDK 19.11 for the 800 Series.

Table 12. RTE FLOW Item Types

Item Type[1]	Description	Specification Structure
END	End marker for item lists.	None
VOID	Used as a placeholder for convenience.	None
ETH	Matches an Ethernet header.	rte_flow_item_eth
VLAN	Matches an 802.1Q/ad VLAN tag.	rte_flow_item_vlan
IPV4	Matches an IPv4 header.	rte_flow_item_ipv4
IPV6	Matches an IPv6 header.	rte_flow_item_ipv6
ICMP	Matches an ICMP header.	rte_flow_item_icmp
UDP	Matches a UDP header.	rte_flow_item_udp
TCP	Matches a TCP header.	rte_flow_item_tcp
SCTP	Matches an SCTP header.	rte_flow_item_sctp
VXLAN	Matches a VXLAN header.	rte_flow_item_vxlan
NVGRE	Matches an NVGRE header.	rte_flow_item_nvgre
GTP	Matches a GTP header.	rte_flow_item_gtp
GTPU	Matches a GTPU header.	rte_flow_item_gtp
ARP_ETH_IPV4	Matches an ARP header for Ethernet/IPv4.	rte_flow_item_arp_eth_ipv4
ICMP6	Matches an ICMPv6 header.	rte_flow_item_icmp6
GTP_PSC	Matches a GTP extension header: PDU session container	rte_flow_item_gtp_psc
PPPOES	Matches a PPPoE header.	rte_flow_item_pppoe
PPPOED	Matches a PPPoE optional proto_id field.	rte_flow_item_pppoe
PPPOE_PROTO_ID	Matches a PPPoE session protocol identifier[2]	rte_flow_item_pppoe_proto_id
ESP	Matches an ESP Header[2]	rte_flow_item_esp
AH	Matches an AH header[2]	rte_flow_item_ah
L2TPV3OIP	Matches a L2TPv3 over IP header[2]	rte_flow_item_l2tpv3oip
PFCP	Matches PFCP Header	rte_flow_item_pfcp

1. RTE FLOW items are prefixed by "RTE_FLOW_ITEM_TYPE_". For example, RTE_FLOW_ITEM_TYPE_END.
2. Supported in DPDK 20.05.

Following are more detailed descriptions of the specification structures referenced in this document:

RTE_FLOW_ITEM_TYPE_ETH

```
struct rte_flow_item_eth {
    struct rte_ether_addr dst; /**< Destination MAC. */
    struct rte_ether_addr src; /**< Source MAC. > */
    rte_be16_t type;           /**< EtherType or TPID.> */
};

struct rte_ether_addr {
    uint8_t addr_bytes[RTE_ETHER_ADDR_LEN]; /**< Addr bytes in tx order */
}
```

RTE_FLOW_ITEM_TYPE_IPV4

```
struct rte_flow_item_ipv4 {
    struct rte_ipv4_hdr hdr; /**< IPv4 header definition. */
};

struct rte_ipv4_hdr {
    uint8_t  version_ihl;       /**< version and header length */
    uint8_t  type_of_service;   /**< type of service */
    rte_be16_t total_length;    /**< length of packet */
    rte_be16_t packet_id;       /**< packet ID */
    rte_be16_t fragment_offset; /**< fragmentation offset */
    uint8_t  time_to_live;      /**< time to live */
    uint8_t  next_proto_id;     /**< protocol ID */
    rte_be16_t hdr_checksum;    /**< header checksum */
    rte_be32_t src_addr;        /**< source address */
    rte_be32_t dst_addr;        /**< destination address */
}
```

RTE_FLOW_ITEM_TYPE_UDP

```
struct rte_flow_item_udp {
    struct rte_udp_hdr hdr; /**< UDP header definition. */
};

struct rte_udp_hdr {
    rte_be16_t src_port;    /**< UDP source port. */
    rte_be16_t dst_port;    /**< UDP destination port. */
    rte_be16_t dgram_len;   /**< UDP datagram length */
    rte_be16_t dgram_cksum; /**< UDP datagram checksum */
}
```

RTE_FLOW_ITEM_TYPE_GTP

```
struct rte_flow_item_gtp {
    /**
      * Version (3b), protocol type (1b), reserved (1b),
      * Extension header flag (1b),
      * Sequence number flag (1b),
      * N-PDU number flag (1b).
      */
     uint8_t v_pt_rsv_flags;
     uint8_t msg_type;   /**< Message type. */
     rte_be16_t msg_len; /**< Message length. */
     rte_be32_t teid;    /**< Tunnel endpoint identifier. */
};
```

RTE_FLOW_ITEM_TYPE_GTP_PSC

```
struct rte_flow_item_gtp_psc {
     uint8_t pdu_type; /**< PDU type. */
     uint8_t qfi;      /**< QoS flow identifier. */
};
```

4.4.3 Matching Patterns

A matching pattern is formed by stacking items starting from the lowest protocol layer to match. Patterns are terminated by END pattern item.

For more detail, refer to Table 14, Table 16, Table 18, Table 20, Table 21, Table 23, and Table 24.

4.4.4 Actions

Each possible action is represented by a type. An action can have an associated configuration object. Actions are terminated by the END action.

Table 13 shows all the RTE FLOW action types and associated configuration structures supported by DPDK 19.11 for the 800 Series.

Table 13. RTE FLOW Actions

Action[1]	Description	Configuration Structure
END	End marker for item lists.	None
VOID	Used as a placeholder for convenience.	None
PASSTHRU	Leaves traffic up for additional processing by subsequent flow rules; makes a flow rule non-terminating.	None
MARK	Attaches an integer value to packets and sets PKT_RX_FDIR and PKT_RX_FDIR_ID *mbuf* flags.	rte_flow_action_mark
QUEUE	Assigns packets to a given queue index.	rte_flow_action_queue
DROP	Drops packets.	None
COUNT	Enables Counters for this flow rule	rte_flow_action_count
RSS	Similar to QUEUE, except RSS is additionally performed on packets to spread them among several queues according to the provided parameters.	rte_flow_action_rss
VF	Directs matching traffic to a given virtual function of the current device[2]	rte_flow_action_vf

1. RTE FLOW actions are prefixed by "RTE_FLOW_ACTION_TYPE_". For example, RTE_FLOW_ACTION_TYPE_END.
2. Supported in DPDK 20.05.

For the purposes described in later sections, the RSS action (RTE_FLOW_ACTION_TYPE_RSS) is used to perform RSS on packets to spread them among several queues according to the provided parameters. The relevant configuration structure is described below:

```
struct rte_flow_action_rss {
    enum rte_eth_hash_function func; /**< RSS hash function to apply. */
    uint32_t level;
    uint64_t types; /**< Specific RSS hash types(see ETH_RSS_*). */
    uint32_t key_len; /**< Hash key length in bytes. */
    uint32_t queue_num; /**< Number of entries in @p queue. */
    const uint8_t *key; /**< Hash key. */
    const uint16_t *queue; /**< Queue indices to use. */
};
```

The RSS action can be used with associated matching patterns to:

- Direct matching packets to one or more queues in a queue group.
- Specify the RSS hash type and associated hash function to perform RSS on matching packets.

Due to a limitation in DPDK, it is not possible to direct packets to queues and set the RSS hash type and function in the same RTE flow rule. Thus, two separate rules must be applied.

When configuring the RSS type:

- The queue_num is set to 0.
- The hash function can be set to one of the following:
  - RTE_ETH_HASH_FUNCTION_DEFAULT
  - RTE_ETH_HASH_FUNCTION_SIMPLE_XOR
  - RTE_ETH_HASH_FUNCTION_SYMMETRIC_TOEPLITZ
- To ensure RSS on the IPv4 source address the types can be set as follows:
  - ETH_RSS_IPV4 | ETH_RSS_L3_SRC_ONLY
- To ensure RSS on the IPv4 destination address the types can be set as follows:
  - ETH_RSS_IPV4 | ETH_RSS_L3_DST_ONLY
- To ensure RSS on the outer flow the types can be set as one of the following:
  - ETH_RSS_NONFRAG_IPV4_UDP
  - ETH_RSS_NONFRAG_IPV4_TCP
- To ensure RSS on the inner flow the types can be set as one of the following:
  - ETH_RSS_NONFRAG_IPV4_UDP or
  - ETH_RSS_NONFRAG_IPV4_TCP

When configuring the Queue group:

- The *queue_num* is set to the number of queues in the group.
- The queue is set to a contiguous list of queue indices.

For more detail, refer to Table 15, Table 17, Table 19, Table 22 and Table 25.

4.5 RSS on Outer Destination IPv4 Address

In this section, an RTE Flow Rule is created to match an IPv4 packet and to then perform RSS based on a hash of the outer Destination IP Address.

4.5.1 Pattern Items

Table 14. Pattern Items to Match IPv4 Packet

Index	Item	Spec	Mask
0	Ethernet	0	0
1	IPv4	0	0
2	END	0	0

The following code sets up the *RTE_FLOW_ITEM_TYPE_ETH* and *RTE_FLOW_ITEM_TYPE_IPV4* pattern items.

```
struct rte_flow_item patterns[MAX_PATTERN_NUM];
struct rte_flow_item_ipv4 ipv4_spec;
struct rte_flow_item_ipv4 ipv4_mask;

/* Ethernet */
patterns[0].type = RTE_FLOW_ITEM_TYPE_ETH;
patterns[0].spec = 0;
patterns[0].mask = 0;

/* IPv4 */
patterns[1].type = RTE_FLOW_ITEM_TYPE_IPV4;
patterns[1].spec = &ipv4_spec;
patterns[1].mask = &ipv4_mask;

/* END the pattern array */
patterns[2].type = RTE_FLOW_ITEM_TYPE_END
```

4.5.2 Actions

Table 15. RSS Actions for Outer Destination Address

Index	Action	Fields	Description	Value
0	RSS	rss_conf	types	ETH_RSS_IPV4 | ETH_RSS_L3_DST_ONLY
			func	RTE_ETH_HASH_FUNCTION_DEFAULT
			key_len	0
		queue_num	Number of entries in queue[].	NULL
		queue[]	Queue indices to use.	0
1	END			

The following code sets up the action `RTE_FLOW_ACTION_TYPE_RSS` and calls the *rte_flow_create* function to create the RTE Flow Rule.

Note: The number of queues is 0. RSS spreads traffic across all receive queues based on the hash of the Outer Destination Address.

```
struct rte_flow *handle;
struct rte_flow_error err;
struct rte_flow_action actions[MAX_ACTION_NUM];
struct rte_flow_attr attributes = {.ingress = 1 };

struct rte_flow_action_rss action_rss = (struct rte_flow_action_rss){
    .types =  ETH_RSS_L3_DST_ONLY,
    .func = RTE_ETH_HASH_FUNCTION_DEFAULT,
    .key_len = 0,
};

actions[0].type = RTE_FLOW_ACTION_TYPE_RSS;
actions[0].conf = &action_rss;

actions[1].type = RTE_FLOW_ACTION_TYPE_END;

handle = rte_flow_create (port_id, &attributes, patterns, actions, &err);
```

4.6 RSS on Inner Source IPv4 Address

In this section, an RTE Flow Rule is created to match a GTP-U encapsulated packet and to perform RSS based on a hash of the inner Source IP Address.

4.6.1 Pattern Items

Table 16. Pattern Items to Match GTP-U Encapsulated Packet

Index	Item	Spec	Mask
0	Ethernet	0	0
1	IPv4	0	0
2	UDP	0	0
3	GTPU	0	0
4	GTPU_PSC	pdu_type = 0/1	pdu_type = 0xFF
5	IPv4	0	0
6	END	0	0

The following code sets up the *RTE_FLOW_ITEM_TYPE_ETH*, *RTE_FLOW_ITEM_TYPE_IPV4*, *RTE_FLOW_ITEM_TYPE_UDP*, *RTE_FLOW_ITEM_TYPE_GTP*, *RTE_FLOW_ITEM_TYPE_GTP_PSC*, and *RTE_FLOW_ITEM_TYPE_IPV4* pattern items.

```
struct rte_flow_item patterns[MAX_PATTERN_NUM];
struct rte_flow_item_gtp_psc gtp_psc_spec;
struct rte_flow_item_gtp_psc gtp_psc_mask;

/* Ethernet */
patterns[0].type = RTE_FLOW_ITEM_TYPE_ETH;
patterns[0].spec = 0;
patterns[0].mask = 0;

/* IPv4 */
patterns[1].type = RTE_FLOW_ITEM_TYPE_IPV4;
patterns[1].spec = 0;
patterns[1].mask = 0;
```

```
/* UDP */
patterns[2].type = RTE_FLOW_ITEM_TYPE_UDP;
patterns[2].spec = 0;
patterns[2].mask = 0;

/* GTP */
patterns[3].type = RTE_FLOW_ITEM_TYPE_GTPU;
patterns[3].spec = 0;
patterns[3].mask = 0;

/* GTP PSC */
patterns[4].type = RTE_FLOW_ITEM_TYPE_GTP_PSC;
patterns[4].spec = &gtp_psc_spec;
patterns[4].mask = &gtp_psc_mask;

/* IPv4 */
patterns[5].type = RTE_FLOW_ITEM_TYPE_IPV4;
patterns[5].spec = 0;
patterns[5].mask = 0;

/* END the pattern array */
patterns[6].type = RTE_FLOW_ITEM_TYPE_END;
```

4.6.2 Actions

Table 17. RSS Actions for Inner Source Address

Index	Action	Fields	Description	Value
0	RSS	rss_conf	types	ETH_RSS_IPV4 | ETH_RSS_L3_SRC_ONLY
			func	RTE_ETH_HASH_FUNCTION_DEFAULT
			key_len	0
		queue_num	Number of entries in queue[].	NULL
		queue[]	Queue indices to use.	0
1	END			

The following code sets up the action *RTE_FLOW_ACTION_TYPE_RSS* and calls the *rte_flow_create* function to create the RTE Flow Rule.

Note: The number of queues is 0. RSS spreads traffic across all receive queues based on the hash of the Inner Source Address.

```
struct rte_flow *handle;
struct rte_flow_error err;
struct rte_flow_action actions[MAX_ACTION_NUM];
struct rte_flow_attr attributes = {.ingress = 1 };

struct rte_flow_action_rss action_rss = (struct rte_flow_action_rss){
    .types =  ETH_RSS_L3_SRC_ONLY,
    .func = RTE_ETH_HASH_FUNCTION_DEFAULT,
    .key_len = 0,
};
```

```
actions[0].type = RTE_FLOW_ACTION_TYPE_RSS;
actions[0].conf = &action_rss;

actions[1].type = RTE_FLOW_ACTION_TYPE_END;

handle = rte_flow_create (port_id, &attributes, patterns, actions, &err);
```

4.7 RSS on Outer Flow

In this section, an RTE Flow Rule is created to match an IPv4/UDP packet and to then perform RSS based on a hash of the outer Destination IP Address/Source IP Address/Destination Port/Source Port.

4.7.1 Pattern Items

Table 18. Pattern Items to Match IPv4 Packet

Index	Item	Spec	Mask
0	Ethernet	0	0
1	IPv4	0	0
2	UDP	0	0
3	END	0	0

The following code sets up the *RTE_FLOW_ITEM_TYPE_ETH*, *RTE_FLOW_ITEM_TYPE_IPV4*, and *RTE_FLOW_ITEM_TYPE_UDP* pattern items.

```
struct rte_flow_item patterns[MAX_PATTERN_NUM];
struct rte_flow_item_ipv4 ipv4_spec;
struct rte_flow_item_ipv4 ipv4_mask;

/* Ethernet */
patterns[0].type = RTE_FLOW_ITEM_TYPE_ETH;
patterns[0].spec = 0;
patterns[0].mask = 0;

/* IPv4 */
patterns[1].type = RTE_FLOW_ITEM_TYPE_IPV4;
patterns[1].spec = &ipv4_spec;
patterns[1].mask = &ipv4_mask;

/* UDP */
patterns[2].type = RTE_FLOW_ITEM_TYPE_UDP;
patterns[2].spec = 0;
patterns[2].mask = 0;

/* END the pattern array */
patterns[3].type = RTE_FLOW_ITEM_TYPE_END;
```

4.7.2 Actions

Table 19. RSS Actions for Outer Flow

Index	Action	Fields	Description	Value
0	RSS	rss_conf	types	ETH_RSS_NONFRAG_IPV4_UDP or ETH_RSS_NONFRAG_IPV4_TCP
			func	RTE_ETH_HASH_FUNCTION_SYMMETRIC_TOEPLITZ
			key_len	0
		queue_num	Number of entries in queue[].	NULL
		queue[]	Queue indices to use.	0
1				

The following code sets up the action *RTE_FLOW_ACTION_TYPE_RSS* and calls the *rte_flow_create* function to create the RTE Flow Rule.

Note: The number of queues is 0. RSS will spread traffic across all receive queues based on the hash of the Outer Destination Address/Source Address/Destination Port/Source Port.

```
struct rte_flow *handle;
struct rte_flow_error err;
struct rte_flow_action actions[MAX_ACTION_NUM];
struct rte_flow_attr attributes = {.ingress = 1 };

struct rte_flow_action_rss action_rss = (struct rte_flow_action_rss){
              .types =   ETH_RSS_NONFRAG_IPV4_UDP ,
              .func =  RTE_ETH_HASH_FUNCTION_SYMMETRIC_TOEPLITZ,
              .key_len = 0,
 };

actions[0].type = RTE_FLOW_ACTION_TYPE_RSS;
actions[0].conf = &action_rss;

actions[1].type = RTE_FLOW_ACTION_TYPE_END;

handle = rte_flow_create (port_id, &attributes, patterns, actions, &err);
```

4.8 RSS on Inner Flow

In this section, an RTE Flow Rule is created to match a GTP-U encapsulated IPv4/UDP packets and to perform based on a hash of the outer Destination IP Address/Source IP Address/Destination Port/ Source Port.

4.8.1 Pattern Items

Table 20. Pattern Items to Match GTP-U Encapsulated Packet

Index	Item	Spec	Mask
0	Ethernet	0	0
1	IPv4	0	0
2	UDP	0	0
3	GTPU	0	0
4	GTP_PSC	pdu_type = 0/1	pdu_type = 0xFF
5	IPv4	0	0
6	UDP	0	0
7	END	0	0

The following code sets up the *RTE_FLOW_ITEM_TYPE_ETH*, *RTE_FLOW_ITEM_TYPE_IPV4*, *RTE_FLOW_ITEM_TYPE_UDP*, *RTE_FLOW_ITEM_TYPE_GTPU*, RTE_FLOW_ITEM_TYPE_GTP_PSC, RTE_FLOW_ITEM_TYPE_IPV4, and RTE_FLOW_ITEM_TYPE_UDP pattern items. The example below is for uplink (pdu_type =1).

```
struct rte_flow_item patterns[MAX_PATTERN_NUM];
struct rte_flow_item_gtp_psc gtp_psc_spec;
struct rte_flow_item_gtp_psc gtp_psc_mask;

/* Ethernet */
patterns[0].type = RTE_FLOW_ITEM_TYPE_ETH;
patterns[0].spec = 0;
patterns[0].mask = 0;

/* IPv4 */
patterns[1].type = RTE_FLOW_ITEM_TYPE_IPV4;
patterns[1].spec = 0;
patterns[1].mask = 0;

/* UDP */
patterns[2].type = RTE_FLOW_ITEM_TYPE_UDP;
patterns[2].spec = 0;
patterns[2].mask = 0;

/* GTP */
patterns[3].type = RTE_FLOW_ITEM_TYPE_GTPU;
patterns[3].spec = 0;
patterns[3].mask = 0;

/* GTP PSC */
patterns[4].type = RTE_FLOW_ITEM_TYPE_GTP_PSC;
gtp_psc_spec.pdu_type=  1;
gtp_psc_mask.pdu_type =  0xFF
```

```
patterns[4].spec = &gtp_psc_spec;
patterns[4].mask = &gtp_psc_mask;

/* IPv4 */
patterns[5].type = RTE_FLOW_ITEM_TYPE_IPV4;
patterns[5].spec = 0;
patterns[5].mask = 0;

/* UDP */
patterns[6].type = RTE_FLOW_ITEM_TYPE_UDP;
patterns[6].spec = 0;
patterns[6].mask = 0;

/* END the pattern array */
patterns[7].type = RTE_FLOW_ITEM_TYPE_END;
```

4.8.2 Actions

The Actions are the same as described in Section 4.7.2.

4.9 Route to Queue Group Based on DSCP

In this section, an RTE Flow Rule is created to match an IPv4 packet with a specific DSCP value and routed to one or more specified queues.

Note: IPv6 packet match with a specific DSCP is not supported.

4.9.1 Pattern Items

Table 21. Pattern Items to Match IPv4 Packet with a Specific DSCP

Index	Item	Spec	Mask
0	Ethernet	0	0
1	IPv4	hdr.type_of_service = dscp << 2	hdr.type_of_service = 0xFF
2	END	0	0

The following code sets up the *RTE_FLOW_ITEM_TYPE_ETH* and *RTE_FLOW_ITEM_TYPE_IPV4* pattern items. The *RTE_FLOW_ITEM_TYPE_IPV4* Pattern is configured to match on the DSCP value (in this case 8).

```
uint8_t dscp = 8;

struct rte_flow_item patterns[MAX_PATTERN_NUM];
struct rte_flow_item_ipv4 ipv4_spec;
struct rte_flow_item_ipv4 ipv4_mask;

/* Ethernet */
patterns[0].type = RTE_FLOW_ITEM_TYPE_ETH;
patterns[0].spec = 0;
patterns[0].mask = 0;

/* IPv4 */
patterns[1].type = RTE_FLOW_ITEM_TYPE_IPV4;
ipv4_spec.hdr.type_of_service = dscp << 2;
ipv4_mask.hdr.type_of_service = 0xFF;
```

```
patterns[1].spec = &ipv4_spec;
patterns[1].mask = &ipv4_mask;

/* END the pattern array */
patterns[2].type = RTE_FLOW_ITEM_TYPE_END
```

4.9.2 Actions

Table 22. RSS Actions for Queue Groups

Index	Action	Fields	Description	Value
0	RSS	rss_conf	types	0
			func	0
			key_len	0
		queue_num	Number of entries in queue[].	1, 2, 4, 8, etc.
		queue[]	Queue indices to use.	Must be contiguous (for example, 0,1,2,3).
1	END			

The following code sets up the action *RTE_FLOW_ACTION_TYPE_RSS* and calls the *rte_flow_create* function to create the RTE Flow Rule.

Note: The number of queues is this example is 4 and the queue indices are 0,1,2,3.

```
uint32_t num_queues = 4;
uint16_t queues[] = {0,1,2,3};

struct rte_flow *handle;
struct rte_flow_error err;
struct rte_flow_action actions[MAX_ACTION_NUM];
struct rte_flow_attr attributes = {.ingress = 1 };

struct rte_flow_action_rss action_rss = (struct rte_flow_action_rss){
    .queue_num =  num_queues,
    .queue= queues,
};

actions[0].type = RTE_FLOW_ACTION_TYPE_RSS;
actions[0].conf = &action_rss;

actions[1].type = RTE_FLOW_ACTION_TYPE_END;

handle = rte_flow_create (port_id, &attributes, patterns, actions, &err);
```

4.10 Route to Queue Group Based on QFI

In this section, an RTE Flow Rule is created to match a GTP-U Encapsulated packet with a specific QFI found within a GTP-U PDU Session Container Extension Header value and route it to one or more specified queues.

4.10.1 Pattern Items

Table 23. Pattern Items to Match GTP-U Encapsulated Packet

Index	Item	Spec	Mask
0	Ethernet	0	0
1	IPv4	0	0
2	UDP	0	0
3	GTPU	0	0
4	GTPU_PSC	pdu_type = 0/1	pdu_type = 0xFF
5	END	0	0

The following code sets up the *RTE_FLOW_ITEM_TYPE_ETH*, *RTE_FLOW_ITEM_TYPE_IPV4*, *RTE_FLOW_ITEM_TYPE_UDP*, *RTE_FLOW_ITEM_TYPE_GTP*, *RTE_FLOW_ITEM_TYPE_GTP_PSC*, and *RTE_FLOW_ITEM_TYPE_IPV4* pattern items. The *RTE_FLOW_ITEM_TYPE_GTP_PSC* Pattern is configured to match on the QFI value (in this case 9).

```
uint8_t qfi = 9;

struct rte_flow_item patterns[MAX_PATTERN_NUM];
struct rte_flow_item_gtp_psc gtp_psc_spec;
struct rte_flow_item_gtp_psc gtp_psc_mask;

/* Ethernet */
patterns[0].type = RTE_FLOW_ITEM_TYPE_ETH;
patterns[0].spec = 0;
patterns[0].mask = 0;

/* IPv4 */
patterns[1].type = RTE_FLOW_ITEM_TYPE_IPV4;
patterns[1].spec = 0;
patterns[1].mask = 0;

/* UDP */
patterns[2].type = RTE_FLOW_ITEM_TYPE_UDP;
patterns[2].spec = 0;
patterns[2].mask = 0;

/* GTP */
patterns[3].type = RTE_FLOW_ITEM_TYPE_GTPU;
patterns[3].spec = 0;
patterns[3].mask= 0;

/* GTP PSC */
patterns[4].type = RTE_FLOW_ITEM_TYPE_GTP_PSC;
gtp_psc_spec.qfi =  qfi & 0x3F;
gtp_psc_mask.qfi =  0xFF;
gtp_psc_spec.pdu_type=  1;
```

```
gtp_psc_mask.pdu_type =  0xFF
patterns[4].spec = &gtp_psc_spec;
patterns[4].mask = &gtp_psc_mask;

/* END the pattern array */
patterns[6].type = RTE_FLOW_ITEM_TYPE_END;
```

4.10.2 Actions

See Table 22 for the Action item descriptions.

The following code sets up the action *RTE_FLOW_ACTION_TYPE_RSS* and calls the *rte_flow_create* function to create the RTE Flow Rule. Note that the number of queues is this example is 8 and the queue indices are 4,5,6,7,8,9,10,11.

```
uint32_t num_queues = 8;
uint16_t queues[] = {4,5,6,7,8,9,10,11};

struct rte_flow *handle;
struct rte_flow_error err;
struct rte_flow_action actions[MAX_ACTION_NUM];
struct rte_flow_attr attributes = {.ingress = 1 };

struct rte_flow_action_rss action_rss = (struct rte_flow_action_rss){
    .queue_num =  num_queues,
    .queue= queues,
};

actions[0].type = RTE_FLOW_ACTION_TYPE_RSS;
actions[0].conf = &action_rss;
actions[1].type = RTE_FLOW_ACTION_TYPE_END;

handle = rte_flow_create (port_id, &attributes, patterns, actions, &err);
```

4.11 MPLS Protocol Extraction

The device can extract IP header's offset from a packet containing 0 and up to 5 MPLS headers into the Rx descriptor.

The PMD copies this data into the mbuf's metadata, then the application can use API rte_net_ice_dynf_proto_xtr_metadata_get to fetch the IP header offset value without parsing the packet headers. This helps to accelerate a specific workload.

To activate this feature, use **testpmd** with an initial parameter like the following:

```
-w 18:00.0,proto_xtr=(0-3):ip_offset
```

This enables the IP header's offset extraction for queue 0 to queue 3.

Note: This feature is enabled for *ice* PF driver in DPDK 20.08, and will be enabled for AVF driver in DPDK 20.11.

4.12 IPv6 Prefixes

RSS for IPv6 prefixes fields are supported in DPDK 20.11. A prefix can be specified instead of full IPv6 address for RSS.

In this section, an RTE Flow Rule is created to match an IPv6 packet.

4.12.1 Pattern Items

Table 24. Pattern Items to Match IPv6 Packet

Index	Item	Spec	Mask
0	Ethernet	0	0
1	IPv6	0	0
2	END	0	0

The following code sets up the *RTE_FLOW_ITEM_TYPE_ETH*. and *RTE_FLOW_ITEM_TYPE_IPV6* pattern items.

```
struct rte_flow_item patterns[MAX_PATTERN_NUM];

/* Ethernet */
patterns[0].type = RTE_FLOW_ITEM_TYPE_ETH;
patterns[0].spec = 0;
patterns[0].mask = 0;

/* IPv6 */
patterns[1].type = RTE_FLOW_ITEM_TYPE_IPV6;
patterns[1].spec = 0;
patterns[1].mask = 0;

/* END the pattern array */
patterns[2].type = RTE_FLOW_ITEM_TYPE_END;
```

4.12.2 Actions

Table 25. RSS Actions for IPv6 Prefix

Index	Action	Fields	Description	Value
0	RSS	rss_conf	types	ETH_RSS_IPV6 | RTE_ETH_RSS_L3_PRE64
			func	RTE_ETH_HASH_FUNCTION_SYMMETRIC_TOEPLITZ
			key_len	0
		queue_num	Number of entries in queue[].	NULL
		queue[]	Queue indices to use.	0
1	END			

The following code sets up the action *RTE_FLOW_ACTION_TYPE_RSS* and calls the *rte_flow_create* function to create the RTE Flow rule. The prefix includes the first 64 bits of the Source and Destination IPv6 address.

```
struct rte_flow *handle;
struct rte_flow_error err;
struct rte_flow_action actions[MAX_ACTION_NUM];
struct rte_flow_attr attributes = {.ingress = 1 };

struct rte_flow_action_rss action_rss = (struct rte_flow_action_rss){
              .types =   ETH_RSS_IPV6 | RTE_ETH_RSS_L3_PRE64
              .func =  RTE_ETH_HASH_FUNCTION_SYMMETRIC_TOEPLITZ,
              .key_len = 0,
 };

actions[0].type = RTE_FLOW_ACTION_TYPE_RSS;
actions[0].conf = &action_rss;

actions[1].type = RTE_FLOW_ACTION_TYPE_END;

handle = rte_flow_create (port_id, &attributes, patterns, actions, &err);
```

Note: In DPDK 20.11, 32-bit (*RTE_ETH_RSS_L3_PRE32*), 48-bit (*RTE_ETH_RSS_L3_PRE48*) and 64-bit (*RTE_ETH_RSS_L3_PRE64*) IPv6 prefixes are supported on PF and only 64-bit (*RTE_ETH_RSS_L3_PRE64*) IPV6 prefixes are supported on VF.

5.0 SR-IOV

From DPDK 20.05, VFs are not supported by the DPDK PF PMD. The ICE Kernel driver is required.

5.1 Intel® Ethernet Flow Director and RSS Filters

From DPDK 20.05, programming of the Intel® Ethernet Flow Director and RSS filters via the DPDK RTE FLOW rules is supported on VFs using the Intel Advanced Virtual Function (IAVF) PMD.

The RSS and Intel® Ethernet Flow Director rules outlined in Section 4.0 for use via the PF PMD are also applicable for use with the IAVF PMD with the following exceptions on the AVF for RSS.

- For GTP encapsulated packets, it is not possible to set the Inset Hash for the Inner Source and Destination ports. This means that directing GTP encapsulated packets based on their inner 5-tuple flow is not possible. At the time of writing there is no patch available to address this issue. Contact Intel for further information.
- It is not possible to set the Inset Hash to Destination IP only or Destination Port only. There is a DPDK and ICE Kernel Driver patch available to address this limitation. Contact Intel for further information.

5.2 Switch Filters

The DPDK 20.05 PF PMD does not support the programming of Switch Filters on the E810 for VFs. This must now be done via a combination of a new Device Config Function and the ICE Kernel PF driver.

The DCF is a special DPDK PMD that provides an interface into the Kernel PF driver to enable advanced DDP capabilities of the E810 to direct packets to one or more VFs based on Switch Rules (for example, direct packets all received packets with VLAN ID 2 to VF 1).

Only a single DCF Entity associated with a trusted VF is allowed per E810 port. The DCF negotiates DCF capabilities with the Kernel PF driver.

Figure 1 shows an SDN agent etc running on a host/container/VM utilizing the DCF PMD to program switch filters to steer packets to a VM/Container consuming a VF. The DPDK IAVF can create Intel® Ethernet Flow Director/RSS Filter rules for packets landing at the VF.

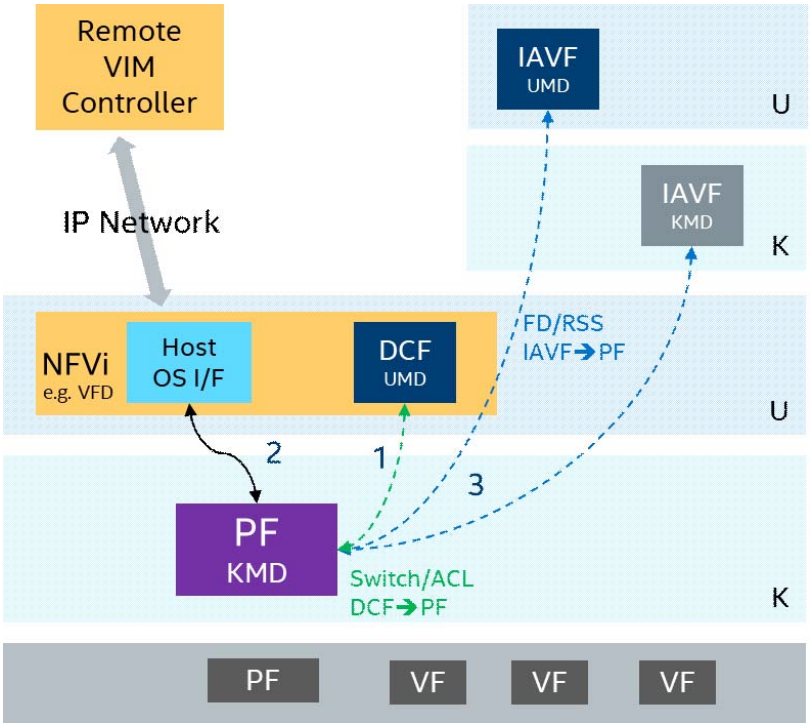

Figure 1. Overall Software Infrastructure

Refer to DPDK ICE Poll Mode Driver for further information on the DCF.

Note: DCF is reliant on ICE Kernel PF Driver Version 0.16.0 or later.

5.2.1 Device and Function Configuration

1. Create a number of VFs (for example, four).

   ```
   #echo 4 > /sys/bus/pci/devices/0000\:18\:00.0/sriov_numvfs
   ```

2. Enable VF0 as the trusted VF:

   ```
   #ip link set dev enp24s0f0 vf 0 trust on
   ```

3. Bind VF0 by running testpmd (or DPDK application) with 'cap=dcf' devarg:

   ```
   #testpmd -l 22-25 -n 4 -w 18:01.0,cap=dcf -- -i
   ```

5.3 Programming Switch Filters on DCF

Programming the Switch filters from the DCF via the DPDK RTE Flow API is similar to the programming of the Flow Director and RSS Filters described in Section 4.0.

Table 26 through Table 28 show the pattern types and input sets available to be programmed via RTE FLOW API for the Switch Filter. Shading in green indicates Comms DDP Package-specific support (which requires support from the DPDK driver), while no color indicates fields that are also supported by the OS-default package.

Table 26. Patterns and Input Sets for Switch Filter

Pattern	Input Set
ETH	S-MAC | S-MAC | ETHERTYPE
ETH / VLAN	S-MAC | S-MAC | VLAN | ETHERTYPE
ETH / IPv4(6)	D-IP | S-IP | PROTO | TOS | TTL
ETH / IPv4(6) / UDP	D-IP | S-IP | PROTO | TOS | TTL | D-PORT | S-PORT
ETH / IPv4(6) / TCP	D-IP | S-IP | PROTO | TOS | TTL | D-PORT | S-PORT
ETH / IPv4 / UDP / VLAN(NVGRE) / IPv4	VNI(TNI) | Inner D-MAC | Inner D-IP | Inner S-IP
ETH / IPv4 / UDP / VLAN(NVGRE) / IPv4 / UDP	VNI(TNI) | Inner D-MAC | Inner D-IP | Inner S-IP | Inner S-PORT | Inner D-PORT
ETH / IPv4 / UDP / VLAN(NVGRE) / IPv4 / TCP	VNI(TNI) | Inner D-MAC | Inner D-IP | Inner S-IP | Inner S-PORT | Inner D-PORT
ETH / PPPOE_D	D-MAC | VLAN | ETHERTYPE | PPPOE_Session
ETH / PPPOE_S	D-MAC | VLAN | ETHERTYPE | PPPOE_Session
ETH / PPPOE_S_PROTO	D-MAC | VLAN | ETHERTYPE | PPPOE_Session | PPPOE_PROTO
ETH / IPv4(6) / AH	SPI
ETH / IPv4(6) / ESP	SPI
ETH / IPv4(6) / UDP / AH	SPI
ETH / IPv4(6) / UDP / ESP	SPI
ETH / IPv4(6) / PFCP (S_FIELD = 1)	SEID
ETH / IPv4(6) / L2TP	SESSION ID

Table 27. Patterns and Input Sets for ACL Filter for 4-Port Devices (DPDK 20.08/20.11)

Pattern	Input Set
ETH / IPv4	D-IP | S-IP
ETH / IPv4 / UDP	S-IP | D-IP | S-PORT | D-PORT
ETH / IPv4 / UDP / TCP	S-IP | D-IP | S-PORT | D-PORT
ETH / IPv4 / UDP / SCTP	S-IP | D-IP | S-PORT | D-PORT

Table 28. Patterns and Input Sets for ACL Filter for 2-Port Devices (DPDK 20.08/20.11)

Pattern	Input Set
ETH / IPv4	S-MAC | D-MAC | D-IP | S-IP
ETH / IPv4 / UDP	S-MAC | D-MAC | D-IP | S-IP | S-PORT | D-PORT
ETH / IPv4 / UDP / TCP	S-MAC | D-MAC | D-IP | S-IP | S-PORT | D-PORT
ETH / IPv4 / UDP / SCTP	S-MAC | D-MAC | D-IP | S-IP | S-PORT | D-PORT

Notes:

- ACL filter only supports “Drop” action.
- ACL filter will only offload rules that need wild card match. For example, #flow create 0 ingress pattern eth / ipv4 src spec 192.168.0.1 src mask 255.255.255.0 / end actions drop / end
- Due to resource limitation, 4-port and 2-port devices have different input set support.

5.3.1 Pattern Items

For example, an RTE Flow Rule is created to match a packet with a VLAN ID = 2 and directs that packet to VF 1.

Table 29. Pattern Items to match VLAN

Index	Item	Spec	Mask
0	ETH	o	o
1	VLAN	tci = 2	tci = 0x0Fff
2	END	0	0

The following code sets up the RTE_FLOW_ITEM_TYPE_ETH and RTE_FLOW_ITEM_TYPE_VLAN pattern items. The RTE_FLOW_ITEM_TYPE_VLAN Pattern is configured to match on the *tci* value 2.

```
uint8_t vid = 2;

struct rte_flow_item patterns[MAX_PATTERN_NUM];
struct struct rte_flow_item_vlan vlan_spec;
struct struct rte_flow_item_vlan vlan_mask;

 /* Ethernet */
patterns[0].type = RTE_FLOW_ITEM_TYPE_ETH;
patterns[0].spec = 0;
patterns[0].mask = 0;

/* VLAN */
patterns[1].type = RTE_FLOW_ITEM_TYPE_VLAN;
vlan_spec.tci =  vid  & 0x0FFF;
vlan_mask.tci =  0x0FFF
patterns[1].spec = &vlan_spec;
patterns[1].mask = &vlan_mask;

 /* END the pattern array */
 patterns[2].type = RTE_FLOW_ITEM_TYPE_END;
```

5.3.2 Actions

Table 30. Switch Filter Actions for VF

Index	Action	Fields	Description	Value
0	VF	vf_id	Target VF ID	1
1	END			

The following code sets up the action RTE_FLOW_ACTION_TYPE_VF and calls the *rte_flow_create* function to create the RTE Flow Rule.

```
uint32_t vf_id = 1;

struct rte_flow *handle;
struct rte_flow_error err;
struct rte_flow_action actions[MAX_ACTION_NUM];
struct rte_flow_attr attributes = {.ingress = 1 };

struct rte_flow_action_vf action_vf;
```

```
action_vf.id = vf_id;

actions[0].type = RTE_FLOW_ACTION_TYPE_VF;
actions[0].conf = &action_vf;

actions[1].type = RTE_FLOW_ACTION_TYPE_END;

handle = rte_flow_create (port_id, &attributes, patterns, actions, &err);
```

6.0 Intel® Ethernet 800 Series Features

Table 31. Basic Features

Feature	DPDK Version							
	19.11		20.05		20.08		20.11	
	PF Mode	VF Mode	PF Mode	VF Mode	PF Mode	VF Mode	PF Mode	VF Mode
Speed Capabilities	Y	Y	Y	Y	Y	Y	Y	Y
Link Status	Y	Y	Y	Y	Y	Y	Y	Y
Link Status Event	Y	N	Y	N	Y	N	Y	N
Rx Interrupt	Y	Y	Y	Y	Y	Y	Y	Y
Fast mbuf Free	Y	N	P	N	P	N	P	N
Queue Start/Stop	Y	Y	Y	Y	Y	Y	Y	Y
Burst Mode Info	Y	N	Y	N	Y	N	Y	N
MTU Update	Y	Y	Y	Y	Y	Y	Y	Y
Jumbo Frame	Y	Y	Y	Y	Y	Y	Y	Y
Scattered Rx	Y	Y	Y	Y	Y	Y	Y	Y
TSO	Y	Y	Y	Y	Y	Y	Y	Y
Promiscuous Mode	Y	Y	Y	Y	Y	Y	Y	Y
Allmulticast Mode	Y	Y	Y	Y	Y	Y	Y	Y
Static RSS Hash	Y	N	Y	N	Y	N	Y	N
Unicast MAC Filter	Y	Y	Y	Y	Y	Y	Y	Y
RSS Hash	Y	Y	Y	Y	Y	Y	Y	Y
RSS Key Update	Y	Y	Y	Y	Y	Y	Y	Y
RSS reta Update	Y	Y	Y	Y	Y	Y	Y	Y
VLAN Filter	Y	Y	Y	Y	Y	Y	Y	Y
CRC Offload	Y	Y	Y	Y	Y	Y	Y	Y
VLAN Offload	Y	Y	Y	Y	Y	Y	Y	Y
Q in Q Offload	Y	N	P	N	P	N	P	N
L3 Checksum Offload	Y	Y	P	P	P	P	P	P
L4 Checksum Offload	Y	Y	P	P	P	P	P	P
Inner L3 Checksum	N	N	P	N	P	N	P	N
Inner L4 Checksum	N	N	P	N	P	N	P	N
Packet Type Parsing	Y	Y	Y	Y	Y	Y	Y	Y
Rx Descriptor Status	Y	Y	Y	Y	Y	Y	Y	Y
Tx Descriptor Status	Y	Y	Y	Y	Y	Y	Y	Y
Basic Stats	Y	Y	Y	Y	Y	Y	Y	Y
Extended Stats	Y	N	Y	N	Y	N	Y	N
Firmware Version	Y	N	Y	N	Y	N	Y	N
Module EEPROM Dump	Y	N	Y	N	Y	N	Y	N

Table 31. Basic Features [continued]

Feature	DPDK Version							
	19.11		20.05		20.08		20.11	
	PF Mode	VF Mode	PF Mode	VF Mode	PF Mode	VF Mode	PF Mode	VF Mode
Multi-Process Aware	N	Y	Y	Y	Y	Y	Y	Y
BSD nic_uio	Y	Y	Y	Y	Y	Y	Y	Y
Linux UIO	Y	Y	Y	Y	Y	Y	Y	Y
Linux VFIO	Y	Y	Y	Y	Y	Y	Y	Y
x86-32	Y	Y	Y	Y	Y	Y	Y	Y
x86-64	Y	Y	Y	Y	Y	Y	Y	Y

Note: P = Partial support

Table 32. Advanced Features

Feature	DPDK Version							
	19.11		20.05		20.08		20.11	
	PF Mode	VF Mode	PF Mode	VF Mode	PF Mode	VF Mode	PF Mode	VF Mode
RTE-FLOW API	Y	N	Y	Y	Y	Y	Y	Y
GTPU (EH)	Y	N	Y	Y	Y	Y	Y	Y
PPPOE	Y	N	Y	N	Y	N	Y	N
PFCP	N	N	N	Y	Y	Y	Y	Y
ESP/AH	N	N	N	Y	Y	Y	Y	Y
L2TPv3	N	N	N	Y	Y	Y	Y	Y
GTPU (5 Tuple Hash)	N	N	N	N	Y	Y	Y	Y
GTPU (w/o EH)	N	N	N	N	Y	Y	Y	Y
Symmetric Hash	N	N	N	N	Y	Y	Y	Y
Queue Numbers/PF	64	16	64	16	64	16	64	256
Separate Configure for Outer IPv4/IPv6 GTPU	N	N	N	N	N	N	Y	Y
IPv6 Hash Mask	N	N	Y	N	Y	N	Y	Y
Multicast MAC Filter	N	N	N	N	N	N	N	Y

LEGAL