



DPDK

DATA PLANE DEVELOPMENT KIT

Network Interface Controller Drivers

Release 19.11.3

Jun 18, 2020

1	Overview of Networking Drivers	1
2	Features Overview	4
2.1	Speed capabilities	4
2.2	Link status	4
2.3	Link status event	4
2.4	Removal event	5
2.5	Queue status event	5
2.6	Rx interrupt	5
2.7	Lock-free Tx queue	5
2.8	Fast mbuf free	5
2.9	Free Tx mbuf on demand	6
2.10	Queue start/stop	6
2.11	MTU update	6
2.12	Jumbo frame	6
2.13	Scattered Rx	6
2.14	LRO	7
2.15	TSO	7
2.16	Promiscuous mode	7
2.17	Allmulticast mode	8
2.18	Unicast MAC filter	8
2.19	Multicast MAC filter	8
2.20	RSS hash	8
2.21	Inner RSS	8
2.22	RSS key update	9
2.23	RSS reta update	9
2.24	VMDq	9
2.25	SR-IOV	9
2.26	DCB	9
2.27	VLAN filter	10
2.28	Flow control	10
2.29	Flow API	10
2.30	Rate limitation	10
2.31	Traffic mirroring	10
2.32	Inline crypto	11
2.33	CRC offload	11
2.34	VLAN offload	11
2.35	QinQ offload	11
2.36	L3 checksum offload	12

2.37	L4 checksum offload	12
2.38	Timestamp offload	12
2.39	MACsec offload	13
2.40	Inner L3 checksum	13
2.41	Inner L4 checksum	13
2.42	Packet type parsing	14
2.43	Timesync	14
2.44	Rx descriptor status	14
2.45	Tx descriptor status	14
2.46	Basic stats	14
2.47	Extended stats	15
2.48	Stats per queue	15
2.49	FW version	15
2.50	EEPROM dump	15
2.51	Module EEPROM dump	15
2.52	Registers dump	16
2.53	LED	16
2.54	Multiprocess aware	16
2.55	BSD nic_uio	16
2.56	Linux UIO	16
2.57	Linux VFIO	16
2.58	Other kdrv	16
2.59	ARMv7	16
2.60	ARMv8	17
2.61	Power8	17
2.62	x86-32	17
2.63	x86-64	17
2.64	Usage doc	17
2.65	Design doc	17
2.66	Perf doc	17
2.67	Runtime Rx queue setup	18
2.68	Runtime Tx queue setup	18
2.69	Burst mode info	18
2.70	Other dev ops not represented by a Feature	18
3	Compiling and testing a PMD for a NIC	19
3.1	Driver Compilation	19
3.2	Running testpmd in Linux	20
4	AF_PACKET Poll Mode Driver	22
4.1	Options and inherent limitations	22
4.2	Prerequisites	23
4.3	Set up an af_packet interface	23
5	AF_XDP Poll Mode Driver	24
5.1	Options	24
5.2	Prerequisites	24
5.3	Set up an af_xdp interface	25
6	ARK Poll Mode Driver	26
6.1	Overview	26
6.2	Device Parameters	27

6.3	Data Path Interface	27
6.4	Configuration Information	27
6.5	Building DPDK	28
6.6	Supported ARK RTL PCIe Instances	28
6.7	Supported Operating Systems	28
6.8	Supported Features	28
6.9	Unsupported Features	28
6.10	Pre-Requisites	29
6.11	Usage Example	29
7	Aquantia Atlantic DPDK Driver	30
7.1	Supported features	30
7.2	Experimental API features	30
7.3	Configuration Information	30
7.3.1	Application Programming Interface	31
7.3.2	Limitations or Known issues	31
7.3.3	Supported Chipsets and NICs	31
8	AVP Poll Mode Driver	32
8.1	Features and Limitations of the AVP PMD	32
8.2	Prerequisites	33
8.3	Launching a VM with an AVP type network attachment	33
9	AXGBE Poll Mode Driver	34
9.1	Supported Features	34
9.2	Configuration Information	34
9.3	Building DPDK	35
9.4	Prerequisites and Pre-conditions	35
9.5	Usage Example	35
10	BNX2X Poll Mode Driver	36
10.1	Supported Features	36
10.2	Non-supported Features	36
10.3	Co-existence considerations	36
10.4	Supported QLogic NICs	37
10.5	Prerequisites	37
10.6	Pre-Installation Configuration	37
10.6.1	Config File Options	37
10.7	Driver compilation and testing	37
10.8	SR-IOV: Prerequisites and sample Application Notes	38
11	BNXT Poll Mode Driver	40
11.1	BNXT PMD Features	40
11.2	BNXT Vector PMD	41
11.2.1	RX Requirements for Vector Mode	41
11.2.2	TX Requirements for Vector Mode	41
11.3	BNXT PMD Supported Chipsets and Adapters	41
12	CXGBE Poll Mode Driver	44
12.1	Features	44
12.2	Limitations	44
12.3	Supported Chelsio T5 NICs	45

12.4	Supported Chelsio T6 NICs	45
12.5	Supported SR-IOV Chelsio NICs	45
12.6	Prerequisites	45
12.7	Pre-Installation Configuration	45
12.7.1	Config File Options	45
12.7.2	Runtime Options	46
12.8	Driver compilation and testing	46
12.9	Linux	46
12.9.1	Linux Installation	46
12.9.2	Running testpmd	47
12.9.3	Configuring SR-IOV Virtual Functions	48
12.10	FreeBSD	49
12.10.1	FreeBSD Installation	49
12.10.2	Running testpmd	50
12.11	Sample Application Notes	52
12.11.1	Enable/Disable Flow Control	52
12.11.2	Jumbo Mode	52
13	DPAA Poll Mode Driver	53
13.1	NXP DPAA (Data Path Acceleration Architecture - Gen 1)	53
13.1.1	DPAA Overview	53
13.2	DPAA DPDK - Poll Mode Driver Overview	54
13.2.1	DPAA Bus driver	54
13.2.2	DPAA NIC Driver (PMD)	55
13.2.3	DPAA Mempool Driver	55
13.3	Whitelisting & Blacklisting	55
13.4	Supported DPAA SoCs	55
13.5	Prerequisites	56
13.6	Pre-Installation Configuration	56
13.6.1	Config File Options	56
13.6.2	Environment Variables	56
13.7	Driver compilation and testing	57
13.8	Limitations	57
13.8.1	Platform Requirement	57
13.8.2	Maximum packet length	57
13.8.3	Multiprocess Support	57
14	DPAA2 Poll Mode Driver	58
14.1	NXP DPAA2 (Data Path Acceleration Architecture Gen2)	58
14.1.1	DPAA2 Overview	58
14.1.2	Overview of DPAA2 Objects	59
14.1.3	DPAA2 Objects for an Ethernet Network Interface	60
14.1.4	Object Connections	61
14.1.5	Interrupts	62
14.2	DPAA2 DPDK - Poll Mode Driver Overview	62
14.2.1	DPAA2 bus driver	62
14.2.2	DPIO driver	62
14.2.3	DPBP based Mempool driver	63
14.2.4	DPAA2 NIC Driver	63
14.3	Supported DPAA2 SoCs	63
14.4	Prerequisites	64
14.5	Pre-Installation Configuration	64

14.5.1	Config File Options	64
14.6	Driver compilation and testing	64
14.7	Enabling logs	65
14.8	Whitelisting & Blacklisting	65
14.9	Limitations	65
14.9.1	Platform Requirement	65
14.9.2	Maximum packet length	66
14.9.3	Other Limitations	66
15	Driver for VM Emulated Devices	67
15.1	Validated Hypervisors	67
15.2	Recommended Guest Operating System in Virtual Machine	67
15.3	Setting Up a KVM Virtual Machine	67
15.4	Known Limitations of Emulated Devices	69
16	ENA Poll Mode Driver	70
16.1	Overview	70
16.2	Management Interface	70
16.3	Data Path Interface	71
16.4	Configuration information	71
16.5	Building DPDK	72
16.6	Supported ENA adapters	72
16.7	Supported Operating Systems	72
16.8	Supported features	72
16.9	Prerequisites	72
16.10	Usage example	73
17	ENETC Poll Mode Driver	74
17.1	ENETC	74
17.1.1	ENETC Overview	74
17.1.2	ENETC Features	74
17.1.3	NIC Driver (PMD)	75
17.1.4	Supported ENETC SoCs	75
17.1.5	Prerequisites	75
17.1.6	Driver compilation and testing	76
18	ENIC Poll Mode Driver	77
18.1	How to obtain ENIC PMD integrated DPDK	77
18.2	Configuration information	77
18.3	SR-IOV mode utilization	78
18.4	Generic Flow API support	80
18.5	Overlay Offload	81
18.6	Ingress VLAN Rewrite	82
18.7	Vectorized Rx Handler	82
18.8	Limitations	83
18.9	How to build the suite	84
18.10	Supported Cisco VIC adapters	84
18.11	Supported Operating Systems	84
18.12	Supported features	84
18.13	Known bugs and unsupported features in this release	85
18.14	Prerequisites	85
18.15	Additional Reference	86

18.16	Contact Information	86
19	FM10K Poll Mode Driver	87
19.1	FTAG Based Forwarding of FM10K	87
19.2	Vector PMD for FM10K	87
19.2.1	RX Constraints	87
19.2.2	TX Constraint	88
19.3	Limitations	88
19.3.1	Switch manager	88
19.3.2	Support for Switch Restart	89
19.3.3	CRC stripping	89
19.3.4	Maximum packet length	89
19.3.5	Statistic Polling Frequency	89
19.3.6	Interrupt mode	89
20	HINIC Poll Mode Driver	90
20.1	Features	90
20.2	Prerequisites	91
20.3	Pre-Installation Configuration	91
20.3.1	Config File Options	91
20.4	Driver compilation and testing	91
20.5	Limitations or Known issues	91
21	HNS3 Poll Mode Driver	92
21.1	Features	92
21.2	Prerequisites	92
21.3	Pre-Installation Configuration	93
21.3.1	Config File Options	93
21.4	Driver compilation and testing	93
21.5	Limitations or Known issues	93
22	I40E Poll Mode Driver	94
22.1	Features	94
22.2	Prerequisites	95
22.3	Recommended Matching List	95
22.4	Pre-Installation Configuration	96
22.4.1	Config File Options	96
22.4.2	Runtime Config Options	96
22.4.3	Vector RX Pre-conditions	97
22.5	Driver compilation and testing	97
22.6	SR-IOV: Prerequisites and sample Application Notes	97
22.7	Sample Application Notes	98
22.7.1	Vlan filter	98
22.7.2	Flow Director	98
22.7.3	Floating VEB	100
22.7.4	Dynamic Device Personalization (DDP)	100
22.7.5	Input set configuration	101
22.7.6	Queue region configuration	101
22.8	Limitations or Known issues	101
22.8.1	MPLS packet classification	101
22.8.2	16 Byte RX Descriptor setting on DPDK VF	102
22.8.3	Receive packets with Ethertype 0x88A8	102

22.8.4	Incorrect Rx statistics when packet is oversize	102
22.8.5	VF & TC max bandwidth setting	102
22.8.6	TC TX scheduling mode setting	102
22.8.7	VF performance is impacted by PCI extended tag setting	103
22.8.8	Vlan strip of VF	103
22.8.9	DCB function	103
22.8.10	Global configuration warning	103
22.9	High Performance of Small Packets on 40GbE NIC	103
22.9.1	Use 16 Bytes RX Descriptor Size	104
22.9.2	Input set requirement of each pctype for FDIR	104
22.10	Example of getting best performance with l3fwd example	104
22.10.1	Tx bytes affected by the link status change	105
23	ICE Poll Mode Driver	106
23.1	Prerequisites	106
23.2	Recommended Matching List	106
23.3	Pre-Installation Configuration	106
23.3.1	Config File Options	106
23.3.2	Runtime Config Options	107
23.4	Driver compilation and testing	109
23.5	Features	109
23.5.1	Vector PMD	109
23.5.2	Malicious driver detection (MDD)	109
23.6	Sample Application Notes	109
23.6.1	Vlan filter	109
23.7	Limitations or Known issues	109
23.7.1	19.02 limitation	110
24	IFCVF vDPA driver	111
24.1	Pre-Installation Configuration	111
24.1.1	Config File Options	111
24.2	IFCVF vDPA Implementation	111
24.2.1	Key IFCVF vDPA driver ops	111
24.2.2	To create a vhost port with IFC VF	112
24.3	Features	112
24.4	Prerequisites	112
24.5	Limitations	112
24.5.1	Dependency on vfio-pci	112
24.5.2	Live Migration with VIRTIO_NET_F_GUEST_ANNOUNCE	112
25	IGB Poll Mode Driver	113
25.1	Features	113
25.2	Limitations or Known issues	113
25.3	Supported Chipsets and NICs	113
26	IPN3KE Poll Mode Driver	114
26.1	Prerequisites	114
26.2	Pre-Installation Configuration	114
26.2.1	Config File Options	114
26.2.2	Runtime Config Options	114
26.3	Driver compilation and testing	115
26.4	Sample Application Notes	115

26.4.1	Packet TX/RX with FPGA Pass-through image	115
26.4.2	HQoS and flow acceleration	115
26.5	Limitations or Known issues	115
26.5.1	19.05 limitation	115
27	IXGBE Driver	116
27.1	Vector PMD for IXGBE	116
27.1.1	RX Constraints	116
27.1.2	TX Constraint	118
27.2	Application Programming Interface	118
27.3	Sample Application Notes	118
27.3.1	l3fwd	118
27.3.2	load_balancer	118
27.4	Limitations or Known issues	118
27.4.1	Malicious Driver Detection not Supported	118
27.4.2	Statistics	119
27.4.3	MTU setting	119
27.4.4	VF MAC address setting	119
27.4.5	X550 does not support legacy interrupt mode	119
27.5	Inline crypto processing support	120
27.6	Virtual Function Port Representors	120
27.7	Supported Chipsets and NICs	120
28	Intel Virtual Function Driver	122
28.1	SR-IOV Mode Utilization in a DPDK Environment	122
28.1.1	Physical and Virtual Function Infrastructure	124
28.1.2	Validated Hypervisors	128
28.1.3	Expected Guest Operating System in Virtual Machine	128
28.2	Setting Up a KVM Virtual Machine Monitor	128
28.3	DPDK SR-IOV PMD PF/VF Driver Usage Model	132
28.3.1	Fast Host-based Packet Processing	132
28.4	SR-IOV (PF/VF) Approach for Inter-VM Communication	132
29	KNI Poll Mode Driver	135
29.1	Usage	135
29.2	Default interface configuration	135
29.3	PMD arguments	136
29.4	PMD log messages	136
29.5	PMD testing	136
30	LiquidIO VF Poll Mode Driver	138
30.1	Supported LiquidIO Adapters	138
30.2	Pre-Installation Configuration	138
30.3	SR-IOV: Prerequisites and Sample Application Notes	139
30.4	Limitations	140
30.4.1	VF MTU	140
30.4.2	VLAN offload	140
30.4.3	Ring size	140
30.4.4	CRC stripping	140
31	Memif Poll Mode Driver	141
31.1	Shared memory	142

31.2	Zero-copy slave	143
31.2.1	Example: testpmd	144
31.2.2	Example: testpmd and VPP	144
32	MLX4 poll mode driver library	146
32.1	Implementation details	146
32.2	Configuration	147
32.2.1	Compilation options	147
32.2.2	Environment variables	147
32.2.3	Run-time configuration	147
32.2.4	Kernel module parameters	148
32.3	Limitations	148
32.4	Prerequisites	149
32.4.1	Current RDMA core package and Linux kernel (recommended)	149
32.4.2	Mellanox OFED as a fallback	150
32.5	Quick Start Guide	150
32.6	Performance tuning	151
32.7	Usage example	152
33	MLX5 poll mode driver	154
33.1	Design	154
33.2	Features	155
33.3	Limitations	156
33.4	Statistics	158
33.5	Configuration	158
33.5.1	Compilation options	158
33.5.2	Environment variables	159
33.5.3	Run-time configuration	159
33.5.4	Firmware configuration	166
33.6	Prerequisites	167
33.6.1	Installation	167
33.7	Supported NICs	169
33.8	Quick Start Guide on OFED/EN	170
33.9	Enable switchdev mode	170
33.10	Performance tuning	171
33.11	Supported hardware offloads	172
33.12	Notes for metadata	174
33.13	Notes for testpmd	174
33.14	Usage example	174
34	MVNETA Poll Mode Driver	177
34.1	Features	177
34.2	Limitations	178
34.3	Prerequisites	178
34.4	Pre-Installation Configuration	178
34.4.1	Config File Options	178
34.4.2	Runtime options	178
34.5	Building DPDK	179
34.6	Usage Example	179
35	MVPP2 Poll Mode Driver	180
35.1	Features	180

35.2	Limitations	181
35.3	Prerequisites	181
35.4	Config File Options	182
35.5	Building DPDK	182
35.6	Usage Example	182
35.7	Extended stats	183
35.8	QoS Configuration	183
35.8.1	Configuration syntax	183
35.9	Flow API	186
35.9.1	Supported flow actions	186
35.9.2	Supported flow items	186
35.9.3	Classifier match engine	187
35.9.4	Flow rules usage example	187
35.9.5	Limitations	188
35.10	Traffic metering and policing	188
35.10.1	Limitations	189
35.10.2	Usage example	189
35.11	Traffic Management API	189
35.11.1	Limitations	190
35.11.2	Usage example	190
36	Netvsc poll mode driver	193
36.1	Features and Limitations of Hyper-V PMD	193
36.2	Installation	193
36.3	Prerequisites	194
36.4	Netvsc PMD arguments	194
37	NFB poll mode driver library	195
37.1	Prerequisites	195
37.1.1	Versions of the packages	195
37.2	Configuration	196
37.3	Using the NFB PMD	196
37.4	NFB card architecture	196
37.5	Limitations	196
37.6	Example of usage	197
38	NFP poll mode driver library	198
38.1	Dependencies	198
38.2	Building the software	198
38.3	Driver compilation and testing	199
38.4	Using the PF	199
38.5	PF multiport support	199
38.6	PF multiprocess support	200
38.7	System configuration	200
39	OCTEON TX Poll Mode driver	201
39.1	Features	201
39.2	Supported OCTEON TX SoCs	201
39.3	Unsupported features	201
39.4	Prerequisites	202
39.5	Pre-Installation Configuration	202
39.5.1	Config File Options	202

39.5.2	Driver compilation and testing	202
39.6	Initialization	203
39.6.1	Device arguments	203
39.6.2	Dependency	203
39.7	Limitations	203
39.7.1	octeontx_fpvf external mempool handler dependency	203
39.7.2	CRC stripping	203
39.7.3	Maximum packet length	204
39.7.4	Maximum mempool size	204
40	OCTEON TX2 Poll Mode driver	205
40.1	Features	205
40.2	Prerequisites	206
40.3	Compile time Config Options	206
40.4	Driver compilation and testing	206
40.5	Runtime Config Options	207
40.6	Limitations	208
40.6.1	mempool_octeontx2 external mempool handler dependency	208
40.6.2	CRC stripping	208
40.6.3	Multicast MAC filtering	208
40.6.4	SDP interface support	208
40.7	Debugging Options	209
40.8	RTE Flow Support	209
41	PFE Poll Mode Driver	211
41.1	PFE	211
41.1.1	PFE Overview	211
41.1.2	PFE Features	212
41.1.3	Supported PFE SoCs	212
41.1.4	Prerequisites	213
41.1.5	Driver compilation and testing	213
41.1.6	Limitations	213
42	QEDE Poll Mode Driver	214
42.1	Supported Features	214
42.2	Non-supported Features	215
42.3	Co-existence considerations	215
42.4	Supported QLogic Adapters	215
42.5	Prerequisites	215
42.5.1	Performance note	216
42.5.2	Config File Options	216
42.5.3	Config notes	216
42.6	Driver compilation and testing	216
42.7	RTE Flow Support	217
42.8	SR-IOV: Prerequisites and Sample Application Notes	217
43	Solarflare libefx-based Poll Mode Driver	220
43.1	Features	220
43.2	Non-supported Features	221
43.3	Limitations	221
43.3.1	Equal stride super-buffer mode	221
43.4	Tunnels support	221

43.5	Flow API support	222
43.5.1	Ethernet destination individual/group match	223
43.5.2	Exceptions to flow rules	223
43.6	Supported NICs	223
43.7	Prerequisites	224
43.8	Pre-Installation Configuration	224
43.8.1	Config File Options	224
43.8.2	Per-Device Parameters	224
43.8.3	Dynamic Logging Parameters	225
44	Soft NIC Poll Mode Driver	227
44.1	Flow	227
44.2	Supported Operating Systems	228
44.3	Build options	228
44.4	Soft NIC PMD arguments	228
44.5	Soft NIC testing	228
44.6	Soft NIC Firmware	230
44.7	QoS API Support:	230
44.8	Flow API support:	231
44.8.1	Example:	231
45	SZEDATA2 poll mode driver library	233
45.1	Prerequisites	233
45.1.1	Versions of the packages	234
45.2	Configuration	234
45.3	Using the SZEDATA2 PMD	234
45.4	NFB card architecture	234
45.5	Limitations	234
45.6	Example of usage	235
46	Tun/Tap Poll Mode Driver	236
46.1	Flow API support	237
46.1.1	Examples of testpmd flow rules	238
46.2	Multi-process sharing	238
46.3	Example	238
46.4	RSS specifics	239
46.5	Systems supporting flow API	240
47	ThunderX NICVF Poll Mode Driver	241
47.1	Features	241
47.2	Supported ThunderX SoCs	241
47.3	Prerequisites	242
47.4	Pre-Installation Configuration	242
47.4.1	Config File Options	242
47.5	Driver compilation and testing	242
47.6	Linux	242
47.6.1	SR-IOV: Prerequisites and sample Application Notes	242
47.6.2	Multiple Queue Set per DPDK port configuration	244
47.6.3	Example device binding	245
47.7	Module params	246
47.7.1	skip_data_bytes	246
47.8	Limitations	246

47.8.1	CRC stripping	246
47.8.2	Maximum packet length	246
47.8.3	Maximum packet segments	246
47.8.4	skip_data_bytes	246
48	VDEV_NETVSC driver	247
48.1	Implementation details	247
48.2	Build options	248
48.3	Run-time parameters	248
49	Poll Mode Driver for Emulated Virtio NIC	249
49.1	Virtio Implementation in DPDK	249
49.2	Features and Limitations of virtio PMD	249
49.3	Prerequisites	250
49.4	Virtio with kni vhost Back End	250
49.5	Virtio with qemu virtio Back End	252
49.6	Virtio PMD Rx/Tx Callbacks	254
49.7	Interrupt mode	255
49.7.1	Prerequisites for Rx interrupts	255
49.7.2	Example	255
49.8	Virtio PMD arguments	256
49.9	Virtio paths Selection and Usage	256
49.9.1	Virtio paths Selection	257
49.9.2	Rx/Tx callbacks of each Virtio path	257
49.9.3	Virtio paths Support Status from Release to Release	258
49.9.4	QEMU Support Status	258
49.9.5	How to Debug	259
50	Poll Mode Driver that wraps vhost library	260
50.1	Vhost Implementation in DPDK	260
50.2	Features and Limitations of vhost PMD	260
50.3	Vhost PMD arguments	260
50.4	Vhost PMD event handling	261
50.5	Vhost PMD with testpmd application	261
51	Poll Mode Driver for Paravirtual VMXNET3 NIC	262
51.1	VMXNET3 Implementation in the DPDK	262
51.2	Features and Limitations of VMXNET3 PMD	263
51.3	Prerequisites	263
51.4	VMXNET3 with a Native NIC Connected to a vSwitch	264
51.5	VMXNET3 Chaining VMs Connected to a vSwitch	265
52	Libpcap and Ring Based Poll Mode Drivers	267
52.1	Using the Drivers from the EAL Command Line	267
52.1.1	Libpcap-based PMD	267
52.1.2	Rings-based PMD	270
52.1.3	Using the Poll Mode Driver from an Application	271
53	Fail-safe poll mode driver library	273
53.1	Features	273
53.2	Compilation option	273
53.3	Using the Fail-safe PMD from the EAL command line	273

53.3.1	Fail-safe command line parameters	274
53.3.2	Usage example	274
53.4	Using the Fail-safe PMD from an application	275
53.5	Plug-in feature	275
53.6	Plug-out feature	275
53.7	Fail-safe glossary	276

OVERVIEW OF NETWORKING DRIVERS

The networking drivers may be classified in two categories:

- physical for real devices
- virtual for emulated devices

Some physical devices may be shaped through a virtual layer as for SR-IOV. The interface seen in the virtual environment is a VF (Virtual Function).

The ethdev layer exposes an API to use the networking functions of these devices. The bottom half part of ethdev is implemented by the drivers. Thus some features may not be implemented.

There are more differences between drivers regarding some internal properties, portability or even documentation availability. Most of these differences are summarized below.

More details about features can be found in [Features Overview](#).

Feature	a f _ x d p	a f p a c k e t	a r k	a t l a n t i c	a v p	a x g b e	b n x 2 x	b r
Speed capabilities			Y	Y		Y	P	P
Link status	Y			Y	Y	Y	Y	Y
Link status event				Y			Y	Y
Removal event								
Queue status event								
Rx interrupt								
Lock-free Tx queue								
Fast mbuf free								
Free Tx mbuf on demand								
Queue start/stop			Y	Y				
Runtime Rx queue setup								
Runtime Tx queue setup								
Burst mode info								
MTU update	Y			Y				
Jumbo frame			Y	Y	Y	Y		
Scattered Rx			Y		Y			
LRO								
TSO								
Promiscuous mode	Y			Y	Y	Y	Y	Y
Allmulticast mode				Y		Y		
Unicast MAC filter				Y	Y		Y	Y

Feature	af_xdp	afpacket	ark	atlantic	avp	axgbe	bnx2x	br
Multicast MAC filter							Y	Y
RSS hash				Y		Y		
RSS key update				Y				
RSS reta update				Y				
Inner RSS								
VMDq								
SR-IOV								Y
DCB								
VLAN filter				Y				
Flow control				Y				
Flow API								
Rate limitation								
Traffic mirroring								
Inline crypto								
CRC offload				Y		Y		
VLAN offload				Y	Y			
QinQ offload								
L3 checksum offload				Y		Y		
L4 checksum offload				Y		Y		
Timestamp offload								
MACsec offload				Y				
Inner L3 checksum								
Inner L4 checksum								
Packet type parsing				Y				
Timesync								
Rx descriptor status				Y				
Tx descriptor status				Y				
Basic stats			Y	Y	Y	Y	Y	Y
Extended stats				Y			Y	Y
Stats per queue	Y		Y	Y	Y			
FW version				Y				
EEPROM dump				Y				
Module EEPROM dump								
Registers dump				Y				
LED								
Multiprocess aware								
BSD nic_uio								
Linux UIO			Y	Y	Y	Y	Y	Y
Linux VFIO								
Other kdrv								
ARMv7								
ARMv8				Y				
Power8								
x86-32				Y		Y		
x86-64	Y		Y	Y	Y	Y	Y	Y
Usage doc			Y				Y	Y
Design doc								

Feature	a f _ x d p	a f p a c k e t	a r k	a t l a n t i c	a v p	a x g b e	b n x 2 x	b r
Perf doc								

Note: Features marked with “P” are partially supported. Refer to the appropriate NIC guide in the following sections for details.

FEATURES OVERVIEW

This section explains the supported features that are listed in the *Overview of Networking Drivers*.

As a guide to implementers it also shows the structs where the features are defined and the APIs that can be used to get/set the values.

Following tags used for feature details, these are from driver point of view:

[uses] : Driver uses some kind of input from the application.

[implements] : Driver implements a functionality.

[provides] : Driver provides some kind of data to the application. It is possible to provide data by implementing some function, but “provides” is used for cases where provided data can’t be represented simply by a function.

[related] : Related API with that feature.

2.1 Speed capabilities

Supports getting the speed capabilities that the current device is capable of.

- [provides] **rte_eth_dev_info**: `speed_capa:ETH_LINK_SPEED_*`.
- [related] **API**: `rte_eth_dev_info_get()`.

2.2 Link status

Supports getting the link speed, duplex mode and link state (up/down).

- [implements] **eth_dev_ops**: `link_update`.
- [implements] **rte_eth_dev_data**: `dev_link`.
- [related] **API**: `rte_eth_link_get()`, `rte_eth_link_get_nowait()`.

2.3 Link status event

Supports Link Status Change interrupts.

- [uses] **user config**: `dev_conf.intr_conf.lsc`.
- [uses] **rte_eth_dev_data**: `dev_flags:RTE_ETH_DEV_INTR_LSC`.

- **[uses]** `rte_eth_event_type`: `RTE_ETH_EVENT_INTR_LSC`.
- **[implements]** `rte_eth_dev_data`: `dev_link`.
- **[provides]** `rte_pci_driver.drv_flags`: `RTE_PCI_DRV_INTR_LSC`.
- **[related]** **API**: `rte_eth_link_get()`, `rte_eth_link_get_nowait()`.

2.4 Removal event

Supports device removal interrupts.

- **[uses]** **user config**: `dev_conf.intr_conf.rmv`.
- **[uses]** `rte_eth_dev_data`: `dev_flags: RTE_ETH_DEV_INTR_RMV`.
- **[uses]** `rte_eth_event_type`: `RTE_ETH_EVENT_INTR_RMV`.
- **[provides]** `rte_pci_driver.drv_flags`: `RTE_PCI_DRV_INTR_RMV`.

2.5 Queue status event

Supports queue enable/disable events.

- **[uses]** `rte_eth_event_type`: `RTE_ETH_EVENT_QUEUE_STATE`.

2.6 Rx interrupt

Supports Rx interrupts.

- **[uses]** **user config**: `dev_conf.intr_conf.rxq`.
- **[implements]** `eth_dev_ops`: `rx_queue_intr_enable`, `rx_queue_intr_disable`.
- **[related]** **API**: `rte_eth_dev_rx_intr_enable()`, `rte_eth_dev_rx_intr_disable()`.

2.7 Lock-free Tx queue

If a PMD advertises `DEV_TX_OFFLOAD_MT_LOCKFREE` capable, multiple threads can invoke `rte_eth_tx_burst()` concurrently on the same Tx queue without SW lock.

- **[uses]** `rte_eth_txconf`, `rte_eth_txmode`: `offloads: DEV_TX_OFFLOAD_MT_LOCKFREE`.
- **[provides]** `rte_eth_dev_info`: `tx_offload_capa`, `tx_queue_offload_capa: DEV_TX_OFFLOAD_MT_LOCKFREE`.
- **[related]** **API**: `rte_eth_tx_burst()`.

2.8 Fast mbuf free

Supports optimization for fast release of mbufs following successful Tx. Requires that per queue, all mbufs come from the same mempool and has `refcnt = 1`.

- **[uses]** `rte_eth_txconf, rte_eth_txmode`: `offloads:DEV_TX_OFFLOAD_MBUF_FAST_FREE`.
- **[provides]** `rte_eth_dev_info`: `tx_offload_capa, tx_queue_offload_capa:DEV_TX_OFFLOAD_MBU`

2.9 Free Tx mbuf on demand

Supports freeing consumed buffers on a Tx ring.

- **[implements]** `eth_dev_ops`: `tx_done_cleanup`.
- **[related]** **API**: `rte_eth_tx_done_cleanup()`.

2.10 Queue start/stop

Supports starting/stopping a specific Rx/Tx queue of a port.

- **[implements]** `eth_dev_ops`: `rx_queue_start, rx_queue_stop, tx_queue_start, tx_queue_stop`.
- **[related]** **API**: `rte_eth_dev_rx_queue_start(), rte_eth_dev_rx_queue_stop(), rte_eth_dev_tx_queue_start(), rte_eth_dev_tx_queue_stop()`.

2.11 MTU update

Supports updating port MTU.

- **[implements]** `eth_dev_ops`: `mtu_set`.
- **[implements]** `rte_eth_dev_data`: `mtu`.
- **[provides]** `rte_eth_dev_info`: `max_rx_pktlen`.
- **[related]** **API**: `rte_eth_dev_set_mtu(), rte_eth_dev_get_mtu()`.

2.12 Jumbo frame

Supports Rx jumbo frames.

- **[uses]** `rte_eth_rxconf, rte_eth_rxmode`: `offloads:DEV_RX_OFFLOAD_JUMBO_FRAME, dev_conf.rxmode.max_rx_pkt_len`.
- **[related]** `rte_eth_dev_info`: `max_rx_pktlen`.
- **[related]** **API**: `rte_eth_dev_set_mtu()`.

2.13 Scattered Rx

Supports receiving segmented mbufs.

- **[uses]** `rte_eth_rxconf, rte_eth_rxmode`: `offloads:DEV_RX_OFFLOAD_SCATTER`.
- **[implements]** `datapath`: Scattered Rx function.

- **[implements]** `rte_eth_dev_data`: `scattered_rx`.
- **[provides]** `eth_dev_ops`: `rxq_info_get:scattered_rx`.
- **[related]** `eth_dev_ops`: `rx_pkt_burst`.

2.14 LRO

Supports Large Receive Offload.

- **[uses]** `rte_eth_rxconf,rte_eth_rxmode`: `offloads:DEV_RX_OFFLOAD_TCP_LRO`.
`dev_conf.rxmode.max_lro_pkt_size`.
- **[implements]** `datapath`: LRO functionality.
- **[implements]** `rte_eth_dev_data`: `lro`.
- **[provides]** `mbuf`: `mbuf.ol_flags:PKT_RX_LRO`, `mbuf.tso_segsz`.
- **[provides]** `rte_eth_dev_info`: `rx_offload_capa`, `rx_queue_offload_capa:DEV_RX_OFFLOAD_TCP_LRO`.
- **[provides]** `rte_eth_dev_info`: `max_lro_pkt_size`.

2.15 TSO

Supports TCP Segmentation Offloading.

- **[uses]** `rte_eth_txconf,rte_eth_txmode`: `offloads:DEV_TX_OFFLOAD_TCP_TSO`.
- **[uses]** `rte_eth_desc_lim`: `nb_seg_max,nb_mtu_seg_max`.
- **[uses]** `mbuf`: `mbuf.ol_flags: PKT_TX_TCP_SEG, PKT_TX_IPV4, PKT_TX_IPV6, PKT_TX_IP_CKSUM`.
- **[uses]** `mbuf`: `mbuf.tso_segsz,mbuf.l2_len,mbuf.l3_len,mbuf.l4_len`.
- **[implements]** `datapath`: TSO functionality.
- **[provides]** `rte_eth_dev_info`: `tx_offload_capa,tx_queue_offload_capa:DEV_TX_OFFLOAD_TCP_TSO`.

2.16 Promiscuous mode

Supports enabling/disabling promiscuous mode for a port.

- **[implements]** `eth_dev_ops`: `promiscuous_enable,promiscuous_disable`.
- **[implements]** `rte_eth_dev_data`: `promiscuous`.
- **[related]** **API**: `rte_eth_promiscuous_enable()`, `rte_eth_promiscuous_disable()`, `rte_eth_promiscuous_get()`.

2.17 Allmulticast mode

Supports enabling/disabling receiving multicast frames.

- **[implements] eth_dev_ops:** `allmulticast_enable`, `allmulticast_disable`.
- **[implements] rte_eth_dev_data:** `all_multicast`.
- **[related] API:** `rte_eth_allmulticast_enable()`, `rte_eth_allmulticast_disable()`, `rte_eth_allmulticast_get()`.

2.18 Unicast MAC filter

Supports adding MAC addresses to enable whitelist filtering to accept packets.

- **[implements] eth_dev_ops:** `mac_addr_set`, `mac_addr_add`, `mac_addr_remove`.
- **[implements] rte_eth_dev_data:** `mac_addrs`.
- **[related] API:** `rte_eth_dev_default_mac_addr_set()`, `rte_eth_dev_mac_addr_add()`, `rte_eth_dev_mac_addr_remove()`, `rte_eth_macaddr_get()`.

2.19 Multicast MAC filter

Supports setting multicast addresses to filter.

- **[implements] eth_dev_ops:** `set_mc_addr_list`.
- **[related] API:** `rte_eth_dev_set_mc_addr_list()`.

2.20 RSS hash

Supports RSS hashing on RX.

- **[uses] user config:** `dev_conf.rxmode.mq_mode = ETH_MQ_RX_RSS_FLAG`.
- **[uses] user config:** `dev_conf.rx_adv_conf.rss_conf`.
- **[uses] rte_eth_rxconf, rte_eth_rxmode:** `offloads:DEV_RX_OFFLOAD_RSS_HASH`.
- **[provides] rte_eth_dev_info:** `flow_type_rss_offloads`.
- **[provides] mbuf:** `mbuf.ol_flags:PKT_RX_RSS_HASH`, `mbuf.rss`.

2.21 Inner RSS

Supports RX RSS hashing on Inner headers.

- **[uses] rte_flow_action_rss:** `level`.
- **[uses] rte_eth_rxconf, rte_eth_rxmode:** `offloads:DEV_RX_OFFLOAD_RSS_HASH`.
- **[provides] mbuf:** `mbuf.ol_flags:PKT_RX_RSS_HASH`, `mbuf.rss`.

2.22 RSS key update

Supports configuration of Receive Side Scaling (RSS) hash computation. Updating Receive Side Scaling (RSS) hash key.

- **[implements] eth_dev_ops:** `rss_hash_update`, `rss_hash_conf_get`.
- **[provides] rte_eth_dev_info:** `hash_key_size`.
- **[related] API:** `rte_eth_dev_rss_hash_update()`, `rte_eth_dev_rss_hash_conf_get()`.

2.23 RSS reta update

Supports updating Redirection Table of the Receive Side Scaling (RSS).

- **[implements] eth_dev_ops:** `reta_update`, `reta_query`.
- **[provides] rte_eth_dev_info:** `reta_size`.
- **[related] API:** `rte_eth_dev_rss_reta_update()`, `rte_eth_dev_rss_reta_query()`.

2.24 VMDq

Supports Virtual Machine Device Queues (VMDq).

- **[uses] user config:** `dev_conf.rxmode.mq_mode = ETH_MQ_RX_VMDQ_FLAG`.
- **[uses] user config:** `dev_conf.rx_adv_conf.vmdq_dcb_conf`.
- **[uses] user config:** `dev_conf.rx_adv_conf.vmdq_rx_conf`.
- **[uses] user config:** `dev_conf.tx_adv_conf.vmdq_dcb_tx_conf`.
- **[uses] user config:** `dev_conf.tx_adv_conf.vmdq_tx_conf`.

2.25 SR-IOV

Driver supports creating Virtual Functions.

- **[implements] rte_eth_dev_data:** `sriov`.

2.26 DCB

Supports Data Center Bridging (DCB).

- **[uses] user config:** `dev_conf.rxmode.mq_mode = ETH_MQ_RX_DCB_FLAG`.
- **[uses] user config:** `dev_conf.rx_adv_conf.vmdq_dcb_conf`.
- **[uses] user config:** `dev_conf.rx_adv_conf.dcb_rx_conf`.
- **[uses] user config:** `dev_conf.tx_adv_conf.vmdq_dcb_tx_conf`.
- **[uses] user config:** `dev_conf.tx_adv_conf.vmdq_tx_conf`.

- **[implements]** `eth_dev_ops`: `get_dcb_info`.
- **[related]** **API**: `rte_eth_dev_get_dcb_info()`.

2.27 VLAN filter

Supports filtering of a VLAN Tag identifier.

- **[uses]** `rte_eth_rxconf`, `rte_eth_rxmode`: `offloads:DEV_RX_OFFLOAD_VLAN_FILTER`.
- **[implements]** `eth_dev_ops`: `vlan_filter_set`.
- **[related]** **API**: `rte_eth_dev_vlan_filter()`.

2.28 Flow control

Supports configuring link flow control.

- **[implements]** `eth_dev_ops`: `flow_ctrl_get`, `flow_ctrl_set`, `priority_flow_ctrl_set`.
- **[related]** **API**: `rte_eth_dev_flow_ctrl_get()`, `rte_eth_dev_flow_ctrl_set()`, `rte_eth_dev_priority_flow_ctrl_set()`.

2.29 Flow API

Supports the DPDK Flow API for generic filtering.

- **[implements]** `eth_dev_ops`: `filter_ctrl:RTE_ETH_FILTER_GENERIC`.
- **[implements]** `rte_flow_ops`: `All`.

2.30 Rate limitation

Supports Tx rate limitation for a queue.

- **[implements]** `eth_dev_ops`: `set_queue_rate_limit`.
- **[related]** **API**: `rte_eth_set_queue_rate_limit()`.

2.31 Traffic mirroring

Supports adding traffic mirroring rules.

- **[implements]** `eth_dev_ops`: `mirror_rule_set`, `mirror_rule_reset`.
- **[related]** **API**: `rte_eth_mirror_rule_set()`, `rte_eth_mirror_rule_reset()`.

2.32 Inline crypto

Supports inline crypto processing (e.g. inline IPsec). See Security library and PMD documentation for more details.

- **[uses] `rte_eth_rxconf, rte_eth_rxmode`:** `offloads:DEV_RX_OFFLOAD_SECURITY`,
- **[uses] `rte_eth_txconf, rte_eth_txmode`:** `offloads:DEV_TX_OFFLOAD_SECURITY`.
- **[implements] `rte_security_ops`:** `session_create`, `session_update`,
`session_stats_get`, `session_destroy`, `set_pkt_metadata`,
`capabilities_get`.
- **[provides] `rte_eth_dev_info`:** `rx_offload_capa, rx_queue_offload_capa:DEV_RX_OFFLOAD_SECURITY`,
`tx_offload_capa, tx_queue_offload_capa:DEV_TX_OFFLOAD_SECURITY`.
- **[provides] `mbuf`:** `mbuf.ol_flags:PKT_RX_SEC_OFFLOAD`,
`mbuf.ol_flags:PKT_TX_SEC_OFFLOAD, mbuf.ol_flags:PKT_RX_SEC_OFFLOAD_FAILED`.

2.33 CRC offload

Supports CRC stripping by hardware. A PMD assumed to support CRC stripping by default. PMD should advertise if it supports keeping CRC.

- **[uses] `rte_eth_rxconf, rte_eth_rxmode`:** `offloads:DEV_RX_OFFLOAD_KEEP_CRC`.

2.34 VLAN offload

Supports VLAN offload to hardware.

- **[uses] `rte_eth_rxconf, rte_eth_rxmode`:** `offloads:DEV_RX_OFFLOAD_VLAN_STRIP, DEV_RX_OFFLOAD_VLAN_INSERT`.
- **[uses] `rte_eth_txconf, rte_eth_txmode`:** `offloads:DEV_TX_OFFLOAD_VLAN_INSERT`.
- **[uses] `mbuf`:** `mbuf.ol_flags:PKT_TX_VLAN, mbuf.vlan_tci`.
- **[implements] `eth_dev_ops`:** `vlan_offload_set`.
- **[provides] `mbuf`:** `mbuf.ol_flags:PKT_RX_VLAN_STRIPPED`,
`mbuf.ol_flags:PKT_RX_VLAN, mbuf.vlan_tci`.
- **[provides] `rte_eth_dev_info`:** `rx_offload_capa, rx_queue_offload_capa:DEV_RX_OFFLOAD_VLAN_INSERT`,
`tx_offload_capa, tx_queue_offload_capa:DEV_TX_OFFLOAD_VLAN_INSERT`.
- **[related] API:** `rte_eth_dev_set_vlan_offload()`, `rte_eth_dev_get_vlan_offload()`.

2.35 QinQ offload

Supports QinQ (queue in queue) offload.

- **[uses] `rte_eth_rxconf, rte_eth_rxmode`:** `offloads:DEV_RX_OFFLOAD_QINQ_STRIP`.
- **[uses] `rte_eth_txconf, rte_eth_txmode`:** `offloads:DEV_TX_OFFLOAD_QINQ_INSERT`.
- **[uses] `mbuf`:** `mbuf.ol_flags:PKT_TX_QINQ, mbuf.vlan_tci_outer`.

- **[provides] mbuf:** mbuf.ol_flags:PKT_RX_QINQ_STRIPPED,
mbuf.ol_flags:PKT_RX_QINQ, mbuf.ol_flags:PKT_RX_VLAN_STRIPPED,
mbuf.ol_flags:PKT_RX_VLAN mbuf.vlan_tci,mbuf.vlan_tci_outer.
- **[provides] rte_eth_dev_info:** rx_offload_capa,rx_queue_offload_capa:DEV_RX_OFFLOAD_QINQ_INSERT,
tx_offload_capa,tx_queue_offload_capa:DEV_TX_OFFLOAD_QINQ_INSERT.

2.36 L3 checksum offload

Supports L3 checksum offload.

- **[uses] rte_eth_rxconf,rte_eth_rxmode:** offloads:DEV_RX_OFFLOAD_IPV4_CKSUM.
- **[uses] rte_eth_txconf,rte_eth_txmode:** offloads:DEV_TX_OFFLOAD_IPV4_CKSUM.
- **[uses] mbuf:** mbuf.ol_flags:PKT_TX_IP_CKSUM,mbuf.ol_flags:PKT_TX_IPV4 |
PKT_TX_IPV6.
- **[uses] mbuf:** mbuf.l2_len,mbuf.l3_len.
- **[provides] mbuf:** mbuf.ol_flags:PKT_RX_IP_CKSUM_UNKNOWN |
PKT_RX_IP_CKSUM_BAD | PKT_RX_IP_CKSUM_GOOD | PKT_RX_IP_CKSUM_NONE.
- **[provides] rte_eth_dev_info:** rx_offload_capa,rx_queue_offload_capa:DEV_RX_OFFLOAD_IPV4_CKSUM,
tx_offload_capa,tx_queue_offload_capa:DEV_TX_OFFLOAD_IPV4_CKSUM.

2.37 L4 checksum offload

Supports L4 checksum offload.

- **[uses] rte_eth_rxconf,rte_eth_rxmode:** offloads:DEV_RX_OFFLOAD_UDP_CKSUM,DEV_RX_OFFLOAD_TCP_CKSUM.
- **[uses] rte_eth_txconf,rte_eth_txmode:** offloads:DEV_TX_OFFLOAD_UDP_CKSUM,DEV_TX_OFFLOAD_TCP_CKSUM.
- **[uses] mbuf:** mbuf.ol_flags:PKT_TX_IPV4 | PKT_TX_IPV6,
mbuf.ol_flags:PKT_TX_L4_NO_CKSUM | PKT_TX_TCP_CKSUM |
PKT_TX_SCTP_CKSUM | PKT_TX_UDP_CKSUM.
- **[uses] mbuf:** mbuf.l2_len,mbuf.l3_len.
- **[provides] mbuf:** mbuf.ol_flags:PKT_RX_L4_CKSUM_UNKNOWN |
PKT_RX_L4_CKSUM_BAD | PKT_RX_L4_CKSUM_GOOD | PKT_RX_L4_CKSUM_NONE.
- **[provides] rte_eth_dev_info:** rx_offload_capa,rx_queue_offload_capa:DEV_RX_OFFLOAD_UDP_CKSUM,
tx_offload_capa,tx_queue_offload_capa:DEV_TX_OFFLOAD_UDP_CKSUM,DEV_TX_OFFLOAD_TCP_CKSUM.

2.38 Timestamp offload

Supports Timestamp.

- **[uses] rte_eth_rxconf,rte_eth_rxmode:** offloads:DEV_RX_OFFLOAD_TIMESTAMP.
- **[provides] mbuf:** mbuf.ol_flags:PKT_RX_TIMESTAMP.
- **[provides] mbuf:** mbuf.timestamp.

- **[provides]** `rte_eth_dev_info`: `rx_offload_capa`, `rx_queue_offload_capa`: `DEV_RX_OFFLOAD_TIMESTAMP`.
- **[related]** `eth_dev_ops`: `read_clock`.

2.39 MACsec offload

Supports MACsec.

- **[uses]** `rte_eth_rxconf`, `rte_eth_rxmode`: `offloads`: `DEV_RX_OFFLOAD_MACSEC_STRIP`.
- **[uses]** `rte_eth_txconf`, `rte_eth_txmode`: `offloads`: `DEV_TX_OFFLOAD_MACSEC_INSERT`.
- **[uses]** `mbuf`: `mbuf.ol_flags`: `PKT_TX_MACSEC`.
- **[provides]** `rte_eth_dev_info`: `rx_offload_capa`, `rx_queue_offload_capa`: `DEV_RX_OFFLOAD_MACSEC_STRIP`, `tx_offload_capa`, `tx_queue_offload_capa`: `DEV_TX_OFFLOAD_MACSEC_INSERT`.

2.40 Inner L3 checksum

Supports inner packet L3 checksum.

- **[uses]** `rte_eth_rxconf`, `rte_eth_rxmode`: `offloads`: `DEV_RX_OFFLOAD_OUTER_IPV4_CKSUM`.
- **[uses]** `rte_eth_txconf`, `rte_eth_txmode`: `offloads`: `DEV_TX_OFFLOAD_OUTER_IPV4_CKSUM`.
- **[uses]** `mbuf`: `mbuf.ol_flags`: `PKT_TX_IP_CKSUM`, `mbuf.ol_flags`: `PKT_TX_IPV4` | `PKT_TX_IPV6`, `mbuf.ol_flags`: `PKT_TX_OUTER_IP_CKSUM`, `mbuf.ol_flags`: `PKT_TX_OUTER_IPV4` | `PKT_TX_OUTER_IPV6`.
- **[uses]** `mbuf`: `mbuf.outer_l2_len`, `mbuf.outer_l3_len`.
- **[provides]** `mbuf`: `mbuf.ol_flags`: `PKT_RX_EIP_CKSUM_BAD`.
- **[provides]** `rte_eth_dev_info`: `rx_offload_capa`, `rx_queue_offload_capa`: `DEV_RX_OFFLOAD_OUTER_IPV4_CKSUM`, `tx_offload_capa`, `tx_queue_offload_capa`: `DEV_TX_OFFLOAD_OUTER_IPV4_CKSUM`.

2.41 Inner L4 checksum

Supports inner packet L4 checksum.

- **[uses]** `rte_eth_rxconf`, `rte_eth_rxmode`: `offloads`: `DEV_RX_OFFLOAD_OUTER_UDP_CKSUM`.
- **[provides]** `mbuf`: `mbuf.ol_flags`: `PKT_RX_OUTER_L4_CKSUM_UNKNOWN` | `PKT_RX_OUTER_L4_CKSUM_BAD` | `PKT_RX_OUTER_L4_CKSUM_GOOD` | `PKT_RX_OUTER_L4_CKSUM_INVALID`.
- **[uses]** `rte_eth_txconf`, `rte_eth_txmode`: `offloads`: `DEV_TX_OFFLOAD_OUTER_UDP_CKSUM`.
- **[uses]** `mbuf`: `mbuf.ol_flags`: `PKT_TX_OUTER_IPV4` | `PKT_TX_OUTER_IPV6`, `mbuf.ol_flags`: `PKT_TX_OUTER_UDP_CKSUM`.
- **[uses]** `mbuf`: `mbuf.outer_l2_len`, `mbuf.outer_l3_len`.
- **[provides]** `rte_eth_dev_info`: `rx_offload_capa`, `rx_queue_offload_capa`: `DEV_RX_OFFLOAD_OUTER_UDP_CKSUM`, `tx_offload_capa`, `tx_queue_offload_capa`: `DEV_TX_OFFLOAD_OUTER_UDP_CKSUM`.

2.42 Packet type parsing

Supports packet type parsing and returns a list of supported types. Allows application to set ptypes it is interested in.

- **[implements] eth_dev_ops:** `dev_supported_ptypes_get`,
- **[related] API:** `rte_eth_dev_get_supported_ptypes()`,
`rte_eth_dev_set_ptypes()`, `dev_ptypes_set`.
- **[provides] mbuf:** `mbuf.packet_type`.

2.43 Timesync

Supports IEEE1588/802.1AS timestamping.

- **[implements] eth_dev_ops:** `timesync_enable`, `timesync_disable`,
`timesync_read_rx_timestamp`, `timesync_read_tx_timestamp`,
`timesync_adjust_time`, `timesync_read_time`, `timesync_write_time`.
- **[related] API:** `rte_eth_timesync_enable()`, `rte_eth_timesync_disable()`,
`rte_eth_timesync_read_rx_timestamp()`, `rte_eth_timesync_read_tx_timestamp`,
`rte_eth_timesync_adjust_time()`, `rte_eth_timesync_read_time()`,
`rte_eth_timesync_write_time()`.

2.44 Rx descriptor status

Supports check the status of a Rx descriptor. When `rx_descriptor_status` is used, status can be “Available”, “Done” or “Unavailable”. When `rx_descriptor_done` is used, status can be “DD bit is set” or “DD bit is not set”.

- **[implements] eth_dev_ops:** `rx_descriptor_status`.
- **[related] API:** `rte_eth_rx_descriptor_status()`.
- **[implements] eth_dev_ops:** `rx_descriptor_done`.
- **[related] API:** `rte_eth_rx_descriptor_done()`.

2.45 Tx descriptor status

Supports checking the status of a Tx descriptor. Status can be “Full”, “Done” or “Unavailable.”

- **[implements] eth_dev_ops:** `tx_descriptor_status`.
- **[related] API:** `rte_eth_tx_descriptor_status()`.

2.46 Basic stats

Support basic statistics such as: `ipackets`, `opackets`, `ibytes`, `obytes`, `imissed`, `ierrors`, `oerrors`, `rx_nombuf`.
And per queue stats: `q_ipackets`, `q_opackets`, `q_ibytes`, `q_obytes`, `q_errors`.

These apply to all drivers.

- **[implements] eth_dev_ops:** stats_get, stats_reset.
- **[related] API:** rte_eth_stats_get, rte_eth_stats_reset().

2.47 Extended stats

Supports Extended Statistics, changes from driver to driver.

- **[implements] eth_dev_ops:** xstats_get, xstats_reset, xstats_get_names.
- **[implements] eth_dev_ops:** xstats_get_by_id, xstats_get_names_by_id.
- **[related] API:** rte_eth_xstats_get(), rte_eth_xstats_reset(),
 rte_eth_xstats_get_names, rte_eth_xstats_get_by_id(),
 rte_eth_xstats_get_names_by_id(), rte_eth_xstats_get_id_by_name().

2.48 Stats per queue

Supports configuring per-queue stat counter mapping.

- **[implements] eth_dev_ops:** queue_stats_mapping_set.
- **[related] API:** rte_eth_dev_set_rx_queue_stats_mapping(),
 rte_eth_dev_set_tx_queue_stats_mapping().

2.49 FW version

Supports getting device hardware firmware information.

- **[implements] eth_dev_ops:** fw_version_get.
- **[related] API:** rte_eth_dev_fw_version_get().

2.50 EEPROM dump

Supports getting/setting device eeprom data.

- **[implements] eth_dev_ops:** get_eeprom_length, get_eeprom, set_eeprom.
- **[related] API:** rte_eth_dev_get_eeprom_length(),
 rte_eth_dev_get_eeprom(), rte_eth_dev_set_eeprom().

2.51 Module EEPROM dump

Supports getting information and data of plugin module eeprom.

- **[implements] eth_dev_ops:** get_module_info, get_module_eeprom.
- **[related] API:** rte_eth_dev_get_module_info(), rte_eth_dev_get_module_eeprom().

2.52 Registers dump

Supports retrieving device registers and registering attributes (number of registers and register size).

- **[implements] eth_dev_ops:** `get_reg`.
- **[related] API:** `rte_eth_dev_get_reg_info()`.

2.53 LED

Supports turning on/off a software controllable LED on a device.

- **[implements] eth_dev_ops:** `dev_led_on`, `dev_led_off`.
- **[related] API:** `rte_eth_led_on()`, `rte_eth_led_off()`.

2.54 Multiprocess aware

Driver can be used for primary-secondary process model.

2.55 BSD nic_uio

BSD `nic_uio` module supported.

2.56 Linux UIO

Works with `igb_uio` kernel module.

- **[provides] RTE_PMD_REGISTER_KMOD_DEP:** `igb_uio`.

2.57 Linux VFIO

Works with `vfio-pci` kernel module.

- **[provides] RTE_PMD_REGISTER_KMOD_DEP:** `vfio-pci`.

2.58 Other kdrv

Kernel module other than above ones supported.

2.59 ARMv7

Support armv7 architecture.

Use `defconfig_arm-armv7a-***`.

2.60 ARMv8

Support armv8a (64bit) architecture.

Use `defconfig_arm64-armv8a-*`

2.61 Power8

Support PowerPC architecture.

Use `defconfig_ppc_64-power8-*`

2.62 x86-32

Support 32bits x86 architecture.

Use `defconfig_x86_x32-native-*` and `defconfig_i686-native-*`.

2.63 x86-64

Support 64bits x86 architecture.

Use `defconfig_x86_64-native-*`.

2.64 Usage doc

Documentation describes usage.

See `doc/guides/nics/*.rst`

2.65 Design doc

Documentation describes design.

See `doc/guides/nics/*.rst`.

2.66 Perf doc

Documentation describes performance values.

See `dpdk.org/doc/perf/*`.

2.67 Runtime Rx queue setup

Supports Rx queue setup after device started.

- **[provides] `rte_eth_dev_info`:** `dev_capa : RTE_ETH_DEV_CAPA_RUNTIME_RX_QUEUE_SETUP`.
- **[related] API:** `rte_eth_dev_info_get()`.

2.68 Runtime Tx queue setup

Supports Tx queue setup after device started.

- **[provides] `rte_eth_dev_info`:** `dev_capa : RTE_ETH_DEV_CAPA_RUNTIME_TX_QUEUE_SETUP`.
- **[related] API:** `rte_eth_dev_info_get()`.

2.69 Burst mode info

Supports to get Rx/Tx packet burst mode information.

- **[implements] `eth_dev_ops`:** `rx_burst_mode_get`, `tx_burst_mode_get`.
- **[related] API:** `rte_eth_rx_burst_mode_get()`, `rte_eth_tx_burst_mode_get()`.

2.70 Other dev ops not represented by a Feature

- `rxq_info_get`
- `txq_info_get`
- `vlan_tpid_set`
- `vlan_strip_queue_set`
- `vlan_pvid_set`
- `rx_queue_count`
- `l2_tunnel_offload_set`
- `uc_hash_table_set`
- `uc_all_hash_table_set`
- `udp_tunnel_port_add`
- `udp_tunnel_port_del`
- `l2_tunnel_eth_type_conf`
- `l2_tunnel_offload_set`
- `tx_pkt_prepare`

COMPILING AND TESTING A PMD FOR A NIC

This section demonstrates how to compile and run a Poll Mode Driver (PMD) for the available Network Interface Cards in DPDK using TestPMD.

TestPMD is one of the reference applications distributed with the DPDK. Its main purpose is to forward packets between Ethernet ports on a network interface and as such is the best way to test a PMD.

Refer to the testpmd application user guide for detailed information on how to build and run testpmd.

3.1 Driver Compilation

To compile a PMD for a platform, run make with appropriate target as shown below. Use “make” command in Linux and “gmake” in FreeBSD. This will also build testpmd.

To check available targets:

```
cd <DPDK-source-directory>
make showconfigs
```

Example output:

```
arm-armv7a-linux-gcc
arm64-armv8a-linux-gcc
arm64-dpaa-linux-gcc
arm64-thunderx-linux-gcc
arm64-xgene1-linux-gcc
i686-native-linux-gcc
i686-native-linux-icc
ppc_64-power8-linux-gcc
x86_64-native-freebsd-clang
x86_64-native-freebsd-gcc
x86_64-native-linux-clang
x86_64-native-linux-gcc
x86_64-native-linux-icc
x86_x32-native-linux-gcc
```

To compile a PMD for Linux x86_64 gcc target, run the following “make” command:

```
make install T=x86_64-native-linux-gcc
```

Use ARM (ThunderX, DPAA, X-Gene) or PowerPC target for respective platform.

For more information, refer to the Getting Started Guide for Linux or Getting Started Guide for FreeBSD depending on your platform.

3.2 Running testpmd in Linux

This section demonstrates how to setup and run testpmd in Linux.

1. Mount huge pages:

```
mkdir /mnt/huge
mount -t hugetlbfs nodev /mnt/huge
```

2. Request huge pages:

Hugepage memory should be reserved as per application requirement. Check hugepage size configured in the system and calculate the number of pages required.

To reserve 1024 pages of 2MB:

```
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

Note: Check /proc/meminfo to find system hugepage size:

```
grep "Hugepagesize:" /proc/meminfo
```

Example output:

```
Hugepagesize:      2048 kB
```

3. Load igb_uio or vfio-pci driver:

```
modprobe uio
insmod ./x86_64-native-linux-gcc/kmod/igb_uio.ko
```

or

```
modprobe vfio-pci
```

4. Setup VFIO permissions for regular users before binding to vfio-pci:

```
sudo chmod a+x /dev/vfio
sudo chmod 0666 /dev/vfio/*
```

5. Bind the adapters to igb_uio or vfio-pci loaded in the previous step:

```
./usertools/dpdk-devbind.py --bind igb_uio DEVICE1 DEVICE2 ...
```

Or setup VFIO permissions for regular users and then bind to vfio-pci:

```
./usertools/dpdk-devbind.py --bind vfio-pci DEVICE1 DEVICE2 ...
```

Note: DEVICE1, DEVICE2 are specified via PCI “domain:bus:slot.func” syntax or “bus:slot.func” syntax.

6. Start testpmd with basic parameters:

```
./x86_64-native-linux-gcc/app/testpmd -l 0-3 -n 4 -- -i
```

Successful execution will show initialization messages from EAL, PMD and testpmd application. A prompt will be displayed at the end for user commands as interactive mode (-i) is on.

```
testpmd>
```

Refer to the testpmd runtime functions for a list of available commands.

Note: When `testpmd` is built with shared library, use option `-d` to load the dynamic PMD for `rte_eal_init`.

AF_PACKET POLL MODE DRIVER

The AF_PACKET socket in Linux allows an application to receive and send raw packets. This Linux-specific PMD driver binds to an AF_PACKET socket and allows a DPDK application to send and receive raw packets through the Kernel.

In order to improve Rx and Tx performance this implementation makes use of PACKET_MMAP, which provides a mmap'ed ring buffer, shared between user space and kernel, that's used to send and receive packets. This helps reducing system calls and the copies needed between user space and Kernel.

The PACKET_FANOUT_HASH behavior of AF_PACKET is used for frame reception.

4.1 Options and inherent limitations

The following options can be provided to set up an af_packet port in DPDK. Some of these, in turn, will be used to configure the PACKET_MMAP settings.

- `iface` - name of the Kernel interface to attach to (required);
- `qpairs` - number of Rx and Tx queues (optional, default 1);
- `qdisc_bypass` - set PACKET_QDISC_BYPASS option in AF_PACKET (optional, disabled by default);
- `blocksz` - PACKET_MMAP block size (optional, default 4096);
- `framesz` - PACKET_MMAP frame size (optional, default 2048B; Note: multiple of 16B);
- `framecnt` - PACKET_MMAP frame count (optional, default 512).

Because this implementation is based on PACKET_MMAP, and PACKET_MMAP has its own pre-requisites, it should be noted that the inner workings of PACKET_MMAP should be carefully considered before modifying some of these options (namely, `blocksz`, `framesz` and `framecnt` above).

As an example, if one changes `framesz` to be 1024B, it is expected that `blocksz` is set to at least 1024B as well (although 2048B in this case would allow two “frames” per “block”).

This restriction happens because PACKET_MMAP expects each single “frame” to fit inside of a “block”. And although multiple “frames” can fit inside of a single “block”, a “frame” may not span across two “blocks”.

For the full details behind PACKET_MMAP's structures and settings, consider reading the [PACKET_MMAP documentation in the Kernel](#).

4.2 Prerequisites

This is a Linux-specific PMD, thus the following prerequisites apply:

- A Linux Kernel;
- A Kernel bound interface to attach to (e.g. a tap interface).

4.3 Set up an af_packet interface

The following example will set up an af_packet interface in DPDK with the default options described above (blocksz=4096B, framesz=2048B and framecnt=512):

```
--vdev=eth_af_packet0,iface=tap0,blocksz=4096,framesz=2048,framecnt=512,qpairs=1,qdisc_bypass=0
```

AF_XDP POLL MODE DRIVER

AF_XDP is an address family that is optimized for high performance packet processing. AF_XDP sockets enable the possibility for XDP program to redirect packets to a memory buffer in userspace.

For the full details behind AF_XDP socket, you can refer to [AF_XDP documentation in the Kernel](#).

This Linux-specific PMD driver creates the AF_XDP socket and binds it to a specific netdev queue, it allows a DPDK application to send and receive raw packets through the socket which would bypass the kernel network stack. Current implementation only supports single queue, multi-queues feature will be added later.

Note that MTU of AF_XDP PMD is limited due to XDP lacks support for fragmentation.

AF_XDP PMD enables `need_wakeup` flag by default if it is supported. This `need_wakeup` feature is used to support executing application and driver on the same core efficiently. This feature not only has a large positive performance impact for the one core case, but also does not degrade 2 core performance and actually improves it for Tx heavy workloads.

5.1 Options

The following options can be provided to set up an `af_xdp` port in DPDK.

- `iface` - name of the Kernel interface to attach to (required);
- `start_queue` - starting netdev queue id (optional, default 0);
- `queue_count` - total netdev queue number (optional, default 1);

5.2 Prerequisites

This is a Linux-specific PMD, thus the following prerequisites apply:

- A Linux Kernel (version > v4.18) with XDP sockets configuration enabled;
- libbpf (within kernel version > v5.1-rc4) with latest `af_xdp` support installed, User can install libbpf via `make install_lib && make install_headers` in <kernel src tree>/tools/lib/bpf;
- A Kernel bound interface to attach to;
- For `need_wakeup` feature, it requires kernel version later than v5.3-rc1;
- For PMD zero copy, it requires kernel version later than v5.4-rc1;

5.3 Set up an af_xdp interface

The following example will set up an af_xdp interface in DPDK:

```
--vdev net_af_xdp,iface=ens786f1
```


ARK POLL MODE DRIVER

The ARK PMD is a DPDK poll-mode driver for the Atomic Rules Arkville (ARK) family of devices. More information can be found at the [Atomic Rules website](#).

6.1 Overview

The Atomic Rules Arkville product is DPDK and AXI compliant product that marshals packets across a PCIe conduit between host DPDK mbufs and FPGA AXI streams.

The ARK PMD, and the spirit of the overall Arkville product, has been to take the DPDK API/ABI as a fixed specification; then implement much of the business logic in FPGA RTL circuits. The approach of *working backwards* from the DPDK API/ABI and having the GPP host software *dictate*, while the FPGA hardware *copies*, results in significant performance gains over a naive implementation.

While this document describes the ARK PMD software, it is helpful to understand what the FPGA hardware is and is not. The Arkville RTL component provides a single PCIe Physical Function (PF) supporting some number of RX/Ingress and TX/Egress Queues. The ARK PMD controls the Arkville core through a dedicated opaque Core BAR (CBAR). To allow users full freedom for their own FPGA application IP, an independent FPGA Application BAR (ABAR) is provided.

One popular way to imagine Arkville's FPGA hardware aspect is as the FPGA PCIe-facing side of a so-called Smart NIC. The Arkville core does not contain any MACs, and is link-speed independent, as well as agnostic to the number of physical ports the application chooses to use. The ARK driver exposes the familiar PMD interface to allow packet movement to and from mbufs across multiple queues.

However FPGA RTL applications could contain a universe of added functionality that an Arkville RTL core does not provide or can not anticipate. To allow for this expectation of user-defined innovation, the ARK PMD provides a dynamic mechanism of adding capabilities without having to modify the ARK PMD.

The ARK PMD is intended to support all instances of the Arkville RTL Core, regardless of configuration, FPGA vendor, or target board. While specific capabilities such as number of physical hardware queue-pairs are negotiated; the driver is designed to remain constant over a broad and extendable feature set.

Intentionally, Arkville by itself DOES NOT provide common NIC capabilities such as offload or receive-side scaling (RSS). These capabilities would be viewed as a gate-level "tax" on Green-box FPGA applications that do not require such function. Instead, they can be added as needed with essentially no overhead to the FPGA Application.

The ARK PMD also supports optional user extensions, through dynamic linking. The ARK PMD user extensions are a feature of Arkville's DPDK net/ark poll mode driver, allowing users to add their own code to extend the net/ark functionality without having to make source code changes to the driver. One

motivation for this capability is that while DPDK provides a rich set of functions to interact with NIC-like capabilities (e.g. MAC addresses and statistics), the Arkville RTL IP does not include a MAC. Users can supply their own MAC or custom FPGA applications, which may require control from the PMD. The user extension is the means providing the control between the user's FPGA application and the existing DPDK features via the PMD.

6.2 Device Parameters

The ARK PMD supports device parameters that are used for packet routing and for internal packet generation and packet checking. This section describes the supported parameters. These features are primarily used for diagnostics, testing, and performance verification under the guidance of an Arkville specialist. The nominal use of Arkville does not require any configuration using these parameters.

“Pkt_dir”

The Packet Director controls connectivity between Arkville's internal hardware components. The features of the Pkt_dir are only used for diagnostics and testing; it is not intended for nominal use. The full set of features are not published at this level.

Format: Pkt_dir=0x00110F10

“Pkt_gen”

The packet generator parameter takes a file as its argument. The file contains configuration parameters used internally for regression testing and are not intended to be published at this level. The packet generator is an internal Arkville hardware component.

Format: Pkt_gen=./config/pg.conf

“Pkt_chkr”

The packet checker parameter takes a file as its argument. The file contains configuration parameters used internally for regression testing and are not intended to be published at this level. The packet checker is an internal Arkville hardware component.

Format: Pkt_chkr=./config/pc.conf

6.3 Data Path Interface

Ingress RX and Egress TX operation is by the nominal DPDK API . The driver supports single-port, multi-queue for both RX and TX.

6.4 Configuration Information

DPDK Configuration Parameters

The following configuration options are available for the ARK PMD:

- **CONFIG_RTE_LIBRTE_ARK_PMD** (default y): Enables or disables inclusion of the ARK PMD driver in the DPDK compilation.
- **CONFIG_RTE_LIBRTE_ARK_PAD_TX** (default y): When enabled TX packets are padded to 60 bytes to support downstream MACS.

- **CONFIG_RTE_LIBRTE_ARK_DEBUG_RX** (default n): Enables or disables debug logging and internal checking of RX ingress logic within the ARK PMD driver.
- **CONFIG_RTE_LIBRTE_ARK_DEBUG_TX** (default n): Enables or disables debug logging and internal checking of TX egress logic within the ARK PMD driver.
- **CONFIG_RTE_LIBRTE_ARK_DEBUG_STATS** (default n): Enables or disables debug logging of detailed packet and performance statistics gathered in the PMD and FPGA.
- **CONFIG_RTE_LIBRTE_ARK_DEBUG_TRACE** (default n): Enables or disables debug logging of detailed PMD events and status.

6.5 Building DPDK

See the DPDK Getting Started Guide for Linux for instructions on how to build DPDK.

By default the ARK PMD library will be built into the DPDK library.

For configuring and using UIO and VFIO frameworks, please also refer the documentation that comes with DPDK suite.

6.6 Supported ARK RTL PCIe Instances

ARK PMD supports the following Arkville RTL PCIe instances including:

- 1d6c:100d - AR-ARKA-FX0 [Arkville 32B DPDK Data Mover]
- 1d6c:100e - AR-ARKA-FX1 [Arkville 64B DPDK Data Mover]

6.7 Supported Operating Systems

Any Linux distribution fulfilling the conditions described in `System Requirements` section of the DPDK documentation or refer to *DPDK Release Notes*. ARM and PowerPC architectures are not supported at this time.

6.8 Supported Features

- Dynamic ARK PMD extensions
- Multiple receive and transmit queues
- Jumbo frames up to 9K
- Hardware Statistics

6.9 Unsupported Features

Features that may be part of, or become part of, the Arkville RTL IP that are not currently supported or exposed by the ARK PMD include:

- PCIe SR-IOV Virtual Functions (VFs)
- Arkville's Packet Generator Control and Status
- Arkville's Packet Director Control and Status
- Arkville's Packet Checker Control and Status
- Arkville's Timebase Management

6.10 Pre-Requisites

1. Prepare the system as recommended by DPDK suite. This includes environment variables, hugepages configuration, tool-chains and configuration
2. Insert `igb_uio` kernel module using the command `'modprobe igb_uio'`
3. Bind the intended ARK device to `igb_uio` module

At this point the system should be ready to run DPDK applications. Once the application runs to completion, the ARK PMD can be detached from `igb_uio` if necessary.

6.11 Usage Example

Follow instructions available in the document [compiling and testing a PMD for a NIC](#) to launch `testpmd` with Atomic Rules ARK devices managed by `librte_pmd_ark`.

Example output:

```
[...]
EAL: PCI device 0000:01:00.0 on NUMA socket -1
EAL:  probe driver: 1d6c:100e rte_ark_pmd
EAL:  PCI memory mapped at 0x7f9b6c400000
PMD: eth_ark_dev_init(): Initializing 0:2:0.1
ARKP PMD CommitID: 378f3a67
Configuring Port 0 (socket 0)
Port 0: DC:3C:F6:00:00:01
Checking link statuses...
Port 0 Link Up - speed 100000 Mbps - full-duplex
Done
testpmd>
```

AQUANTIA ATLANTIC DPDK DRIVER

Atlantic DPDK driver provides DPDK support for Aquantia's AQtion family of chipsets: AQC107/AQC108/AQC109

More information can be found at [Aquantia Official Website](#).

7.1 Supported features

- Base L2 features
- Promiscuous mode
- Multicast mode
- Port statistics
- RSS (Receive Side Scaling)
- Checksum offload
- Jumbo Frame up to 16K
- MACSEC offload

7.2 Experimental API features

- MACSEC PMD API is considered as experimental and is subject to change/removal in next DPDK releases.

7.3 Configuration Information

- `CONFIG_RTE_LIBRTE_ATLANTIC_PMD` (default y)

7.3.1 Application Programming Interface

7.3.2 Limitations or Known issues

Statistics

MTU setting

Atlantic NIC supports up to 16K jumbo frame size

7.3.3 Supported Chipsets and NICs

- Aquantia AQtion AQC107 10 Gigabit Ethernet Controller
- Aquantia AQtion AQC108 5 Gigabit Ethernet Controller
- Aquantia AQtion AQC109 2.5 Gigabit Ethernet Controller

AVP POLL MODE DRIVER

The Accelerated Virtual Port (AVP) device is a shared memory based device only available on [virtualization platforms](#) from Wind River Systems. The Wind River Systems virtualization platform currently uses QEMU/KVM as its hypervisor and as such provides support for all of the QEMU supported virtual and/or emulated devices (e.g., virtio, e1000, etc.). The platform offers the virtio device type as the default device when launching a virtual machine or creating a virtual machine port. The AVP device is a specialized device available to customers that require increased throughput and decreased latency to meet the demands of their performance focused applications.

The AVP driver binds to any AVP PCI devices that have been exported by the Wind River Systems QEMU/KVM hypervisor. As a user of the DPDK driver API it supports a subset of the full Ethernet device API to enable the application to use the standard device configuration functions and packet receive/transmit functions.

These devices enable optimized packet throughput by bypassing QEMU and delivering packets directly to the virtual switch via a shared memory mechanism. This provides DPDK applications running in virtual machines with significantly improved throughput and latency over other device types.

The AVP device implementation is integrated with the QEMU/KVM live-migration mechanism to allow applications to seamlessly migrate from one hypervisor node to another with minimal packet loss.

8.1 Features and Limitations of the AVP PMD

The AVP PMD driver provides the following functionality.

- Receive and transmit of both simple and chained mbuf packets,
- Chained mbufs may include up to 5 chained segments,
- Up to 8 receive and transmit queues per device,
- Only a single MAC address is supported,
- The MAC address cannot be modified,
- The maximum receive packet length is 9238 bytes,
- VLAN header stripping and inserting,
- Promiscuous mode
- VM live-migration
- PCI hotplug insertion and removal

8.2 Prerequisites

The following prerequisites apply:

- A virtual machine running in a Wind River Systems virtualization environment and configured with at least one neutron port defined with a vif-model set to “avp”.

8.3 Launching a VM with an AVP type network attachment

The following example will launch a VM with three network attachments. The first attachment will have a default vif-model of “virtio”. The next two network attachments will have a vif-model of “avp” and may be used with a DPDK application which is built to include the AVP PMD driver.

```
nova boot --flavor small --image my-image \  
  --nic net-id=${NETWORK1_UUID} \  
  --nic net-id=${NETWORK2_UUID},vif-model=avp \  
  --nic net-id=${NETWORK3_UUID},vif-model=avp \  
  --security-group default my-instance1
```


AXGBE POLL MODE DRIVER

The AXGBE poll mode driver library (**librte_pmd_axgbe**) implements support for AMD 10 Gbps family of adapters. It is compiled and tested in standard linux distro like Ubuntu.

Detailed information about SoCs that use these devices can be found here:

- [AMD EPYC™ EMBEDDED 3000 family](#).

9.1 Supported Features

AXGBE PMD has support for:

- Base L2 features
- TSS (Transmit Side Scaling)
- Promiscuous mode
- Port statistics
- Multicast mode
- RSS (Receive Side Scaling)
- Checksum offload
- Jumbo Frame up to 9K

9.2 Configuration Information

The following options can be modified in the `.config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_AXGBE_PMD` (default **y**)
Toggle compilation of axgbe PMD.
- `CONFIG_RTE_LIBRTE_AXGBE_PMD_DEBUG` (default **n**)
Toggle display for PMD debug related messages.

9.3 Building DPDK

See the DPDK Getting Started Guide for Linux for instructions on how to build DPDK.

By default the AXGBE PMD library will be built into the DPDK library.

For configuring and using UIO frameworks, please also refer the documentation that comes with DPDK suite.

9.4 Prerequisites and Pre-conditions

- Prepare the system as recommended by DPDK suite.
- Bind the intended AMD device to `igb_uio` or `vfio-pci` module.

Now system is ready to run DPDK application.

9.5 Usage Example

Refer to the document *compiling and testing a PMD for a NIC* for details.

Example output:

```
[...]
EAL: PCI device 0000:02:00.4 on NUMA socket 0
EAL:  probe driver: 1022:1458 net_axgbe
Interactive-mode selected
USER1: create a new mbuf pool <mbuf_pool_socket_0>: n=171456, size=2176, socket=0
USER1: create a new mbuf pool <mbuf_pool_socket_1>: n=171456, size=2176, socket=1
USER1: create a new mbuf pool <mbuf_pool_socket_2>: n=171456, size=2176, socket=2
USER1: create a new mbuf pool <mbuf_pool_socket_3>: n=171456, size=2176, socket=3
Configuring Port 0 (socket 0)
Port 0: 00:00:1A:1C:6A:17
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

BNX2X POLL MODE DRIVER

The BNX2X poll mode driver library (**librte_pmd_bnx2x**) implements support for **QLogic 578xx** 10/20 Gbps family of adapters as well as their virtual functions (VF) in SR-IOV context. It is supported on several standard Linux distros like RHEL and SLES. It is compile-tested under FreeBSD OS.

More information can be found at [QLogic Corporation's Official Website](#).

10.1 Supported Features

BNX2X PMD has support for:

- Base L2 features
- Unicast/multicast filtering
- Promiscuous mode
- Port hardware statistics
- SR-IOV VF

10.2 Non-supported Features

The features not yet supported include:

- TSS (Transmit Side Scaling)
- RSS (Receive Side Scaling)
- LRO/TSO offload
- Checksum offload
- SR-IOV PF
- Rx TX scatter gather

10.3 Co-existence considerations

- QLogic 578xx CNAs support Ethernet, iSCSI and FCoE functionalities. These functionalities are supported using QLogic Linux kernel drivers `bnx2x`, `cnic`, `bnx2i` and `bnx2fc`. DPDK is supported on these adapters using `bnx2x` PMD.

- When SR-IOV is not enabled on the adapter, QLogic Linux kernel drivers (bnx2x, cnic, bnx2i and bnx2fc) and bnx2x PMD can't be attached to different PFs on a given QLogic 578xx adapter. A given adapter needs to be completely used by DPDK or Linux drivers. Before binding DPDK driver to one or more PFs on the adapter, please make sure to unbind Linux drivers from all PFs of the adapter. If there are multiple adapters on the system, one or more adapters can be used by DPDK driver completely and other adapters can be used by Linux drivers completely.
- When SR-IOV is enabled on the adapter, Linux kernel drivers (bnx2x, cnic, bnx2i and bnx2fc) can be bound to the PFs of a given adapter and either bnx2x PMD or Linux drivers bnx2x can be bound to the VFs of the adapter.

10.4 Supported QLogic NICs

- 578xx

10.5 Prerequisites

- Requires firmware version **7.13.11.0**. It is included in most of the standard Linux distros. If it is not available visit [linux-firmware git repository](#) to get the required firmware.

10.6 Pre-Installation Configuration

10.6.1 Config File Options

The following options can be modified in the `.config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_BNX2X_PMD` (default **n**)
Toggle compilation of bnx2x driver. To use bnx2x PMD set this config parameter to 'y'. Also, in order for firmware binary to load user will need zlib devel package installed.
- `CONFIG_RTE_LIBRTE_BNX2X_DEBUG_TX` (default **n**)
Toggle display of transmit fast path run-time messages.
- `CONFIG_RTE_LIBRTE_BNX2X_DEBUG_RX` (default **n**)
Toggle display of receive fast path run-time messages.
- `CONFIG_RTE_LIBRTE_BNX2X_DEBUG_PERIODIC` (default **n**)
Toggle display of register reads and writes.

10.7 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

10.8 SR-IOV: Prerequisites and sample Application Notes

This section provides instructions to configure SR-IOV with Linux OS.

1. Verify SR-IOV and ARI capabilities are enabled on the adapter using `lspci`:

```
lspci -s <slot> -vvv
```

Example output:

```
[...]
Capabilities: [1b8 v1] Alternative Routing-ID Interpretation (ARI)
[...]
Capabilities: [1c0 v1] Single Root I/O Virtualization (SR-IOV)
[...]
Kernel driver in use: igb_uio
```

2. Load the kernel module:

```
modprobe bnx2x
```

Example output:

```
systemd-udevd[4848]: renamed network interface eth0 to ens5f0
systemd-udevd[4848]: renamed network interface eth1 to ens5f1
```

3. Bring up the PF ports:

```
ifconfig ens5f0 up
ifconfig ens5f1 up
```

4. Create VF device(s):

Echo the number of VFs to be created into “sriov_numvfs” sysfs entry of the parent PF.

Example output:

```
echo 2 > /sys/devices/pci0000:00/0000:00:03.0/0000:81:00.0/sriov_numvfs
```

5. Assign VF MAC address:

Assign MAC address to the VF using `iproute2` utility. The syntax is: `ip link set <PF iface> vf <VF id> mac <macaddr>`

Example output:

```
ip link set ens5f0 vf 0 mac 52:54:00:2f:9d:e8
```

6. PCI Passthrough:

The VF devices may be passed through to the guest VM using `virt-manager` or `virsh` etc. `bnx2x` PMD should be used to bind the VF devices in the guest VM using the instructions outlined in the Application notes below.

7. Running `testpmd`: (Supply `--log-level="pmd.net.bnx2x.driver"`, 7 to view informational messages):

Follow instructions available in the document *compiling and testing a PMD for a NIC* to run `testpmd`.

Example output:

```
[...]
EAL: PCI device 0000:84:00.0 on NUMA socket 1
EAL: probe driver: 14e4:168e rte_bnx2x_pmd
EAL: PCI memory mapped at 0x7f14f6fe5000
```

```
EAL:   PCI memory mapped at 0x7f14f67e5000
EAL:   PCI memory mapped at 0x7f15fbd9b000
EAL: PCI device 0000:84:00.1 on NUMA socket 1
EAL:   probe driver: 14e4:168e rte_bnx2x_pmd
EAL:   PCI memory mapped at 0x7f14f5fe5000
EAL:   PCI memory mapped at 0x7f14f57e5000
EAL:   PCI memory mapped at 0x7f15fbd4f000
Interactive-mode selected
Configuring Port 0 (socket 0)
PMD: bnx2x_dev_tx_queue_setup(): fp[00] req_bd=512, thresh=512,
      usable_bd=1020, total_bd=1024,
      tx_pages=4
PMD: bnx2x_dev_rx_queue_setup(): fp[00] req_bd=128, thresh=0,
      usable_bd=510, total_bd=512,
      rx_pages=1, cq_pages=8
PMD: bnx2x_print_adapter_info():
[...]
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

BNXT POLL MODE DRIVER

The BNXT PMD (`librte_pmd_bnxt`) implements support for adapters based on Ethernet controllers and SoCs belonging to the **Broadcom BCM5730X NetXtreme-C® Family of Ethernet Network Controllers**, the **Broadcom BCM574XX/BCM575XX NetXtreme-E® Family of Ethernet Network Controllers**, the **Broadcom BCM588XX Stingray Family of SmartNIC Adapters**, and the **Broadcom StrataGX® BCM5871X Series of Communications Processors**. A complete list with links to reference material is included below.

11.1 BNXT PMD Features

The BNXT PMD includes support for the following features:

- Multiple transmit and receive queues
- Queue start/stop
- RSS hash
- RSS key configuration
- RSS reta configuration
- VMDq
- Packet type parsing
- Configurable RX CRC stripping
- L3/L4 checksum offload
- LRO offload
- TSO offload
- VLAN offload
- SR-IOV VF
- Basic and extended port statistics
- Link state reporting
- Flow control
- Ethertype filtering
- N-tuple filtering

- Promiscuous mode
- Unicast and multicast MAC filtering
- Scatter/gather transmit and receive
- Jumbo frames
- Vector PMD

11.2 BNXT Vector PMD

The BNXT PMD includes support for SSE vector mode on x86 platforms. Vector provides significantly improved performance over the base implementation, however it does not support all of the features that are supported by the base (non-vector) implementation. Vector mode will be selected and enabled automatically when the port is started if allowed by the current configuration.

11.2.1 RX Requirements for Vector Mode

Vector mode receive will be enabled if the following constraints are met:

- Packets must fit within a single mbuf (no scatter RX).
- LRO offload must be disabled.

11.2.2 TX Requirements for Vector Mode

Vector mode transmit will be enabled if the following constraints are met:

- Packets must be contained within a single mbuf (no gather TX).
- All transmit offloads other than VLAN insertion must be disabled.

11.3 BNXT PMD Supported Chipsets and Adapters

Chipsets and adapters supported by the bnxt PMD include:

- **Broadcom BCM5730X NetXtreme-C® Family of Ethernet Network Controllers**
 - M150c - Single-port 40/50 Gigabit Ethernet Adapter
 - P150c - Single-port 40/50 Gigabit Ethernet Adapter
 - P225c - Dual-port 10/25 Gigabit Ethernet Adapter
- **Broadcom BCM574XX/BCM575XX NetXtreme-E® Family of Ethernet Network Controllers**
 - M125P - Single-port OCP 2.0 10/25 Gigabit Ethernet Adapter
 - M150P - Single-port OCP 2.0 50 Gigabit Ethernet Adapter
 - M150PM - Single-port OCP 2.0 Multi-Host 50 Gigabit Ethernet Adapter
 - M210P - Dual-port OCP 2.0 10 Gigabit Ethernet Adapter

- M210TP - Dual-port OCP 2.0 10 Gigabit Ethernet Adapter
- M11000G - Single-port OCP 2.0 10/25/50/100 Gigabit Ethernet Adapter
- N150G - Single-port OCP 3.0 50 Gigabit Ethernet Adapter
- M225P - Dual-port OCP 2.0 10/25 Gigabit Ethernet Adapter
- N210P - Dual-port OCP 3.0 10 Gigabit Ethernet Adapter
- N210TP - Dual-port OCP 3.0 10 Gigabit Ethernet Adapter
- N225P - Dual-port OCP 3.0 10/25 Gigabit Ethernet Adapter
- N250G - Dual-port OCP 3.0 50 Gigabit Ethernet Adapter
- N410SG - Quad-port OCP 3.0 10 Gigabit Ethernet Adapter
- N410SGBT - Quad-port OCP 3.0 10 Gigabit Ethernet Adapter
- N425G - Quad-port OCP 3.0 10/25 Gigabit Ethernet Adapter
- N1100G - Single-port OCP 3.0 10/25/50/100 Gigabit Ethernet Adapter
- N2100G - Dual-port OCP 3.0 10/25/50/100 Gigabit Ethernet Adapter
- N2200G - Dual-port OCP 3.0 10/25/50/100/200 Gigabit Ethernet Adapter
- P150P - Single-port 50 Gigabit Ethernet Adapter
- P210P - Dual-port 10 Gigabit Ethernet Adapter
- P210TP - Dual-port 10 Gigabit Ethernet Adapter
- P225P - Dual-port 10/25 Gigabit Ethernet Adapter
- P410SG - Quad-port 10 Gigabit Ethernet Adapter
- P410SGBT - Quad-port 10 Gigabit Ethernet Adapter
- P425G - Quad-port 10/25 Gigabit Ethernet Adapter
- P1100G - Single-port 10/25/50/100 Gigabit Ethernet Adapter
- P2100G - Dual-port 10/25/50/100 Gigabit Ethernet Adapter
- P2200G - Dual-port 10/25/50/100/200 Gigabit Ethernet Adapter

Information about Ethernet adapters in the NetXtreme family of adapters can be found in the [NetXtreme® Brand](#) section of the [Broadcom website](#).

- **Broadcom BCM588XX Stingray Family of SmartNIC Adapters**

- PS410T - Quad-port 10 Gigabit Ethernet SmartNIC
- PS225 - Dual-port 25 Gigabit Ethernet SmartNIC
- PS250 - Dual-Port 50 Gigabit Ethernet SmartNIC

Information about the Stingray family of SmartNIC adapters can be found in the [Stingray® Brand](#) section of the [Broadcom website](#).

- **Broadcom StrataGX® BCM5871X Series of Communications Processors**

These ARM based processors target a broad range of networking applications including virtual CPE (vCPE) and NFV appliances, 10G service routers and gateways, control plane processing for Ethernet switches and network attached storage (NAS).

Information about the StrataGX family of adapters can be found in the [StrataGX® BCM58712](#) and [StrataGX® BCM58713](#) sections of the [Broadcom website](#).

CXGBE POLL MODE DRIVER

The CXGBE PMD (`librte_pmd_cxgbe`) provides poll mode driver support for **Chelsio Terminator** 10/25/40/100 Gbps family of adapters. CXGBE PMD has support for the latest Linux and FreeBSD operating systems.

CXGBEVF PMD provides poll mode driver support for SR-IOV Virtual functions and has support for the latest Linux operating systems.

More information can be found at [Chelsio Communications Official Website](#).

12.1 Features

CXGBE and CXGBEVF PMD has support for:

- Multiple queues for TX and RX
- Receiver Side Steering (RSS) Receiver Side Steering (RSS) on IPv4, IPv6, IPv4-TCP/UDP, IPv6-TCP/UDP. For 4-tuple, enabling 'RSS on TCP' and 'RSS on TCP + UDP' is supported.
- VLAN filtering
- Checksum offload
- Promiscuous mode
- All multicast mode
- Port hardware statistics
- Jumbo frames
- Flow API - Support for both Wildcard (LE-TCAM) and Exact (HASH) match filters.

12.2 Limitations

The Chelsio Terminator series of devices provide two/four ports but expose a single PCI bus address, thus, `librte_pmd_cxgbe` registers itself as a PCI driver that allocates one Ethernet device per detected port.

For this reason, one cannot whitelist/blacklist a single port without whitelisting/blacklisting the other ports on the same device.

12.3 Supported Chelsio T5 NICs

- 1G NICs: T502-BT
- 10G NICs: T520-BT, T520-CR, T520-LL-CR, T520-SO-CR, T540-CR
- 40G NICs: T580-CR, T580-LP-CR, T580-SO-CR
- Other T5 NICs: T522-CR

12.4 Supported Chelsio T6 NICs

- 25G NICs: T6425-CR, T6225-CR, T6225-LL-CR, T6225-SO-CR
- 100G NICs: T62100-CR, T62100-LP-CR, T62100-SO-CR

12.5 Supported SR-IOV Chelsio NICs

SR-IOV virtual functions are supported on all the Chelsio NICs listed in *Supported Chelsio T5 NICs* and *Supported Chelsio T6 NICs*.

12.6 Prerequisites

- Requires firmware version **1.23.4.0** and higher. Visit [Chelsio Download Center](#) to get latest firmware bundled with the latest Chelsio Unified Wire package.

For Linux, installing and loading the latest cxgb4 kernel driver from the Chelsio Unified Wire package should get you the latest firmware. More information can be obtained from the User Guide that is bundled with the Chelsio Unified Wire package.

For FreeBSD, the latest firmware obtained from the Chelsio Unified Wire package must be manually flashed via cxgbetool available in FreeBSD source repository.

Instructions on how to manually flash the firmware are given in section *Linux Installation* for Linux and section *FreeBSD Installation* for FreeBSD.

12.7 Pre-Installation Configuration

12.7.1 Config File Options

The following options can be modified in the `.config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_CXGBE_PMD` (default `y`)

Toggle compilation of `librte_pmd_cxgbe` driver.

Note: This controls compilation of both CXGBE and CXGBEVF PMD.

12.7.2 Runtime Options

The following devargs options can be enabled at runtime. They must be passed as part of EAL arguments. For example,

```
testpmd -w 02:00.4,keep_ovlan=1 -- -i
```

Common Runtime Options

- `keep_ovlan` (default 0)

Toggle behavior to keep/strip outer VLAN in Q-in-Q packets. If enabled, the outer VLAN tag is preserved in Q-in-Q packets. Otherwise, the outer VLAN tag is stripped in Q-in-Q packets.

- `tx_mode_latency` (default 0)

When set to 1, Tx doesn't wait for max number of packets to get coalesced and sends the packets immediately at the end of the current Tx burst. When set to 0, Tx waits across multiple Tx bursts until the max number of packets have been coalesced. In this case, Tx only sends the coalesced packets to hardware once the max coalesce limit has been reached.

CXGBE VF Only Runtime Options

- `force_link_up` (default 0)

When set to 1, CXGBEVF PMD always forces link as up for all VFs on underlying Chelsio NICs. This enables multiple VFs on the same NIC to send traffic to each other even when the physical link is down.

12.8 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

12.9 Linux

12.9.1 Linux Installation

Steps to manually install the latest firmware from the downloaded Chelsio Unified Wire package for Linux operating system are as follows:

1. Load the kernel module:

```
modprobe cxgb4
```

2. Use `ifconfig` to get the interface name assigned to Chelsio card:

```
ifconfig -a | grep "00:07:43"
```

Example output:

```
plp1      Link encap:Ethernet  HWaddr 00:07:43:2D:EA:C0
plp2      Link encap:Ethernet  HWaddr 00:07:43:2D:EA:C8
```

3. Install `cxgbtool`:

```
cd <path_to_uwire>/tools/cxgbtool
make install
```

4. Use cxgbtool to load the firmware config file onto the card:

```
cxgbtool plp1 loadcfg <path_to_uwire>/src/network/firmware/t5-config.txt
```

5. Use cxgbtool to load the firmware image onto the card:

```
cxgbtool plp1 loadfw <path_to_uwire>/src/network/firmware/t5fw-*.bin
```

6. Unload and reload the kernel module:

```
modprobe -r cxgb4
modprobe cxgb4
```

7. Verify with ethtool:

```
ethtool -i plp1 | grep "firmware"
```

Example output:

```
firmware-version: 1.23.4.0, TP 0.1.23.2
```

12.9.2 Running testpmd

This section demonstrates how to launch **testpmd** with Chelsio devices managed by `librte_pmd_cxgbe` in Linux operating system.

1. Load the kernel module:

```
modprobe cxgb4
```

2. Get the PCI bus addresses of the interfaces bound to cxgb4 driver:

```
dmesg | tail -2
```

Example output:

```
cxgb4 0000:02:00.4 plp1: renamed from eth0
cxgb4 0000:02:00.4 plp2: renamed from eth1
```

Note: Both the interfaces of a Chelsio 2-port adapter are bound to the same PCI bus address.

3. Unload the kernel module:

```
modprobe -ar cxgb4 csistor
```

4. Running testpmd

Follow instructions available in the document *compiling and testing a PMD for a NIC* to run testpmd.

Note: Currently, CXGBE PMD only supports the binding of PF4 for Chelsio NICs.

Example output:

```
[...]
EAL: PCI device 0000:02:00.4 on NUMA socket -1
EAL:  probe driver: 1425:5401 rte_cxgbe_pmd
EAL:  PCI memory mapped at 0x7fd7c0200000
EAL:  PCI memory mapped at 0x7fd77cdfd000
```

```

EAL:   PCI memory mapped at 0x7fd7c10b7000
PMD: rte_cxgbe_pmd: fw: 1.23.4.0, TP: 0.1.23.2
PMD: rte_cxgbe_pmd: Coming up as MASTER: Initializing adapter
Interactive-mode selected
Configuring Port 0 (socket 0)
Port 0: 00:07:43:2D:EA:C0
Configuring Port 1 (socket 0)
Port 1: 00:07:43:2D:EA:C8
Checking link statuses...
PMD: rte_cxgbe_pmd: Port0: passive DA port module inserted
PMD: rte_cxgbe_pmd: Port1: passive DA port module inserted
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>

```

Note: Flow control pause TX/RX is disabled by default and can be enabled via testpmd. Refer section [Enable/Disable Flow Control](#) for more details.

12.9.3 Configuring SR-IOV Virtual Functions

This section demonstrates how to enable SR-IOV virtual functions on Chelsio NICs and demonstrates how to run testpmd with SR-IOV virtual functions.

1. Load the kernel module:

```
modprobe cxgb4
```

2. Get the PCI bus addresses of the interfaces bound to cxgb4 driver:

```
dmesg | tail -2
```

Example output:

```

cxgb4 0000:02:00.4 p1p1: renamed from eth0
cxgb4 0000:02:00.4 p1p2: renamed from eth1

```

Note: Both the interfaces of a Chelsio 2-port adapter are bound to the same PCI bus address.

3. Use ifconfig to get the interface name assigned to Chelsio card:

```
ifconfig -a | grep "00:07:43"
```

Example output:

```

p1p1      Link encap:Ethernet  HWaddr 00:07:43:2D:EA:C0
p1p2      Link encap:Ethernet  HWaddr 00:07:43:2D:EA:C8

```

4. Bring up the interfaces:

```

ifconfig p1p1 up
ifconfig p1p2 up

```

5. Instantiate SR-IOV Virtual Functions. PF0..3 can be used for SR-IOV VFs. Multiple VFs can be instantiated on each of PF0..3. To instantiate one SR-IOV VF on each PF0 and PF1:

```

echo 1 > /sys/bus/pci/devices/0000\:02\:00.0/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000\:02\:00.1/sriov_numvfs

```

6. Get the PCI bus addresses of the virtual functions:

```
lspci | grep -i "Chelsio" | grep -i "VF"
```

Example output:

```
02:01.0 Ethernet controller: Chelsio Communications Inc T540-CR Unified Wire Ethernet Cont
02:01.1 Ethernet controller: Chelsio Communications Inc T540-CR Unified Wire Ethernet Cont
```

7. Running testpmd

Follow instructions available in the document *compiling and testing a PMD for a NIC* to bind virtual functions and run testpmd.

Example output:

```
[...]
EAL: PCI device 0000:02:01.0 on NUMA socket 0
EAL:   probe driver: 1425:5803 net_cxgbevf
PMD: rte_cxgbe_pmd: Firmware version: 1.23.4.0
PMD: rte_cxgbe_pmd: TP Microcode version: 0.1.23.2
PMD: rte_cxgbe_pmd: Chelsio rev 0
PMD: rte_cxgbe_pmd: No bootstrap loaded
PMD: rte_cxgbe_pmd: No Expansion ROM loaded
PMD: rte_cxgbe_pmd: 0000:02:01.0 Chelsio rev 0 1G/10GBASE-SFP
EAL: PCI device 0000:02:01.1 on NUMA socket 0
EAL:   probe driver: 1425:5803 net_cxgbevf
PMD: rte_cxgbe_pmd: Firmware version: 1.23.4.0
PMD: rte_cxgbe_pmd: TP Microcode version: 0.1.23.2
PMD: rte_cxgbe_pmd: Chelsio rev 0
PMD: rte_cxgbe_pmd: No bootstrap loaded
PMD: rte_cxgbe_pmd: No Expansion ROM loaded
PMD: rte_cxgbe_pmd: 0000:02:01.1 Chelsio rev 0 1G/10GBASE-SFP
Configuring Port 0 (socket 0)
Port 0: 06:44:29:44:40:00
Configuring Port 1 (socket 0)
Port 1: 06:44:29:44:40:10
Checking link statuses...
Done
testpmd>
```

12.10 FreeBSD

12.10.1 FreeBSD Installation

Steps to manually install the latest firmware from the downloaded Chelsio Unified Wire package for FreeBSD operating system are as follows:

1. Load the kernel module:

```
kldload if_cxgbe
```

2. Use dmesg to get the t5nex instance assigned to the Chelsio card:

```
dmesg | grep "t5nex"
```

Example output:

```
t5nex0: <Chelsio T520-CR> irq 16 at device 0.4 on pci2
cxl0: <port 0> on t5nex0
cxl1: <port 1> on t5nex0
t5nex0: PCIe x8, 2 ports, 14 MSI-X interrupts, 31 eq, 13 iq
```

In the example above, a Chelsio T520-CR card is bound to a t5nex0 instance.

3. Install cxgbetool from FreeBSD source repository:

```
cd <path_to_FreeBSD_source>/tools/tools/cxgbetool/
make && make install
```

4. Use cxgbetool to load the firmware image onto the card:

```
cxgbetool t5nex0 loadfw <path_to_uwire>/src/network/firmware/t5fw-*.bin
```

5. Unload and reload the kernel module:

```
kldunload if_cxgbe
kldload if_cxgbe
```

6. Verify with sysctl:

```
sysctl -a | grep "t5nex" | grep "firmware"
```

Example output:

```
dev.t5nex.0.firmware_version: 1.23.4.0
```

12.10.2 Running testpmd

This section demonstrates how to launch **testpmd** with Chelsio devices managed by `librte_pmd_cxgbe` in FreeBSD operating system.

1. Change to DPDK source directory where the target has been compiled in section *Driver compilation and testing*:

```
cd <DPDK-source-directory>
```

2. Copy the contigmem kernel module to /boot/kernel directory:

```
cp x86_64-native-freebsd-clang/kmod/contigmem.ko /boot/kernel/
```

3. Add the following lines to /boot/loader.conf:

```
# reserve 2 x 1G blocks of contiguous memory using contigmem driver
hw.contigmem.num_buffers=2
hw.contigmem.buffer_size=1073741824
# load contigmem module during boot process
contigmem_load="YES"
```

The above lines load the contigmem kernel module during boot process and allocate 2 x 1G blocks of contiguous memory to be used for DPDK later on. This is to avoid issues with potential memory fragmentation during later system up time, which may result in failure of allocating the contiguous memory required for the contigmem kernel module.

4. Restart the system and ensure the contigmem module is loaded successfully:

```
reboot
kldstat | grep "contigmem"
```

Example output:

```
2      1 0xffffffff817f1000 3118      contigmem.ko
```

5. Repeat step 1 to ensure that you are in the DPDK source directory.

6. Load the cxgbe kernel module:

```
kldload if_cxgbe
```

7. Get the PCI bus addresses of the interfaces bound to t5nex driver:

```
pciconf -l | grep "t5nex"
```

Example output:

```
t5nex0@pci0:2:0:4: class=0x020000 card=0x00001425 chip=0x54011425 rev=0x00
```

In the above example, the t5nex0 is bound to 2:0:4 bus address.

Note: Both the interfaces of a Chelsio 2-port adapter are bound to the same PCI bus address.

8. Unload the kernel module:

```
kldunload if_cxgbe
```

9. Set the PCI bus addresses to hw.nic_uio.bdfs kernel environment parameter:

```
kenv hw.nic_uio.bdfs="2:0:4"
```

This automatically binds 2:0:4 to nic_uio kernel driver when it is loaded in the next step.

Note: Currently, CXGBE PMD only supports the binding of PF4 for Chelsio NICs.

10. Load nic_uio kernel driver:

```
kldload ./x86_64-native-freebsd-clang/kmod/nic_uio.ko
```

11. Start testpmd with basic parameters:

```
./x86_64-native-freebsd-clang/app/testpmd -l 0-3 -n 4 -w 0000:02:00.4 -- -i
```

Example output:

```
[...]
EAL: PCI device 0000:02:00.4 on NUMA socket 0
EAL: probe driver: 1425:5401 rte_cxgbe_pmd
EAL: PCI memory mapped at 0x8007ec000
EAL: PCI memory mapped at 0x842800000
EAL: PCI memory mapped at 0x80086c000
PMD: rte_cxgbe_pmd: fw: 1.23.4.0, TP: 0.1.23.2
PMD: rte_cxgbe_pmd: Coming up as MASTER: Initializing adapter
Interactive-mode selected
Configuring Port 0 (socket 0)
Port 0: 00:07:43:2D:EA:C0
Configuring Port 1 (socket 0)
Port 1: 00:07:43:2D:EA:C8
Checking link statuses...
PMD: rte_cxgbe_pmd: Port0: passive DA port module inserted
PMD: rte_cxgbe_pmd: Port1: passive DA port module inserted
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

Note: Flow control pause TX/RX is disabled by default and can be enabled via testpmd. Refer section [Enable/Disable Flow Control](#) for more details.

12.11 Sample Application Notes

12.11.1 Enable/Disable Flow Control

Flow control pause TX/RX is disabled by default and can be enabled via testpmd as follows:

```
testpmd> set flow_ctrl rx on tx on 0 0 0 0 mac_ctrl_frame_fwd off autoneg on 0
testpmd> set flow_ctrl rx on tx on 0 0 0 0 mac_ctrl_frame_fwd off autoneg on 1
```

To disable again, run:

```
testpmd> set flow_ctrl rx off tx off 0 0 0 0 mac_ctrl_frame_fwd off autoneg off 0
testpmd> set flow_ctrl rx off tx off 0 0 0 0 mac_ctrl_frame_fwd off autoneg off 1
```

12.11.2 Jumbo Mode

There are two ways to enable sending and receiving of jumbo frames via testpmd. One method involves using the **mtu** command, which changes the mtu of an individual port without having to stop the selected port. Another method involves stopping all the ports first and then running **max-pkt-len** command to configure the mtu of all the ports with a single command.

- To configure each port individually, run the mtu command as follows:

```
testpmd> port config mtu 0 9000
testpmd> port config mtu 1 9000
```

- To configure all the ports at once, stop all the ports first and run the max-pkt-len command as follows:

```
testpmd> port stop all
testpmd> port config all max-pkt-len 9000
```

DPAA POLL MODE DRIVER

The DPAA NIC PMD (`librte_pmd_dpaa`) provides poll mode driver support for the inbuilt NIC found in the **NXP DPAA** SoC family.

More information can be found at [NXP Official Website](#).

13.1 NXP DPAA (Data Path Acceleration Architecture - Gen 1)

This section provides an overview of the NXP DPAA architecture and how it is integrated into the DPDK.

Contents summary

- DPAA overview
- DPAA driver architecture overview

13.1.1 DPAA Overview

Reference: [FSL DPAA Architecture](#).

The QorIQ Data Path Acceleration Architecture (DPAA) is a set of hardware components on specific QorIQ series multicore processors. This architecture provides the infrastructure to support simplified sharing of networking interfaces and accelerators by multiple CPU cores, and the accelerators themselves.

DPAA includes:

- Cores
- Network and packet I/O
- Hardware offload accelerators
- Infrastructure required to facilitate flow of packets between the components above

Infrastructure components are:

- The Queue Manager (QMan) is a hardware accelerator that manages frame queues. It allows CPUs and other accelerators connected to the SoC datapath to enqueue and dequeue ethernet frames, thus providing the infrastructure for data exchange among CPUs and datapath accelerators.
- The Buffer Manager (BMan) is a hardware buffer pool management block that allows software and accelerators on the datapath to acquire and release buffers in order to build frames.

Hardware accelerators are:

- SEC - Cryptographic accelerator
- PME - Pattern matching engine

The Network and packet I/O component:

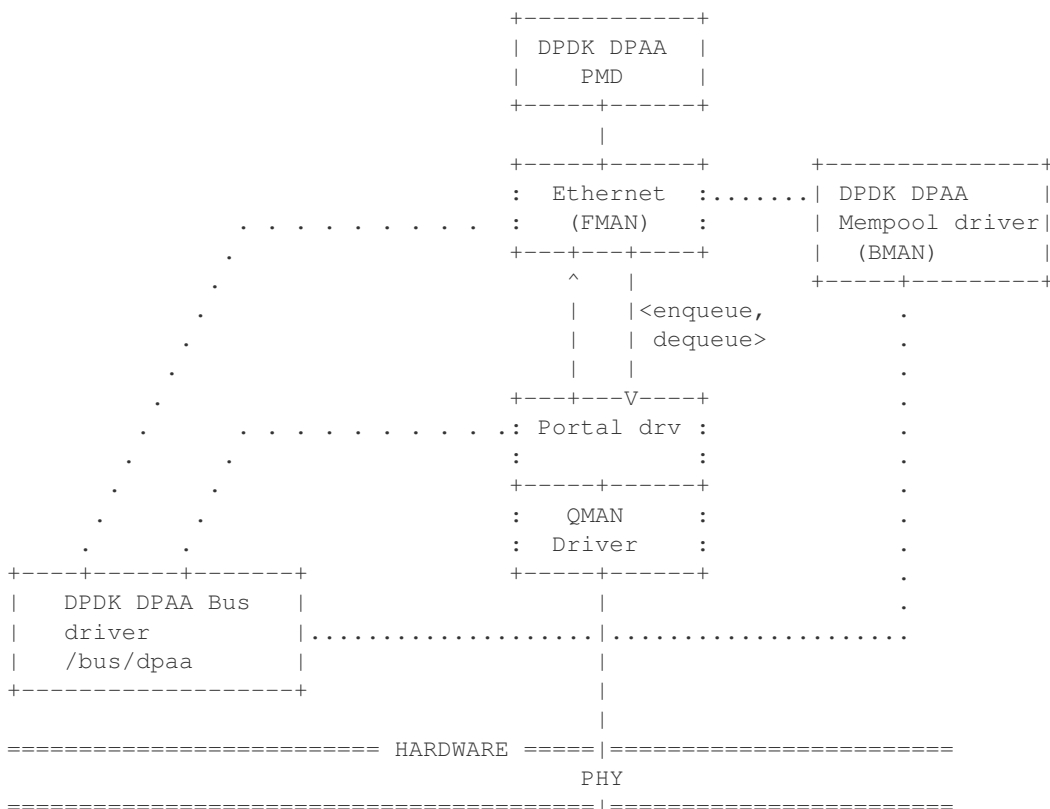
- The Frame Manager (FMan) is a key component in the DPAA and makes use of the DPAA infrastructure (QMan and BMan). FMan is responsible for packet distribution and policing. Each frame can be parsed, classified and results may be attached to the frame. This meta data can be used to select particular QMan queue, which the packet is forwarded to.

13.2 DPAA DPDK - Poll Mode Driver Overview

This section provides an overview of the drivers for DPAA:

- Bus driver and associated “DPAA infrastructure” drivers
- Functional object drivers (such as Ethernet).

Brief description of each driver is provided in layout below as well as in the following sections.



In the above representation, solid lines represent components which interface with DPDK RTE Framework and dotted lines represent DPAA internal components.

13.2.1 DPAA Bus driver

The DPAA bus driver is a `rte_bus` driver which scans the platform like bus. Key functions include:

- Scanning and parsing the various objects and adding them to their respective device list.

- Performing probe for available drivers against each scanned device
- Creating necessary ethernet instance before passing control to the PMD

13.2.2 DPAA NIC Driver (PMD)

DPAA PMD is traditional DPDK PMD which provides necessary interface between RTE framework and DPAA internal components/drivers.

- Once devices have been identified by DPAA Bus, each device is associated with the PMD
- PMD is responsible for implementing necessary glue layer between RTE APIs and lower level QMan and FMan blocks. The Ethernet driver is bound to a FMAN port and implements the interfaces needed to connect the DPAA network interface to the network stack. Each FMAN Port corresponds to a DPDK network interface.

Features

Features of the DPAA PMD are:

- Multiple queues for TX and RX
- Receive Side Scaling (RSS)
- Packet type information
- Checksum offload
- Promiscuous mode

13.2.3 DPAA Mempool Driver

DPAA has a hardware offloaded buffer pool manager, called BMan, or Buffer Manager.

- Using standard Mempools operations RTE API, the mempool driver interfaces with RTE to service each mempool creation, deletion, buffer allocation and deallocation requests.
- Each FMAN instance has a BMan pool attached to it during initialization. Each Tx frame can be automatically released by hardware, if allocated from this pool.

13.3 Whitelisting & Blacklisting

For blacklisting a DPAA device, following commands can be used.

```
<dpdk app> <EAL args> -b "dpaa_bus:fmX-macY" -- ...  
e.g. "dpaa_bus:fm1-mac4"
```

13.4 Supported DPAA SoCs

- LS1043A/LS1023A
- LS1046A/LS1026A

13.5 Prerequisites

See `../platform/dpaa` for setup information

- Follow the DPDK Getting Started Guide for Linux to setup the basic DPDK environment.

Note: Some part of dpaa bus code (qbman and fman - library) routines are dual licensed (BSD & GPLv2), however they are used as BSD in DPDK in userspace.

13.6 Pre-Installation Configuration

13.6.1 Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_DPAA_BUS` (default `y`)
Toggle compilation of the `librte_bus_dpaa` driver.
- `CONFIG_RTE_LIBRTE_DPAA_PMD` (default `y`)
Toggle compilation of the `librte_pmd_dpaa` driver.
- `CONFIG_RTE_LIBRTE_DPAA_DEBUG_DRIVER` (default `n`)
Toggles display of bus configurations and enables a debugging queue to fetch error (Rx/Tx) packets to driver. By default, packets with errors (like wrong checksum) are dropped by the hardware.
- `CONFIG_RTE_LIBRTE_DPAA_HWDEBUG` (default `n`)
Enables debugging of the Queue and Buffer Manager layer which interacts with the DPAA hardware.

13.6.2 Environment Variables

DPAA drivers uses the following environment variables to configure its state during application initialization:

- `DPAA_NUM_RX_QUEUES` (default 1)
This defines the number of Rx queues configured for an application, per port. Hardware would distribute across these many number of queues on Rx of packets. In case the application is configured to use lesser number of queues than configured above, it might result in packet loss (because of distribution).
- `DPAA_PUSH_QUEUES_NUMBER` (default 4)
This defines the number of High performance queues to be used for ethdev Rx. These queues use one private HW portal per queue configured, so they are limited in the system. The first configured ethdev queues will be automatically be assigned from the these high perf PUSH queues. Any queue configuration beyond that will be standard Rx queues. The application can choose to change their number if HW portals are limited. The valid values are from '0' to '4'. The values shall be

set to '0' if the application want to use eventdev with DPAA device. Currently these queues are not used for LS1023/LS1043 platform by default.

13.7 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

1. Running testpmd:

Follow instructions available in the document *compiling and testing a PMD for a NIC* to run testpmd.

Example output:

```
./arm64-dpaa-linux-gcc/testpmd -c 0xff -n 1 \  
-- -i --portmask=0x3 --nb-cores=1 --no-flush-rx  
  
.....  
EAL: Registered [pci] bus.  
EAL: Registered [dpaa] bus.  
EAL: Detected 4 lcore(s)  
.....  
EAL: dpaa: Bus scan completed  
.....  
Configuring Port 0 (socket 0)  
Port 0: 00:00:00:00:00:01  
Configuring Port 1 (socket 0)  
Port 1: 00:00:00:00:00:02  
.....  
Checking link statuses...  
Port 0 Link Up - speed 10000 Mbps - full-duplex  
Port 1 Link Up - speed 10000 Mbps - full-duplex  
Done  
testpmd>
```

13.8 Limitations

13.8.1 Platform Requirement

DPAA drivers for DPDK can only work on NXP SoCs as listed in the Supported DPAA SoCs.

13.8.2 Maximum packet length

The DPAA SoC family support a maximum of a 10240 jumbo frame. The value is fixed and cannot be changed. So, even when the `rxmode.max_rx_pkt_len` member of `struct rte_eth_conf` is set to a value lower than 10240, frames up to 10240 bytes can still reach the host interface.

13.8.3 Multiprocess Support

Current version of DPAA driver doesn't support multi-process applications where I/O is performed using secondary processes. This feature would be implemented in subsequent versions.

DPAA2 POLL MODE DRIVER

The DPAA2 NIC PMD (`librte_pmd_dpaa2`) provides poll mode driver support for the inbuilt NIC found in the **NXP DPAA2** SoC family.

More information can be found at [NXP Official Website](#).

14.1 NXP DPAA2 (Data Path Acceleration Architecture Gen2)

This section provides an overview of the NXP DPAA2 architecture and how it is integrated into the DPDK.

Contents summary

- DPAA2 overview
- Overview of DPAA2 objects
- DPAA2 driver architecture overview

14.1.1 DPAA2 Overview

Reference: [FSL MC BUS in Linux Kernel](#).

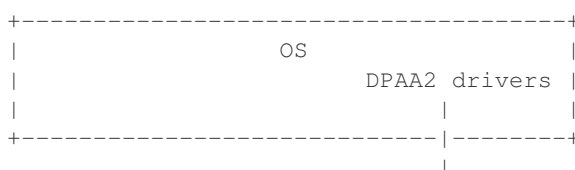
DPAA2 is a hardware architecture designed for high-speed network packet processing. DPAA2 consists of sophisticated mechanisms for processing Ethernet packets, queue management, buffer management, autonomous L2 switching, virtual Ethernet bridging, and accelerator (e.g. crypto) sharing.

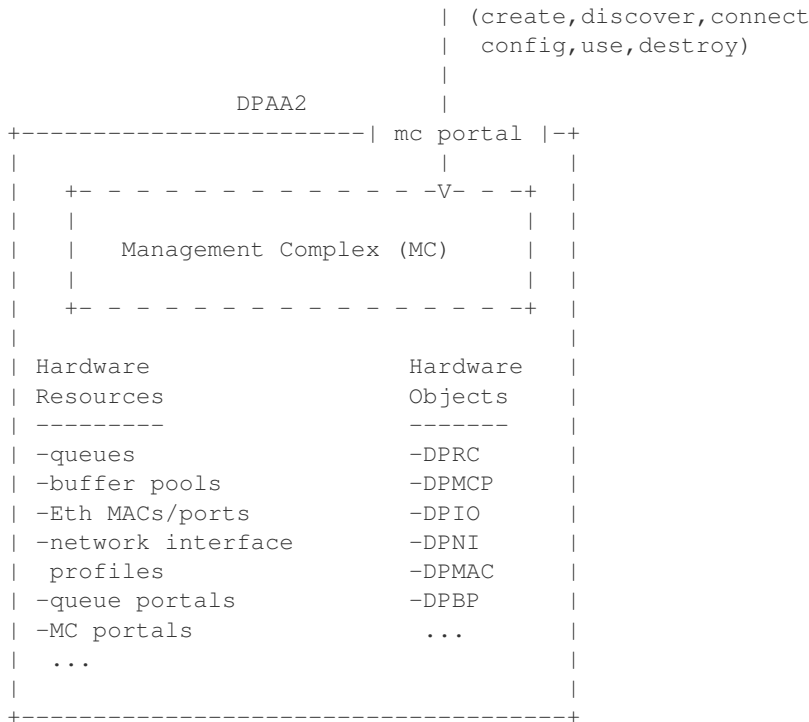
A DPAA2 hardware component called the Management Complex (or MC) manages the DPAA2 hardware resources. The MC provides an object-based abstraction for software drivers to use the DPAA2 hardware.

The MC uses DPAA2 hardware resources such as queues, buffer pools, and network ports to create functional objects/devices such as network interfaces, an L2 switch, or accelerator instances.

The MC provides memory-mapped I/O command interfaces (MC portals) which DPAA2 software drivers use to operate on DPAA2 objects:

The diagram below shows an overview of the DPAA2 resource management architecture:





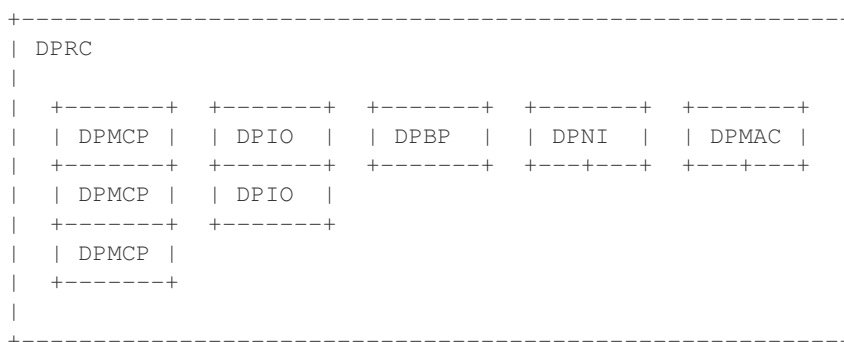
The MC mediates operations such as create, discover, connect, configuration, and destroy. Fast-path operations on data, such as packet transmit/receive, are not mediated by the MC and are done directly using memory mapped regions in DPIO objects.

14.1.2 Overview of DPAA2 Objects

The section provides a brief overview of some key DPAA2 objects. A simple scenario is described illustrating the objects involved in creating a network interfaces.

DPRC (Datapath Resource Container)

A DPRC is a container object that holds all the other types of DPAA2 objects. In the example diagram below there are 8 objects of 5 types (DPMCP, DPIO, DPBP, DPNI, and DPMAC) in the container.



From the point of view of an OS, a DPRC behaves similar to a plug and play bus, like PCI. DPRC commands can be used to enumerate the contents of the DPRC, discover the hardware objects present (including mappable regions and interrupts).

```

DPRC.1 (bus)
|
+-----+

```

```

      |           |           |           |           |
DPMCP.1  DPIO.1  DPBP.1  DPNI.1  DPMAC.1
DPMCP.2  DPIO.2
DPMCP.3

```

Hardware objects can be created and destroyed dynamically, providing the ability to hot plug/unplug objects in and out of the DPRC.

A DPRC has a mappable MMIO region (an MC portal) that can be used to send MC commands. It has an interrupt for status events (like hotplug).

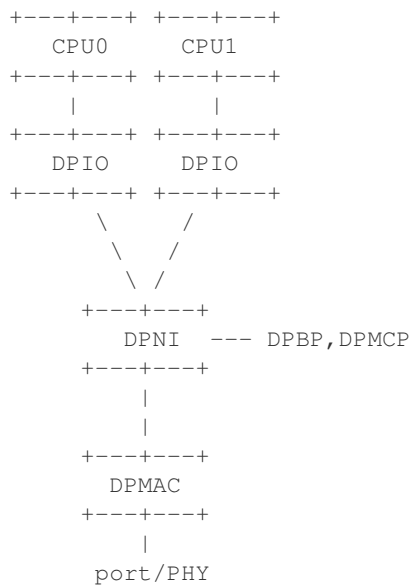
All objects in a container share the same hardware “isolation context”. This means that with respect to an IOMMU the isolation granularity is at the DPRC (container) level, not at the individual object level.

DPRCs can be defined statically and populated with objects via a config file passed to the MC when firmware starts it. There is also a Linux user space tool called “restool” that can be used to create/destroy containers and objects dynamically.

14.1.3 DPAA2 Objects for an Ethernet Network Interface

A typical Ethernet NIC is monolithic– the NIC device contains TX/RX queuing mechanisms, configuration mechanisms, buffer management, physical ports, and interrupts. DPAA2 uses a more granular approach utilizing multiple hardware objects. Each object provides specialized functions. Groups of these objects are used by software to provide Ethernet network interface functionality. This approach provides efficient use of finite hardware resources, flexibility, and performance advantages.

The diagram below shows the objects needed for a simple network interface configuration on a system with 2 CPUs.



Below the objects are described. For each object a brief description is provided along with a summary of the kinds of operations the object supports and a summary of key resources of the object (MMIO regions and IRQs).

DPMAC (Datapath Ethernet MAC): represents an Ethernet MAC, a hardware device that connects to an Ethernet PHY and allows physical transmission and reception of Ethernet frames.

- MMIO regions: none
- IRQs: DPNI link change

- commands: set link up/down, link config, get stats, IRQ config, enable, reset

DPNI (Datapath Network Interface): contains TX/RX queues, network interface configuration, and RX buffer pool configuration mechanisms. The TX/RX queues are in memory and are identified by queue number.

- MMIO regions: none
- IRQs: link state
- commands: port config, offload config, queue config, parse/classify config, IRQ config, enable, reset

DPIO (Datapath I/O): provides interfaces to enqueue and dequeue packets and do hardware buffer pool management operations. The DPAA2 architecture separates the mechanism to access queues (the DPIO object) from the queues themselves. The DPIO provides an MMIO interface to enqueue/dequeue packets. To enqueue something a descriptor is written to the DPIO MMIO region, which includes the target queue number. There will typically be one DPIO assigned to each CPU. This allows all CPUs to simultaneously perform enqueue/dequeue operations. DPIOs are expected to be shared by different DPAA2 drivers.

- MMIO regions: queue operations, buffer management
- IRQs: data availability, congestion notification, buffer pool depletion
- commands: IRQ config, enable, reset

DPBP (Datapath Buffer Pool): represents a hardware buffer pool.

- MMIO regions: none
- IRQs: none
- commands: enable, reset

DPMCP (Datapath MC Portal): provides an MC command portal. Used by drivers to send commands to the MC to manage objects.

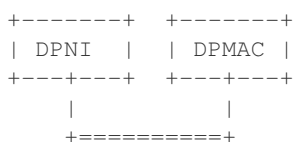
- MMIO regions: MC command portal
- IRQs: command completion
- commands: IRQ config, enable, reset

14.1.4 Object Connections

Some objects have explicit relationships that must be configured:

- DPNI <--> DPMAC
- DPNI <--> DPNI
- DPNI <--> L2-switch-port

A DPNI must be connected to something such as a DPMAC, another DPNI, or L2 switch port. The DPNI connection is made via a DPRC command.



- DPNI <--> DPBP

A network interface requires a ‘buffer pool’ (DPBP object) which provides a list of pointers to memory where received Ethernet data is to be copied. The Ethernet driver configures the DPBPs associated with the network interface.

14.1.5 Interrupts

All interrupts generated by DPAA2 objects are message interrupts. At the hardware level message interrupts generated by devices will normally have 3 components– 1) a non-spoofable ‘device-id’ expressed on the hardware bus, 2) an address, 3) a data value.

In the case of DPAA2 devices/objects, all objects in the same container/DPRC share the same ‘device-id’. For ARM-based SoC this is the same as the stream ID.

14.2 DPAA2 DPDK - Poll Mode Driver Overview

This section provides an overview of the drivers for DPAA2– 1) the bus driver and associated “DPAA2 infrastructure” drivers and 2) functional object drivers (such as Ethernet).

As described previously, a DPRC is a container that holds the other types of DPAA2 objects. It is functionally similar to a plug-and-play bus controller.

Each object in the DPRC is a Linux “device” and is bound to a driver. The diagram below shows the dpaa2 drivers involved in a networking scenario and the objects bound to each driver. A brief description of each driver follows.

A brief description of each driver is provided below.

14.2.1 DPAA2 bus driver

The DPAA2 bus driver is a `rte_bus` driver which scans the `fsl-mc` bus. Key functions include:

- Reading the container and setting up vfio group
- Scanning and parsing the various MC objects and adding them to their respective device list.

Additionally, it also provides the object driver for generic MC objects.

14.2.2 DPIO driver

The DPIO driver is bound to DPIO objects and provides services that allow other drivers such as the Ethernet driver to enqueue and dequeue data for their respective objects. Key services include:

- Data availability notifications
- Hardware queuing operations (enqueue and dequeue of data)
- Hardware buffer pool management

To transmit a packet the Ethernet driver puts data on a queue and invokes a DPIO API. For receive, the Ethernet driver registers a data availability notification callback. To dequeue a packet a DPIO API is used.

There is typically one DPIO object per physical CPU for optimum performance, allowing different CPUs to simultaneously enqueue and dequeue data.

The DPIO driver operates on behalf of all DPAA2 drivers active – Ethernet, crypto, compression, etc.

14.2.3 DPBP based Mempool driver

The DPBP driver is bound to a DPBP objects and provides services to create a hardware offloaded packet buffer mempool.

14.2.4 DPAA2 NIC Driver

The Ethernet driver is bound to a DPNI and implements the kernel interfaces needed to connect the DPAA2 network interface to the network stack.

Each DPNI corresponds to a DPDK network interface.

Features

Features of the DPAA2 PMD are:

- Multiple queues for TX and RX
- Receive Side Scaling (RSS)
- MAC/VLAN filtering
- Packet type information
- Checksum offload
- Promiscuous mode
- Multicast mode
- Port hardware statistics
- Jumbo frames
- Link flow control
- Scattered and gather for TX and RX

14.3 Supported DPAA2 SoCs

- LX2160A
- LS2084A/LS2044A
- LS2088A/LS2048A
- LS1088A/LS1048A

14.4 Prerequisites

See `../platform/dpaa2` for setup information

Currently supported by DPDK:

- NXP LSDK **19.08+**.
- MC Firmware version **10.18.0** and higher.
- Supported architectures: **arm64 LE**.
- Follow the DPDK Getting Started Guide for Linux to setup the basic DPDK environment.

Note: Some part of fslmc bus code (mc flib - object library) routines are dual licensed (BSD & GPLv2), however they are used as BSD in DPDK in userspace.

14.5 Pre-Installation Configuration

14.5.1 Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_FSLMC_BUS` (default `y`)
Toggle compilation of the `librte_bus_fslmc` driver.
- `CONFIG_RTE_LIBRTE_DPAA2_PMD` (default `y`)
Toggle compilation of the `librte_pmd_dpaa2` driver.
- `CONFIG_RTE_LIBRTE_DPAA2_DEBUG_DRIVER` (default `n`)
Toggle display of debugging messages/logic
- `CONFIG_RTE_LIBRTE_DPAA2_USE_PHYS_IOVA` (default `n`)
Toggle to use physical address vs virtual address for hardware accelerators.

14.6 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

1. Running testpmd:

Follow instructions available in the document *compiling and testing a PMD for a NIC* to run testpmd.

Example output:

```
./testpmd -c 0xff -n 1 -- -i --portmask=0x3 --nb-cores=1 --no-flush-rx

.....
EAL: Registered [pci] bus.
EAL: Registered [fslmc] bus.
```

```

EAL: Detected 8 lcore(s)
EAL: Probing VFIO support...
EAL: VFIO support initialized
.....
PMD: DPAA2: Processing Container = dprc.2
EAL: fslmc: DPRC contains = 51 devices
EAL: fslmc: Bus scan completed
.....
Configuring Port 0 (socket 0)
Port 0: 00:00:00:00:00:01
Configuring Port 1 (socket 0)
Port 1: 00:00:00:00:00:02
.....
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>

```

- Use dev arg option `drv_loopback=1` to loopback packets at driver level. Any packet received will be reflected back by the driver on same port. e.g. `fslmc:dpni.1,drv_loopback=1`
- Use dev arg option `drv_no_prefetch=1` to disable prefetching of the packet pull command which is issued in the previous cycle. e.g. `fslmc:dpni.1,drv_no_prefetch=1`

14.7 Enabling logs

For enabling logging for DPAA2 PMD, following log-level prefix can be used:

```
<dpdk app> <EAL args> --log-level=bus.fslmc:<level> -- ...
```

Using `bus.fslmc` as log matching criteria, all FSLMC bus logs can be enabled which are lower than logging level.

Or

```
<dpdk app> <EAL args> --log-level=pmd.net.dpaa2:<level> -- ...
```

Using `pmd.net.dpaa2` as log matching criteria, all PMD logs can be enabled which are lower than logging level.

14.8 Whitelisting & Blacklisting

For blacklisting a DPAA2 device, following commands can be used.

```
<dpdk app> <EAL args> -b "fslmc:dpni.x" -- ...
```

Where x is the device object id as configured in resource container.

14.9 Limitations

14.9.1 Platform Requirement

DPAA2 drivers for DPDK can only work on NXP SoCs as listed in the Supported DPAA2 SoCs.

14.9.2 Maximum packet length

The DPAA2 SoC family support a maximum of a 10240 jumbo frame. The value is fixed and cannot be changed. So, even when the `rxmode.max_rx_pkt_len` member of `struct rte_eth_conf` is set to a value lower than 10240, frames up to 10240 bytes can still reach the host interface.

14.9.3 Other Limitations

- RSS hash key cannot be modified.
- RSS RETA cannot be configured.

DRIVER FOR VM EMULATED DEVICES

The DPDK EM poll mode driver supports the following emulated devices:

- qemu-kvm emulated Intel® 82540EM Gigabit Ethernet Controller (qemu e1000 device)
- VMware* emulated Intel® 82545EM Gigabit Ethernet Controller
- VMware emulated Intel® 8274L Gigabit Ethernet Controller.

15.1 Validated Hypervisors

The validated hypervisors are:

- KVM (Kernel Virtual Machine) with Qemu, version 0.14.0
- KVM (Kernel Virtual Machine) with Qemu, version 0.15.1
- VMware ESXi 5.0, Update 1

15.2 Recommended Guest Operating System in Virtual Machine

The recommended guest operating system in a virtualized environment is:

- Fedora* 18 (64-bit)

For supported kernel versions, refer to the *DPDK Release Notes*.

15.3 Setting Up a KVM Virtual Machine

The following describes a target environment:

- Host Operating System: Fedora 14
- Hypervisor: KVM (Kernel Virtual Machine) with Qemu version, 0.14.0
- Guest Operating System: Fedora 14
- Linux Kernel Version: Refer to the DPDK Getting Started Guide
- Target Applications: testpmd

The setup procedure is as follows:

1. Download qemu-kvm-0.14.0 from <http://sourceforge.net/projects/kvm/files/qemu-kvm/> and install it in the Host OS using the following steps:

When using a recent kernel (2.6.25+) with kvm modules included:

```
tar xzf qemu-kvm-release.tar.gz cd qemu-kvm-release
./configure --prefix=/usr/local/kvm
make
sudo make install
sudo /sbin/modprobe kvm-intel
```

When using an older kernel or a kernel from a distribution without the kvm modules, you must download (from the same link), compile and install the modules yourself:

```
tar xjf kvm-kmod-release.tar.bz2
cd kvm-kmod-release
./configure
make
sudo make install
sudo /sbin/modprobe kvm-intel
```

Note that qemu-kvm installs in the /usr/local/bin directory.

For more details about KVM configuration and usage, please refer to: <http://www.linux-kvm.org/page/HOWTO1>.

2. Create a Virtual Machine and install Fedora 14 on the Virtual Machine. This is referred to as the Guest Operating System (Guest OS).
3. Start the Virtual Machine with at least one emulated e1000 device.

Note: The Qemu provides several choices for the emulated network device backend. Most commonly used is a TAP networking backend that uses a TAP networking device in the host. For more information about Qemu supported networking backends and different options for configuring networking at Qemu, please refer to:

- <http://www.linux-kvm.org/page/Networking>
- <http://wiki.qemu.org/Documentation/Networking>
- <http://qemu.weilnetz.de/qemu-doc.html>

For example, to start a VM with two emulated e1000 devices, issue the following command:

```
/usr/local/kvm/bin/qemu-system-x86_64 -cpu host -smp 4 -hda qemu1.raw -m 1024
-net nic,model=e1000,vlan=1,macaddr=DE:AD:1E:00:00:01
-net tap,vlan=1,ifname=tapvm01,script=no,downscript=no
-net nic,model=e1000,vlan=2,macaddr=DE:AD:1E:00:00:02
-net tap,vlan=2,ifname=tapvm02,script=no,downscript=no
```

where:

- -m = memory to assign
- -smp = number of smp cores
- -hda = virtual disk image

This command starts a new virtual machine with two emulated 82540EM devices, backed up with two TAP networking host interfaces, tapvm01 and tapvm02.

```
# ip tuntap show
tapvm01: tap
tapvm02: tap
```

4. Configure your TAP networking interfaces using ip/ifconfig tools.
5. Log in to the guest OS and check that the expected emulated devices exist:

```
# lspci -d 8086:100e
00:04.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 03)
00:05.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 03)
```

6. Install the DPDK and run testpmd.

15.4 Known Limitations of Emulated Devices

The following are known limitations:

1. The Qemu e1000 RX path does not support multiple descriptors/buffers per packet. Therefore, `rte_mbuf` should be big enough to hold the whole packet. For example, to allow testpmd to receive jumbo frames, use the following:

`testpmd [options] --mbuf-size=<your-max-packet-size>`
2. Qemu e1000 does not validate the checksum of incoming packets.
3. Qemu e1000 only supports one interrupt source, so link and Rx interrupt should be exclusive.
4. Qemu e1000 does not support interrupt auto-clear, application should disable interrupt immediately when woken up.

ENA POLL MODE DRIVER

The ENA PMD is a DPDK poll-mode driver for the Amazon Elastic Network Adapter (ENA) family.

16.1 Overview

The ENA driver exposes a lightweight management interface with a minimal set of memory mapped registers and an extendable command set through an Admin Queue.

The driver supports a wide range of ENA adapters, is link-speed independent (i.e., the same driver is used for 10GbE, 25GbE, 40GbE, etc.), and it negotiates and supports an extendable feature set.

ENA adapters allow high speed and low overhead Ethernet traffic processing by providing a dedicated Tx/Rx queue pair per CPU core.

The ENA driver supports industry standard TCP/IP offload features such as checksum offload and TCP transmit segmentation offload (TSO).

Receive-side scaling (RSS) is supported for multi-core scaling.

Some of the ENA devices support a working mode called Low-latency Queue (LLQ), which saves several more microseconds.

16.2 Management Interface

ENA management interface is exposed by means of:

- Device Registers
- Admin Queue (AQ) and Admin Completion Queue (ACQ)

ENA device memory-mapped PCIe space for registers (MMIO registers) are accessed only during driver initialization and are not involved in further normal device operation.

AQ is used for submitting management commands, and the results/responses are reported asynchronously through ACQ.

ENA introduces a very small set of management commands with room for vendor-specific extensions. Most of the management operations are framed in a generic Get/Set feature command.

The following admin queue commands are supported:

- Create I/O submission queue
- Create I/O completion queue

- Destroy I/O submission queue
- Destroy I/O completion queue
- Get feature
- Set feature
- Get statistics

Refer to `ena_admin_defs.h` for the list of supported Get/Set Feature properties.

16.3 Data Path Interface

I/O operations are based on Tx and Rx Submission Queues (Tx SQ and Rx SQ correspondingly). Each SQ has a completion queue (CQ) associated with it.

The SQs and CQs are implemented as descriptor rings in contiguous physical memory.

Refer to `ena_eth_io_defs.h` for the detailed structure of the descriptor

The driver supports multi-queue for both Tx and Rx.

16.4 Configuration information

DPDK Configuration Parameters

The following configuration options are available for the ENA PMD:

- **CONFIG_RTE_LIBRTE_ENA_PMD** (default y): Enables or disables inclusion of the ENA PMD driver in the DPDK compilation.
- **CONFIG_RTE_LIBRTE_ENA_DEBUG_RX** (default n): Enables or disables debug logging of RX logic within the ENA PMD driver.
- **CONFIG_RTE_LIBRTE_ENA_DEBUG_TX** (default n): Enables or disables debug logging of TX logic within the ENA PMD driver.
- **CONFIG_RTE_LIBRTE_ENA_COM_DEBUG** (default n): Enables or disables debug logging of low level tx/rx logic in `ena_com(base)` within the ENA PMD driver.

ENA Configuration Parameters

- **Number of Queues**

This is the requested number of queues upon initialization, however, the actual number of receive and transmit queues to be created will be the minimum between the maximal number supported by the device and number of queues requested.

- **Size of Queues**

This is the requested size of receive/transmit queues, while the actual size will be the minimum between the requested size and the maximal receive/transmit supported by the device.

16.5 Building DPDK

See the DPDK Getting Started Guide for Linux for instructions on how to build DPDK.

By default the ENA PMD library will be built into the DPDK library.

For configuring and using UIO and VFIO frameworks, please also refer the documentation that comes with DPDK suite.

16.6 Supported ENA adapters

Current ENA PMD supports the following ENA adapters including:

- `1d0f:ec20` - ENA VF
- `1d0f:ec21` - ENA VF with LLQ support

16.7 Supported Operating Systems

Any Linux distribution fulfilling the conditions described in `System Requirements` section of the DPDK documentation or refer to *DPDK Release Notes*.

16.8 Supported features

- MTU configuration
- Jumbo frames up to 9K
- IPv4/TCP/UDP checksum offload
- TSO offload
- Multiple receive and transmit queues
- RSS hash
- RSS indirection table configuration
- Low Latency Queue for Tx
- Basic and extended statistics
- LSC event notification
- Watchdog (requires handling of timers in the application)
- Device reset upon failure

16.9 Prerequisites

1. Prepare the system as recommended by DPDK suite. This includes environment variables, hugepages configuration, tool-chains and configuration.

2. ENA PMD can operate with `vfio-pci` (*) or `igb_uio` driver.

(*) ENAv2 hardware supports Low Latency Queue v2 (LLQv2). This feature reduces the latency of the packets by pushing the header directly through the PCI to the device, before the DMA is even triggered. For proper work kernel PCI driver must support write combining (WC). In mainline version of `igb_uio` (in DPDK repo) it must be enabled by loading module with `wc_activate=1` flag (example below). However, mainline's `vfio-pci` driver in kernel doesn't have WC support yet (planned to be added). If `vfio-pci` used user should be either turn off ENAv2 (to avoid performance impact) or recompile `vfio-pci` driver with patch provided in [amzn-github](#).

3. Insert `vfio-pci` or `igb_uio` kernel module using the command `modprobe vfio-pci` or `modprobe uio; insmod igb_uio.ko wc_activate=1` respectively.
4. For `vfio-pci` users only: Please make sure that IOMMU is enabled in your system, or use `vfio` driver in `noiommu` mode:

```
echo 1 > /sys/module/vfio/parameters/enable_unsafe_noiommu_mode
```

5. Bind the intended ENA device to `vfio-pci` or `igb_uio` module.

At this point the system should be ready to run DPDK applications. Once the application runs to completion, the ENA can be detached from `igb_uio` if necessary.

16.10 Usage example

Follow instructions available in the document *[compiling and testing a PMD for a NIC](#)* to launch `testpmd` with Amazon ENA devices managed by `librte_pmd_ena`.

Example output:

```
[...]
EAL: PCI device 0000:00:06.0 on NUMA socket -1
EAL:   Invalid NUMA socket, default to 0
EAL:   probe driver: 1d0f:ec20 net_ena

Interactive-mode selected
testpmd: create a new mbuf pool <mbuf_pool_socket_0>: n=171456, size=2176, socket=0
testpmd: preferred mempool ops selected: ring_mp_mc
Warning! port-topology=paired and odd forward ports number, the last port will pair with itself
Configuring Port 0 (socket 0)
Port 0: 00:00:00:11:00:01
Checking link statuses...

Done
testpmd>
```


ENETC POLL MODE DRIVER

The ENETC NIC PMD (**librte_pmd_enetc**) provides poll mode driver support for the inbuilt NIC found in the **NXP LS1028** SoC.

More information can be found at [NXP Official Website](#).

17.1 ENETC

This section provides an overview of the NXP ENETC and how it is integrated into the DPDK.

Contents summary

- ENETC overview
- ENETC features
- PCI bus driver
- NIC driver
- Supported ENETC SoCs
- Prerequisites
- Driver compilation and testing

17.1.1 ENETC Overview

ENETC is a PCI Integrated End Point (IEP). IEP implements peripheral devices in an SoC such that software sees them as PCIe device. ENETC is an evolution of BDR(Buffer Descriptor Ring) based networking IPs.

This infrastructure simplifies adding support for IEP and facilitates in following:

- Device discovery and location
- Resource requirement discovery and allocation (e.g. interrupt assignment, device register address)
- Event reporting

17.1.2 ENETC Features

- Link Status
- Packet type information

- Basic stats
- Promiscuous
- Multicast
- Jumbo packets
- Queue Start/Stop
- Deferred Queue Start
- CRC offload

17.1.3 NIC Driver (PMD)

ENETC PMD is traditional DPDK PMD which provides necessary interface between RTE framework and ENETC internal drivers.

- Driver registers the device vendor table in PCI subsystem.
- RTE framework scans the PCI bus for connected devices.
- This scanning will invoke the probe function of ENETC driver.
- The probe function will set the basic device registers and also setups BD rings.
- On packet Rx the respective BD Ring status bit is set which is then used for packet processing.
- Then Tx is done first followed by Rx.

17.1.4 Supported ENETC SoCs

- LS1028

17.1.5 Prerequisites

There are three main pre-requisites for executing ENETC PMD on a ENETC compatible board:

1. ARM 64 Tool Chain

For example, the **aarch64** [Linaro Toolchain](#).

2. Linux Kernel

It can be obtained from [NXP's Github hosting](#).

3. Rootfile system

Any *aarch64* supporting filesystem can be used. For example, Ubuntu 16.04 LTS (Xenial) or 18.04 (Bionic) userland which can be obtained from [here](#).

The following dependencies are not part of DPDK and must be installed separately:

- **NXP Linux LSDK**

NXP Layerscape software development kit (LSDK) includes support for family of QorIQ® ARM-Architecture-based system on chip (SoC) processors and corresponding boards.

It includes the Linux board support packages (BSPs) for NXP SoCs, a fully operational tool chain, kernel and board specific modules.

LSDK and related information can be obtained from: [LSDK](#)

17.1.6 Driver compilation and testing

Follow instructions available in the document *compiling and testing a PMD for a NIC* to launch **testpmd**

To compile in performance mode, please set `CONFIG_RTE_CACHE_LINE_SIZE=64`

ENIC POLL MODE DRIVER

ENIC PMD is the DPDK poll-mode driver for the Cisco System Inc. VIC Ethernet NICs. These adapters are also referred to as vNICs below. If you are running or would like to run DPDK software applications on Cisco UCS servers using Cisco VIC adapters the following documentation is relevant.

18.1 How to obtain ENIC PMD integrated DPDK

ENIC PMD support is integrated into the DPDK suite. `dpdk-<version>.tar.gz` should be downloaded from <https://core.dpdk.org/download/>

18.2 Configuration information

- **DPDK Configuration Parameters**

The following configuration options are available for the ENIC PMD:

- **CONFIG_RTE_LIBRTE_ENIC_PMD** (default y): Enables or disables inclusion of the ENIC PMD driver in the DPDK compilation.

- **vNIC Configuration Parameters**

- **Number of Queues**

The maximum number of receive queues (RQs), work queues (WQs) and completion queues (CQs) are configurable on a per vNIC basis through the Cisco UCS Manager (CIMC or UCSM).

These values should be configured as follows:

- * The number of WQs should be greater or equal to the value of the expected `nb_tx_q` parameter in the call to `rte_eth_dev_configure()`
- * The number of RQs configured in the vNIC should be greater or equal to *twice* the value of the expected `nb_rx_q` parameter in the call to `rte_eth_dev_configure()`. With the addition of Rx scatter, a pair of RQs on the vnic is needed for each receive queue used by DPDK, even if Rx scatter is not being used. Having a vNIC with only 1 RQ is not a valid configuration, and will fail with an error message.
- * The number of CQs should set so that there is one CQ for each WQ, and one CQ for each pair of RQs.

For example: If the application requires 3 Rx queues, and 3 Tx queues, the vNIC should be configured to have at least 3 WQs, 6 RQs (3 pairs), and 6 CQs (3 for use by WQs + 3 for use by the 3 pairs of RQs).

– Size of Queues

Likewise, the number of receive and transmit descriptors are configurable on a per-vNIC basis via the UCS Manager and should be greater than or equal to the `nb_rx_desc` and `nb_tx_desc` parameters expected to be used in the calls to `rte_eth_rx_queue_setup()` and `rte_eth_tx_queue_setup()` respectively. An application requesting more than the set size will be limited to that size.

Unless there is a lack of resources due to creating many vNICs, it is recommended that the WQ and RQ sizes be set to the maximum. This gives the application the greatest amount of flexibility in its queue configuration.

* *Note:* Since the introduction of Rx scatter, for performance reasons, this PMD uses two RQs on the vNIC per receive queue in DPDK. One RQ holds descriptors for the start of a packet, and the second RQ holds the descriptors for the rest of the fragments of a packet. This means that the `nb_rx_desc` parameter to `rte_eth_rx_queue_setup()` can be a greater than 4096. The exact amount will depend on the size of the mbufs being used for receives, and the MTU size.

For example: If the mbuf size is 2048, and the MTU is 9000, then receiving a full size packet will take 5 descriptors, 1 from the start-of-packet queue, and 4 from the second queue. Assuming that the RQ size was set to the maximum of 4096, then the application can specify up to $1024 + 4096$ as the `nb_rx_desc` parameter to `rte_eth_rx_queue_setup()`.

– Interrupts

At least one interrupt per vNIC interface should be configured in the UCS manager regardless of the number receive/transmit queues. The ENIC PMD uses this interrupt to get information about link status and errors in the fast path.

In addition to the interrupt for link status and errors, when using Rx queue interrupts, increase the number of configured interrupts so that there is at least one interrupt for each Rx queue. For example, if the app uses 3 Rx queues and wants to use per-queue interrupts, configure 4 (3 + 1) interrupts.

– Receive Side Scaling

In order to fully utilize RSS in DPDK, enable all RSS related settings in CIMC or UCSM. These include the following items listed under Receive Side Scaling: TCP, IPv4, TCP-IPv4, IPv6, TCP-IPv6, IPv6 Extension, TCP-IPv6 Extension.

18.3 SR-IOV mode utilization

UCS blade servers configured with dynamic vNIC connection policies in UCSM are capable of supporting SR-IOV. SR-IOV virtual functions (VFs) are specialized vNICs, distinct from regular Ethernet vNICs. These VFs can be directly assigned to virtual machines (VMs) as ‘passthrough’ devices.

In UCS, SR-IOV VFs require the use of the Cisco Virtual Machine Fabric Extender (VM-FEX), which gives the VM a dedicated interface on the Fabric Interconnect (FI). Layer 2 switching is done at the FI. This may eliminate the requirement for software switching on the host to route intra-host VM traffic.

Please refer to [Creating a Dynamic vNIC Connection Policy](#) for information on configuring SR-IOV adapter policies and port profiles using UCSM.

Once the policies are in place and the host OS is rebooted, VFs should be visible on the host, E.g.:

```
# lspci | grep Cisco | grep Ethernet
0d:00.0 Ethernet controller: Cisco Systems Inc VIC Ethernet NIC (rev a2)
0d:00.1 Ethernet controller: Cisco Systems Inc VIC SR-IOV VF (rev a2)
0d:00.2 Ethernet controller: Cisco Systems Inc VIC SR-IOV VF (rev a2)
0d:00.3 Ethernet controller: Cisco Systems Inc VIC SR-IOV VF (rev a2)
0d:00.4 Ethernet controller: Cisco Systems Inc VIC SR-IOV VF (rev a2)
0d:00.5 Ethernet controller: Cisco Systems Inc VIC SR-IOV VF (rev a2)
0d:00.6 Ethernet controller: Cisco Systems Inc VIC SR-IOV VF (rev a2)
0d:00.7 Ethernet controller: Cisco Systems Inc VIC SR-IOV VF (rev a2)
```

Enable Intel IOMMU on the host and install KVM and libvirt, and reboot again as required. Then, using libvirt, create a VM instance with an assigned device. Below is an example interface block (part of the domain configuration XML) that adds the host VF 0d:00:01 to the VM. `profileid='pp-vlan-25'` indicates the port profile that has been configured in UCSM.

```
<interface type='hostdev' managed='yes'>
  <mac address='52:54:00:ac:ff:b6' />
  <driver name='vfio' />
  <source>
    <address type='pci' domain='0x0000' bus='0x0d' slot='0x00' function='0x1' />
  </source>
  <virtualport type='802.1Qbh'>
    <parameters profileid='pp-vlan-25' />
  </virtualport>
</interface>
```

Alternatively, the configuration can be done in a separate file using the `network` keyword. These methods are described in the libvirt documentation for [Network XML format](#).

When the VM instance is started, libvirt will bind the host VF to `vfio`, complete provisioning on the FI and bring up the link.

Note: It is not possible to use a VF directly from the host because it is not fully provisioned until libvirt brings up the VM that it is assigned to.

In the VM instance, the VF will now be visible. E.g., here the VF 00:04.0 is seen on the VM instance and should be available for binding to a DPDK.

```
# lspci | grep Ether
00:04.0 Ethernet controller: Cisco Systems Inc VIC SR-IOV VF (rev a2)
```

Follow the normal DPDK install procedure, binding the VF to either `igb_uio` or `vfio` in non-IOMMU mode.

In the VM, the kernel `enic` driver may be automatically bound to the VF during boot. Unbinding it currently hangs due to a known issue with the driver. To work around the issue, blacklist the `enic` module as follows. Please see [Limitations](#) for limitations in the use of SR-IOV.

```
# cat /etc/modprobe.d/enic.conf
blacklist enic

# dracut --force
```

Note: Passthrough does not require SR-IOV. If VM-FEX is not desired, the user may create as many

regular vNICs as necessary and assign them to VMs as passthrough devices. Since these vNICs are not SR-IOV VFs, using them as passthrough devices do not require libvirt, port profiles, and VM-FEX.

18.4 Generic Flow API support

Generic Flow API (also called “rte_flow” API) is supported. More advanced capabilities are available when “Advanced Filtering” is enabled on the adapter. Advanced filtering was added to 1300 series VIC firmware starting with version 2.0.13 for C-series UCS servers and version 3.1.2 for UCSM managed blade servers. Advanced filtering is available on 1400 series adapters and beyond. To enable advanced filtering, the ‘Advanced filter’ radio button should be selected via CIMC or UCSM followed by a reboot of the server.

- **1200 series VICs**

5-tuple exact flow support for 1200 series adapters. This allows:

- Attributes: ingress
- Items: ipv4, ipv6, udp, tcp (must exactly match src/dst IP addresses and ports and all must be specified)
- Actions: queue and void
- Selectors: ‘is’

- **1300 and later series VICS with advanced filters disabled**

With advanced filters disabled, an IPv4 or IPv6 item must be specified in the pattern.

- Attributes: ingress
- Items: eth, vlan, ipv4, ipv6, udp, tcp, vxlan, inner eth, vlan, ipv4, ipv6, udp, tcp
- Actions: queue and void
- Selectors: ‘is’, ‘spec’ and ‘mask’. ‘last’ is not supported
- In total, up to 64 bytes of mask is allowed across all headers

- **1300 and later series VICS with advanced filters enabled**

- Attributes: ingress
- Items: eth, vlan, ipv4, ipv6, udp, tcp, vxlan, raw, inner eth, vlan, ipv4, ipv6, udp, tcp
- Actions: queue, mark, drop, flag, rss, passthru, and void
- Selectors: ‘is’, ‘spec’ and ‘mask’. ‘last’ is not supported
- In total, up to 64 bytes of mask is allowed across all headers

- **1400 and later series VICs with Flow Manager API enabled**

- Attributes: ingress, egress
- Items: eth, vlan, ipv4, ipv6, sctp, udp, tcp, vxlan, raw, inner eth, vlan, ipv4, ipv6, sctp, udp, tcp
- Ingress Actions: count, drop, flag, jump, mark, port_id, passthru, queue, rss, vxlan_decap, vxlan_encap, and void

- Egress Actions: count, drop, jump, passthru, vxlan_encap, and void
- Selectors: ‘is’, ‘spec’ and ‘mask’. ‘last’ is not supported
- In total, up to 64 bytes of mask is allowed across all headers

The VIC performs packet matching after applying VLAN strip. If VLAN stripping is enabled, EtherType in the ETH item corresponds to the stripped VLAN header’s EtherType. Stripping does not affect the VLAN item. TCI and EtherType in the VLAN item are matched against those in the (stripped) VLAN header whether stripping is enabled or disabled.

More features may be added in future firmware and new versions of the VIC. Please refer to the release notes.

18.5 Overlay Offload

Recent hardware models support overlay offload. When enabled, the NIC performs the following operations for VXLAN, NVGRE, and GENEVE packets. In all cases, inner and outer packets can be IPv4 or IPv6.

- TSO for VXLAN and GENEVE packets.

Hardware supports NVGRE TSO, but DPDK currently has no NVGRE offload flags.

- Tx checksum offloads.

The NIC fills in IPv4/UDP/TCP checksums for both inner and outer packets.

- Rx checksum offloads.

The NIC validates IPv4/UDP/TCP checksums of both inner and outer packets. Good checksum flags (e.g. `PKT_RX_L4_CKSUM_GOOD`) indicate that the inner packet has the correct checksum, and if applicable, the outer packet also has the correct checksum. Bad checksum flags (e.g. `PKT_RX_L4_CKSUM_BAD`) indicate that the inner and/or outer packets have invalid checksum values.

- Inner Rx packet type classification

PMD sets inner L3/L4 packet types (e.g. `RTE_PTYPE_INNER_L4_TCP`), and `RTE_PTYPE_TUNNEL_GRENAT` to indicate that the packet is tunneled. PMD does not set L3/L4 packet types for outer packets.

- Inner RSS

RSS hash calculation, therefore queue selection, is done on inner packets.

In order to enable overlay offload, the ‘Enable VXLAN’ box should be checked via CIMC or UCSM followed by a reboot of the server. When PMD successfully enables overlay offload, it prints the following message on the console.

```
Overlay offload is enabled
```

By default, PMD enables overlay offload if hardware supports it. To disable it, set `devargs` parameter `disable-overlay=1`. For example:

```
-w 12:00.0,disable-overlay=1
```

By default, the NIC uses 4789 as the VXLAN port. The user may change it through `rte_eth_dev_udp_tunnel_port_{add,delete}`. However, as the current NIC has a single VXLAN port number, the user cannot configure multiple port numbers.

Geneve headers with non-zero options are not supported by default. To use Geneve with options, update the VIC firmware to the latest version and then set `devargs` parameter `geneve-opt=1`. When Geneve with options is enabled, flow API cannot be used as the features are currently mutually exclusive. When this feature is successfully enabled, PMD prints the following message.

```
Geneve with options is enabled
```

18.6 Ingress VLAN Rewrite

VIC adapters can tag, untag, or modify the VLAN headers of ingress packets. The ingress VLAN rewrite mode controls this behavior. By default, it is set to pass-through, where the NIC does not modify the VLAN header in any way so that the application can see the original header. This mode is sufficient for many applications, but may not be suitable for others. Such applications may change the mode by setting `devargs` parameter `ig-vlan-rewrite` to one of the following.

- `pass`: Pass-through mode. The NIC does not modify the VLAN header. This is the default mode.
- `priority`: Priority-tag default VLAN mode. If the ingress packet is tagged with the default VLAN, the NIC replaces its VLAN header with the priority tag (VLAN ID 0).
- `trunk`: Default trunk mode. The NIC tags untagged ingress packets with the default VLAN. Tagged ingress packets are not modified. To the application, every packet appears as tagged.
- `untag`: Untag default VLAN mode. If the ingress packet is tagged with the default VLAN, the NIC removes or untags its VLAN header so that the application sees an untagged packet. As a result, the default VLAN becomes *untagged*. This mode can be useful for applications such as OVS-DPDK performance benchmarks that utilize only the default VLAN and want to see only untagged packets.

18.7 Vectorized Rx Handler

ENIC PMD includes a version of the receive handler that is vectorized using AVX2 SIMD instructions. It is meant for bulk, throughput oriented workloads where reducing cycles/packet in PMD is a priority. In order to use the vectorized handler, take the following steps.

- Use a recent version of gcc, icc, or clang and build 64-bit DPDK. If the compiler is known to support AVX2, DPDK build system automatically compiles the vectorized handler. Otherwise, the handler is not available.
- Set `devargs` parameter `enable-avx2-rx=1` to explicitly request that PMD consider the vectorized handler when selecting the receive handler. For example:

```
-w 12:00.0,enable-avx2-rx=1
```

As the current implementation is intended for field trials, by default, the vectorized handler is not considered (`enable-avx2-rx=0`).

- Run on a UCS M4 or later server with CPUs that support AVX2.

PMD selects the vectorized handler when the handler is compiled into the driver, the user requests its use via `enable-avx2-rx=1`, CPU supports AVX2, and scatter Rx is not used. To verify that the vectorized handler is selected, enable debug logging (`--log-level=pmd,debug`) and check the following message.

```
enic_use_vector_rx_handler use the non-scatter avx2 Rx handler
```

18.8 Limitations

• VLAN 0 Priority Tagging

If a vNIC is configured in TRUNK mode by the UCS manager, the adapter will priority tag egress packets according to 802.1Q if they were not already VLAN tagged by software. If the adapter is connected to a properly configured switch, there will be no unexpected behavior.

In test setups where an Ethernet port of a Cisco adapter in TRUNK mode is connected point-to-point to another adapter port or connected through a router instead of a switch, all ingress packets will be VLAN tagged. Programs such as l3fwd may not account for VLAN tags in packets and may misbehave. One solution is to enable VLAN stripping on ingress so the VLAN tag is removed from the packet and put into the mbuf->vlan_tci field. Here is an example of how to accomplish this:

```
vlan_offload = rte_eth_dev_get_vlan_offload(port);
vlan_offload |= ETH_VLAN_STRIP_OFFLOAD;
rte_eth_dev_set_vlan_offload(port, vlan_offload);
```

Another alternative is modify the adapter's ingress VLAN rewrite mode so that packets with the default VLAN tag are stripped by the adapter and presented to DPDK as untagged packets. In this case mbuf->vlan_tci and the PKT_RX_VLAN and PKT_RX_VLAN_STRIPPED mbuf flags would not be set. This mode is enabled with the devargs parameter ig-vlan-rewrite=untag. For example:

```
-w 12:00.0,ig-vlan-rewrite=untag
```

• SR-IOV

- KVM hypervisor support only. VMware has not been tested.
- Requires VM-FEX, and so is only available on UCS managed servers connected to Fabric Interconnects. It is not on standalone C-Series servers.
- VF devices are not usable directly from the host. They can only be used as assigned devices on VM instances.
- Currently, unbind of the ENIC kernel mode driver 'enic.ko' on the VM instance may hang. As a workaround, enic.ko should be blacklisted or removed from the boot process.
- pci_generic cannot be used as the uio module in the VM. igb_uio or vfio in non-IOMMU mode can be used.
- The number of RQs in UCSM dynamic vNIC configurations must be at least 2.
- The number of SR-IOV devices is limited to 256. Components on target system might limit this number to fewer than 256.

• Flow API

- The number of filters that can be specified with the Generic Flow API is dependent on how many header fields are being masked. Use 'flow create' in a loop to determine how many filters your VIC will support (not more than 1000 for 1300 series VICs). Filters are checked for matching in the order they were added. Since there currently is no grouping or priority support, 'catch-all' filters should be added last.
- The supported range of IDs for the 'MARK' action is 0 - 0xFFFFD.
- RSS and PASSTHRU actions only support "receive normally". They are limited to supporting MARK + RSS and PASSTHRU + MARK to allow the application to mark packets and then receive them normally. These require 1400 series VIC adapters and latest firmware.

- RAW items are limited to matching UDP tunnel headers like VXLAN.
- **Statistics**
 - `rx_good_bytes` (ibytes) always includes VLAN header (4B) and CRC bytes (4B). This behavior applies to 1300 and older series VIC adapters. 1400 series VICs do not count CRC bytes, and count VLAN header only when VLAN stripping is disabled.
 - When the NIC drops a packet because the Rx queue has no free buffers, `rx_good_bytes` still increments by 4B if the packet is not VLAN tagged or VLAN stripping is disabled, or by 8B if the packet is VLAN tagged and stripping is enabled. This behavior applies to 1300 and older series VIC adapters. 1400 series VICs do not increment this byte counter when packets are dropped.
- **RSS Hashing**
 - Hardware enables and disables UDP and TCP RSS hashing together. The driver cannot control UDP and TCP hashing individually.

18.9 How to build the suite

The build instructions for the DPDK suite should be followed. By default the ENIC PMD library will be built into the DPDK library.

Refer to the document *compiling and testing a PMD for a NIC* for details.

For configuring and using UIO and VFIO frameworks, please refer to the documentation that comes with DPDK suite.

18.10 Supported Cisco VIC adapters

ENIC PMD supports all recent generations of Cisco VIC adapters including:

- VIC 1200 series
- VIC 1300 series
- VIC 1400 series

18.11 Supported Operating Systems

Any Linux distribution fulfilling the conditions described in Dependencies section of DPDK documentation.

18.12 Supported features

- Unicast, multicast and broadcast transmission and reception
- Receive queue polling
- Port Hardware Statistics

- Hardware VLAN acceleration
- IP checksum offload
- Receive side VLAN stripping
- Multiple receive and transmit queues
- Promiscuous mode
- Setting RX VLAN (supported via UCSM/CIMC only)
- VLAN filtering (supported via UCSM/CIMC only)
- Execution of application by unprivileged system users
- IPV4, IPV6 and TCP RSS hashing
- UDP RSS hashing (1400 series and later adapters)
- Scattered Rx
- MTU update
- SR-IOV on UCS managed servers connected to Fabric Interconnects
- Flow API
- Overlay offload
 - Rx/Tx checksum offloads for VXLAN, NVGRE, GENEVE
 - TSO for VXLAN and GENEVE packets
 - Inner RSS

18.13 Known bugs and unsupported features in this release

- Signature or flex byte based flow direction
- Drop feature of flow direction
- VLAN based flow direction
- Non-IPV4 flow direction
- Setting of extended VLAN
- MTU update only works if Scattered Rx mode is disabled
- Maximum receive packet length is ignored if Scattered Rx mode is used

18.14 Prerequisites

- Prepare the system as recommended by DPDK suite. This includes environment variables, hugepages configuration, tool-chains and configuration.
- Insert vfio-pci kernel module using the command ‘modprobe vfio-pci’ if the user wants to use VFIO framework.

- Insert uio kernel module using the command ‘modprobe uio’ if the user wants to use UIO framework.
- DPDK suite should be configured based on the user’s decision to use VFIO or UIO framework.
- If the vNIC device(s) to be used is bound to the kernel mode Ethernet driver use ‘ip’ to bring the interface down. The dpdk-devbind.py tool can then be used to unbind the device’s bus id from the ENIC kernel mode driver.
- Bind the intended vNIC to vfio-pci in case the user wants ENIC PMD to use VFIO framework using dpdk-devbind.py.
- Bind the intended vNIC to igb_uio in case the user wants ENIC PMD to use UIO framework using dpdk-devbind.py.

At this point the system should be ready to run DPDK applications. Once the application runs to completion, the vNIC can be detached from vfio-pci or igb_uio if necessary.

Root privilege is required to bind and unbind vNICs to/from VFIO/UIO. VFIO framework helps an unprivileged user to run the applications. For an unprivileged user to run the applications on DPDK and ENIC PMD, it may be necessary to increase the maximum locked memory of the user. The following command could be used to do this.

```
sudo sh -c "ulimit -l <value in Kilo Bytes>"
```

The value depends on the memory configuration of the application, DPDK and PMD. Typically, the limit has to be raised to higher than 2GB. e.g., 2621440

The compilation of any unused drivers can be disabled using the configuration file in config/ directory (e.g., config/common_linux). This would help in bringing down the time taken for building the libraries and the initialization time of the application.

18.15 Additional Reference

- <https://www.cisco.com/c/en/us/products/servers-unified-computing/index.html>
- <https://www.cisco.com/c/en/us/products/interfaces-modules/unified-computing-system-adapters/index.html>

18.16 Contact Information

Any questions or bugs should be reported to DPDK community and to the ENIC PMD maintainers:

- John Daley <johndale@cisco.com>
- Hyong Youb Kim <hyonkim@cisco.com>

FM10K POLL MODE DRIVER

The FM10K poll mode driver library provides support for the Intel FM10000 (FM10K) family of 40GbE/100GbE adapters.

19.1 FTAG Based Forwarding of FM10K

FTAG Based Forwarding is a unique feature of FM10K. The FM10K family of NICs support the addition of a Fabric Tag (FTAG) to carry special information. The FTAG is placed at the beginning of the frame, it contains information such as where the packet comes from and goes, and the vlan tag. In FTAG based forwarding mode, the switch logic forwards packets according to glort (global resource tag) information, rather than the mac and vlan table. Currently this feature works only on PF.

To enable this feature, the user should pass a devargs parameter to the eal like “-w 84:00.0,enable_ftag=1”, and the application should make sure an appropriate FTAG is inserted for every frame on TX side.

19.2 Vector PMD for FM10K

Vector PMD (vPMD) uses Intel® SIMD instructions to optimize packet I/O. It improves load/store bandwidth efficiency of L1 data cache by using a wider SSE/AVX ‘register (1)’. The wider register gives space to hold multiple packet buffers so as to save on the number of instructions when bulk processing packets.

There is no change to the PMD API. The RX/TX handlers are the only two entries for vPMD packet I/O. They are transparently registered at runtime RX/TX execution if all required conditions are met.

1. To date, only an SSE version of FM10K vPMD is available. To ensure that vPMD is in the binary code, set `CONFIG_RTE_LIBRTE_FM10K_INC_VECTOR=y` in the configure file.

Some constraints apply as pre-conditions for specific optimizations on bulk packet transfers. The following sections explain RX and TX constraints in the vPMD.

19.2.1 RX Constraints

Prerequisites and Pre-conditions

For Vector RX it is assumed that the number of descriptor rings will be a power of 2. With this pre-condition, the ring pointer can easily scroll back to the head after hitting the tail without a conditional check. In addition Vector RX can use this assumption to do a bit mask using `ring_size - 1`.

Features not Supported by Vector RX PMD

Some features are not supported when trying to increase the throughput in vPMD. They are:

- IEEE1588
- Flow director
- Header split
- RX checksum offload

Other features are supported using optional MACRO configuration. They include:

- HW VLAN strip
- L3/L4 packet type

To enable via `RX_OLFLAGS` use `RTE_LIBRTE_FM10K_RX_OLFLAGS_ENABLE=y`.

To guarantee the constraint, the following capabilities in `dev_conf.rxmode.offloads` will be checked:

- `DEV_RX_OFFLOAD_VLAN_EXTEND`
- `DEV_RX_OFFLOAD_CHECKSUM`
- `DEV_RX_OFFLOAD_HEADER_SPLIT`
- `fdir_conf->mode`

RX Burst Size

As vPMD is focused on high throughput, it processes 4 packets at a time. So it assumes that the RX burst should be greater than 4 packets per burst. It returns zero if using `nb_pkt < 4` in the receive handler. If `nb_pkt` is not a multiple of 4, a floor alignment will be applied.

19.2.2 TX Constraint

Features not Supported by TX Vector PMD

TX vPMD only works when offloads is set to 0

This means that it does not support any TX offload.

19.3 Limitations

19.3.1 Switch manager

The Intel FM10000 family of NICs integrate a hardware switch and multiple host interfaces. The FM10000 PMD driver only manages host interfaces. For the switch component another switch driver has to be loaded prior to the FM10000 PMD driver. The switch driver can be acquired from Intel support. Only Testpoint is validated with DPDK, the latest version that has been validated with DPDK is 4.1.6.

19.3.2 Support for Switch Restart

For FM10000 multi host based design a DPDK app running in the VM or host needs to be aware of the switch's state since it may undergo a quit-restart. When the switch goes down the DPDK app will receive a LSC event indicating link status down, and the app should stop the worker threads that are polling on the Rx/Tx queues. When switch comes up, a LSC event indicating `LINK_UP` is sent to the app, which can then restart the FM10000 port to resume network processing.

19.3.3 CRC stripping

The FM10000 family of NICs strip the CRC for every packets coming into the host interface. So, keeping CRC is not supported.

19.3.4 Maximum packet length

The FM10000 family of NICS support a maximum of a 15K jumbo frame. The value is fixed and cannot be changed. So, even when the `rxmode.max_rx_pkt_len` member of `struct rte_eth_conf` is set to a value lower than 15364, frames up to 15364 bytes can still reach the host interface.

19.3.5 Statistic Polling Frequency

The FM10000 NICs expose a set of statistics via the PCI BARs. These statistics are read from the hardware registers when `rte_eth_stats_get()` or `rte_eth_xstats_get()` is called. The packet counting registers are 32 bits while the byte counting registers are 48 bits. As a result, the statistics must be polled regularly in order to ensure the consistency of the returned reads.

Given the PCIe Gen3 x8, about 50Gbps of traffic can occur. With 64 byte packets this gives almost 100 million packets/second, causing 32 bit integer overflow after approx 40 seconds. To ensure these overflows are detected and accounted for in the statistics, it is necessary to read statistic regularly. It is suggested to read stats every 20 seconds, which will ensure the statistics are accurate.

19.3.6 Interrupt mode

The FM10000 family of NICS need one separate interrupt for mailbox. So only drivers which support multiple interrupt vectors e.g. `vfiopci` can work for fm10k interrupt mode.

HINIC POLL MODE DRIVER

The hinic PMD (librte_pmd_hinic) provides poll mode driver support for 25Gbps Huawei Intelligent PCIE Network Adapters based on the Huawei Ethernet Controller Hi1822.

20.1 Features

- Multi arch support: x86_64, ARMv8.
- Multiple queues for TX and RX
- Receiver Side Scaling (RSS)
- MAC/VLAN filtering
- Checksum offload
- TSO offload
- Promiscuous mode
- Port hardware statistics
- Link state information
- Link flow control
- Scattered and gather for TX and RX
- SR-IOV - Partially supported at this point, VFIO only
- VLAN filter and VLAN offload
- Allmulticast mode
- MTU update
- Unicast MAC filter
- Multicast MAC filter
- Flow API
- Set Link down or up
- FW version
- LRO

20.2 Prerequisites

- Learning about Huawei Hi1822 IN200 Series Intelligent NICs using <https://e.huawei.com/en/products/cloud-computing-dc/servers/pcie-ssd/in-card>.
- Getting the latest product documents and software supports using <https://support.huawei.com/enterprise/en/intelligent-accelerator-components/in500-solution-pid-23507369>.
- Follow the DPDK Getting Started Guide for Linux to setup the basic DPDK environment.

20.3 Pre-Installation Configuration

20.3.1 Config File Options

The following options can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_HINIC_PMD` (default `y`)

20.4 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

20.5 Limitations or Known issues

Build with ICC is not supported yet. X86-32, Power8, ARMv7 and BSD are not supported yet.

HNS3 POLL MODE DRIVER

The hns3 PMD (librte_pmd_hns3) provides poll mode driver support for the inbuilt Hisilicon Network Subsystem(HNS) network engine found in the Hisilicon Kunpeng 920 SoC.

21.1 Features

Features of the HNS3 PMD are:

- Multiple queues for TX and RX
- Receive Side Scaling (RSS)
- Packet type information
- Checksum offload
- Promiscuous mode
- Multicast mode
- Port hardware statistics
- Jumbo frames
- Link state information
- Interrupt mode for RX
- VLAN stripping
- NUMA support

21.2 Prerequisites

- Get the information about Kunpeng920 chip using <http://www.hisilicon.com/en/Products/ProductList/Kunpeng>.
- Follow the DPDK Getting Started Guide for Linux to setup the basic DPDK environment.

21.3 Pre-Installation Configuration

21.3.1 Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_HNS3_PMD` (default `y`)

21.4 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

21.5 Limitations or Known issues

Currently, we only support VF device is bound to `vfio_pci` or `igb_uio` and then driven by DPDK driver when PF is driven by kernel mode `hns3` `ethdev` driver, VF is not supported when PF is driven by DPDK driver.

Build with ICC is not supported yet. X86-32, Power8, ARMv7 and BSD are not supported yet.

I40E POLL MODE DRIVER

The i40e PMD (librte_pmd_i40e) provides poll mode driver support for 10/25/40 Gbps Intel® Ethernet 700 Series Network Adapters based on the Intel Ethernet Controller X710/XL710/XXV710 and Intel Ethernet Connection X722 (only support part of features).

22.1 Features

Features of the i40e PMD are:

- Multiple queues for TX and RX
- Receiver Side Scaling (RSS)
- MAC/VLAN filtering
- Packet type information
- Flow director
- Cloud filter
- Checksum offload
- VLAN/QinQ stripping and inserting
- TSO offload
- Promiscuous mode
- Multicast mode
- Port hardware statistics
- Jumbo frames
- Link state information
- Link flow control
- Mirror on port, VLAN and VSI
- Interrupt mode for RX
- Scattered and gather for TX and RX
- Vector Poll mode driver
- DCB

- VMDQ
- SR-IOV VF
- Hot plug
- IEEE1588/802.1AS timestamping
- VF Daemon (VFD) - EXPERIMENTAL
- Dynamic Device Personalization (DDP)
- Queue region configuration
- Virtual Function Port Representors

22.2 Prerequisites

- Identifying your adapter using [Intel Support](#) and get the latest NVM/FW images.
- Follow the DPDK Getting Started Guide for Linux to setup the basic DPDK environment.
- To get better performance on Intel platforms, please follow the “How to get best performance with NICs on Intel platforms” section of the Getting Started Guide for Linux.
- Upgrade the NVM/FW version following the [Intel® Ethernet NVM Update Tool Quick Usage Guide for Linux](#) and [Intel® Ethernet NVM Update Tool: Quick Usage Guide for EFI](#) if needed.

22.3 Recommended Matching List

It is highly recommended to upgrade the i40e kernel driver and firmware to avoid the compatibility issues with i40e PMD. Here is the suggested matching list which has been tested and verified. The detailed information can refer to chapter Tested Platforms/Tested NICs in release notes.

DPDK version	Kernel driver version	Firmware version
19.11	2.9.21	7.00
19.08	2.8.43	7.00
19.05	2.7.29	6.80
19.02	2.7.26	6.80
18.11	2.4.6	6.01
18.08	2.4.6	6.01
18.05	2.4.6	6.01
18.02	2.4.3	6.01
17.11	2.1.26	6.01
17.08	2.0.19	6.01
17.05	1.5.23	5.05
17.02	1.5.23	5.05
16.11	1.5.23	5.05
16.07	1.4.25	5.04
16.04	1.4.25	5.02

22.4 Pre-Installation Configuration

22.4.1 Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_I40E_PMD` (default `y`)
Toggle compilation of the `librte_pmd_i40e` driver.
- `CONFIG_RTE_LIBRTE_I40E_DEBUG_*` (default `n`)
Toggle display of generic debugging messages.
- `CONFIG_RTE_LIBRTE_I40E_RX_ALLOW_BULK_ALLOC` (default `y`)
Toggle bulk allocation for RX.
- `CONFIG_RTE_LIBRTE_I40E_INC_VECTOR` (default `n`)
Toggle the use of Vector PMD instead of normal RX/TX path. To enable vPMD for RX, bulk allocation for Rx must be allowed.
- `CONFIG_RTE_LIBRTE_I40E_16BYTE_RX_DESC` (default `n`)
Toggle to use a 16-byte RX descriptor, by default the RX descriptor is 32 byte.
- `CONFIG_RTE_LIBRTE_I40E_QUEUE_NUM_PER_PF` (default 64)
Number of queues reserved for PF.
- `CONFIG_RTE_LIBRTE_I40E_QUEUE_NUM_PER_VM` (default 4)
Number of queues reserved for each VMDQ Pool.

22.4.2 Runtime Config Options

- Reserved number of Queues per VF (default 4)
The number of reserved queue per VF is determined by its host PF. If the PCI address of an i40e PF is `aaaa:bb.cc`, the number of reserved queues per VF can be configured with EAL parameter like `-w aaaa:bb.cc,queue-num-per-vf=n`. The value `n` can be 1, 2, 4, 8 or 16. If no such parameter is configured, the number of reserved queues per VF is 4 by default. If VF request more than reserved queues per VF, PF will able to allocate max to 16 queues after a VF reset.
- Support multiple driver (default disable)
There was a multiple driver support issue during use of 700 series Ethernet Adapter with both Linux kernel and DPDK PMD. To fix this issue, `devargs` parameter `support-multi-driver` is introduced, for example:

```
-w 84:00.0,support-multi-driver=1
```


With the above configuration, DPDK PMD will not change global registers, and will switch PF interrupt from `IntN` to `Int0` to avoid interrupt conflict between DPDK and Linux Kernel.
- Support VF Port Representor (default not enabled)
The i40e PF PMD supports the creation of VF port representors for the control and monitoring of i40e virtual function devices. Each port representor corresponds to a single virtual function of that

device. Using the `devargs` option `representor` the user can specify which virtual functions to create port representors for on initialization of the PF PMD by passing the VF IDs of the VFs which are required.:

```
-w DBDF,representor=[0,1,4]
```

Currently hot-plugging of `representor` ports is not supported so all required `representors` must be specified on the creation of the PF.

- Use `latest` supported vector (default `disable`)

Latest supported vector path may not always get the best perf so vector path was recommended to use only on later platform. But users may want the latest vector path since it can get better perf in some real work loading cases. So `devargs` param `use-latest-supported-vec` is introduced, for example:

```
-w 84:00.0,use-latest-supported-vec=1
```

- Enable validation for VF message (default `not enabled`)

The PF counts messages from each VF. If in any period of seconds the message statistic from a VF exceeds maximal limitation, the PF will ignore any new message from that VF for some seconds. Format – “`maximal-message@period-seconds:ignore-seconds`” For example:

```
-w 84:00.0,vf_msg_cfg=80@120:180
```

22.4.3 Vector RX Pre-conditions

For Vector RX it is assumed that the number of descriptor rings will be a power of 2. With this pre-condition, the ring pointer can easily scroll back to the head after hitting the tail without a conditional check. In addition Vector RX can use this assumption to do a bit mask using `ring_size -1`.

22.5 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

22.6 SR-IOV: Prerequisites and sample Application Notes

1. Load the kernel module:

```
modprobe i40e
```

Check the output in `dmesg`:

```
i40e 0000:83:00.1 ens802f0: renamed from eth0
```

2. Bring up the PF ports:

```
ifconfig ens802f0 up
```

3. Create VF device(s):

Echo the number of VFs to be created into the `sriov_numvfs` sysfs entry of the parent PF.

Example:

```
echo 2 > /sys/devices/pci0000:00/0000:00:03.0/0000:81:00.0/sriov_numvfs
```


4. Assign VF MAC address:

Assign MAC address to the VF using `iproute2` utility. The syntax is:

```
ip link set <PF netdev id> vf <VF id> mac <macaddr>
```

Example:

```
ip link set ens802f0 vf 0 mac a0:b0:c0:d0:e0:f0
```

5. Assign VF to VM, and bring up the VM. Please see the documentation for the *I40E/IXGBE/IGB Virtual Function Driver*.

6. Running testpmd:

Follow instructions available in the document *compiling and testing a PMD for a NIC* to run testpmd.

Example output:

```
...
EAL: PCI device 0000:83:00.0 on NUMA socket 1
EAL: probe driver: 8086:1572 rte_i40e_pmd
EAL: PCI memory mapped at 0x7f7f80000000
EAL: PCI memory mapped at 0x7f7f80800000
PMD: eth_i40e_dev_init(): FW 5.0 API 1.5 NVM 05.00.02 eetrack 8000208a
Interactive-mode selected
Configuring Port 0 (socket 0)
...

PMD: i40e_dev_rx_queue_setup(): Rx Burst Bulk Alloc Preconditions are
satisfied.Rx Burst Bulk Alloc function will be used on port=0, queue=0.

...
Port 0: 68:05:CA:26:85:84
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Done

testpmd>
```

22.7 Sample Application Notes

22.7.1 Vlan filter

Vlan filter only works when Promiscuous mode is off.

To start testpmd, and add vlan 10 to port 0:

```
./app/testpmd -l 0-15 -n 4 -- -i --forward-mode=mac
...

testpmd> set promisc 0 off
testpmd> rx_vlan add 10 0
```

22.7.2 Flow Director

The Flow Director works in receive mode to identify specific flows or sets of flows and route them to specific queues. The Flow Director filters can match the different fields for different type of packet: flow type, specific input set per flow type and the flexible payload.

The default input set of each flow type is:

```

ipv4-other : src_ip_address, dst_ip_address
ipv4-frag  : src_ip_address, dst_ip_address
ipv4-tcp   : src_ip_address, dst_ip_address, src_port, dst_port
ipv4-udp   : src_ip_address, dst_ip_address, src_port, dst_port
ipv4-sctp  : src_ip_address, dst_ip_address, src_port, dst_port,
             verification_tag
ipv6-other : src_ip_address, dst_ip_address
ipv6-frag  : src_ip_address, dst_ip_address
ipv6-tcp   : src_ip_address, dst_ip_address, src_port, dst_port
ipv6-udp   : src_ip_address, dst_ip_address, src_port, dst_port
ipv6-sctp  : src_ip_address, dst_ip_address, src_port, dst_port,
             verification_tag
l2_payload : ether_type

```

The flex payload is selected from offset 0 to 15 of packet's payload by default, while it is masked out from matching.

Start testpmd with `--disable-rss` and `--pkt-filter-mode=perfect`:

```

./app/testpmd -l 0-15 -n 4 -- -i --disable-rss --pkt-filter-mode=perfect \
--rxq=8 --txq=8 --nb-cores=8 --nb-ports=1

```

Add a rule to direct ipv4-udp packet whose `dst_ip=2.2.2.5`, `src_ip=2.2.2.3`, `src_port=32`, `dst_port` to queue 1:

```

testpmd> flow_director_filter 0 mode IP add flow ipv4-udp \
src 2.2.2.3 32 dst 2.2.2.5 32 vlan 0 flexbytes () \
fwd pf queue 1 fd_id 1

```

Check the flow director status:

```

testpmd> show port fdir 0

```

```

##### FDIR infos for port 0 #####
MODE:    PERFECT
SUPPORTED FLOW TYPE:  ipv4-frag ipv4-tcp ipv4-udp ipv4-sctp ipv4-other
                      ipv6-frag ipv6-tcp ipv6-udp ipv6-sctp ipv6-other
                      l2_payload

FLEX PAYLOAD INFO:
max_len:      16          payload_limit: 480
payload_unit: 2          payload_seg:   3
bitmask_unit: 2          bitmask_num:   2
MASK:
  vlan_tci: 0x0000,
  src_ipv4: 0x00000000,
  dst_ipv4: 0x00000000,
  src_port: 0x0000,
  dst_port: 0x0000
  src_ipv6: 0x00000000,0x00000000,0x00000000,0x00000000,
  dst_ipv6: 0x00000000,0x00000000,0x00000000,0x00000000
FLEX PAYLOAD SRC OFFSET:
L2_PAYLOAD:  0      1      2      3      4      5      6      ...
L3_PAYLOAD:  0      1      2      3      4      5      6      ...
L4_PAYLOAD:  0      1      2      3      4      5      6      ...
FLEX MASK CFG:
ipv4-udp:    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ipv4-tcp:    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ipv4-sctp:    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ipv4-other:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ipv4-frag:    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ipv6-udp:    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ipv6-tcp:    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ipv6-sctp:    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

```

    ipv6-other:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    ipv6-frag:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    l2_payload:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    guarant_count: 1          best_count: 0
    guarant_space: 512        best_space: 7168
    collision:    0           free: 0
    maxhash:      0           maxlen: 0
    add:          0           remove: 0
    f_add:        0           f_remove: 0

```

Delete all flow director rules on a port:

```
testpmd> flush_flow_director 0
```

22.7.3 Floating VEB

The Intel® Ethernet 700 Series support a feature called “Floating VEB”.

A Virtual Ethernet Bridge (VEB) is an IEEE Edge Virtual Bridging (EVB) term for functionality that allows local switching between virtual endpoints within a physical endpoint and also with an external bridge/network.

A “Floating” VEB doesn’t have an uplink connection to the outside world so all switching is done internally and remains within the host. As such, this feature provides security benefits.

In addition, a Floating VEB overcomes a limitation of normal VEBs where they cannot forward packets when the physical link is down. Floating VEBs don’t need to connect to the NIC port so they can still forward traffic from VF to VF even when the physical link is down.

Therefore, with this feature enabled VFs can be limited to communicating with each other but not an outside network, and they can do so even when there is no physical uplink on the associated NIC port.

To enable this feature, the user should pass a `devargs` parameter to the EAL, for example:

```
-w 84:00.0,enable_floating_veb=1
```

In this configuration the PMD will use the floating VEB feature for all the VFs created by this PF device.

Alternatively, the user can specify which VFs need to connect to this floating VEB using the `floating_veb_list` argument:

```
-w 84:00.0,enable_floating_veb=1,floating_veb_list=1;3-4
```

In this example VF1, VF3 and VF4 connect to the floating VEB, while other VFs connect to the normal VEB.

The current implementation only supports one floating VEB and one regular VEB. VFs can connect to a floating VEB or a regular VEB according to the configuration passed on the EAL command line.

The floating VEB functionality requires a NIC firmware version of 5.0 or greater.

22.7.4 Dynamic Device Personalization (DDP)

The Intel® Ethernet 700 Series except for the Intel Ethernet Connection X722 support a feature called “Dynamic Device Personalization (DDP)”, which is used to configure hardware by downloading a profile to support protocols/filters which are not supported by default. The DDP functionality requires a NIC firmware version of 6.0 or greater.

Current implementation supports GTP-C/GTP-U/PPPoE/PPPoL2TP, steering can be used with `rte_flow` API.

GTPv1 package is released, and it can be downloaded from <https://downloadcenter.intel.com/download/27587>.

PPPoE package is released, and it can be downloaded from <https://downloadcenter.intel.com/download/28040>.

Load a profile which supports GTP and store backup profile:

```
testpmd> ddp add 0 ./gtp.pkggo,./backup.pkggo
```

Delete a GTP profile and restore backup profile:

```
testpmd> ddp del 0 ./backup.pkggo
```

Get loaded DDP package info list:

```
testpmd> ddp get list 0
```

Display information about a GTP profile:

```
testpmd> ddp get info ./gtp.pkggo
```

22.7.5 Input set configuration

Input set for any PCTYPE can be configured with user defined configuration, For example, to use only 48bit prefix for IPv6 src address for IPv6 TCP RSS:

```
testpmd> port config 0 pctype 43 hash_inset clear all
testpmd> port config 0 pctype 43 hash_inset set field 13
testpmd> port config 0 pctype 43 hash_inset set field 14
testpmd> port config 0 pctype 43 hash_inset set field 15
```

22.7.6 Queue region configuration

The Intel® Ethernet 700 Series supports a feature of queue regions configuration for RSS in the PF, so that different traffic classes or different packet classification types can be separated to different queues in different queue regions. There is an API for configuration of queue regions in RSS with a command line. It can parse the parameters of the region index, queue number, queue start index, user priority, traffic classes and so on. Depending on commands from the command line, it will call i40e private APIs and start the process of setting or flushing the queue region configuration. As this feature is specific for i40e only private APIs are used. These new `test_pmd` commands are as shown below. For details please refer to `../testpmd_app Ug/index`.

```
testpmd> set port (port_id) queue-region region_id (value) \
        queue_start_index (value) queue_num (value)
testpmd> set port (port_id) queue-region region_id (value) flowtype (value)
testpmd> set port (port_id) queue-region UP (value) region_id (value)
testpmd> set port (port_id) queue-region flush (on|off)
testpmd> show port (port_id) queue-region
```

22.8 Limitations or Known issues

22.8.1 MPLS packet classification

For firmware versions prior to 5.0, MPLS packets are not recognized by the NIC. The L2 Payload flow type in flow director can be used to classify MPLS packet by using a command in `testpmd` like:

```
testpmd> flow_director_filter 0 mode IP add flow l2_payload ether 0x8847 flexbytes
() fwd pf queue <N> fd_id <M>
```

With the NIC firmware version 5.0 or greater, some limited MPLS support is added: Native MPLS (MPLS in Ethernet) skip is implemented, while no new packet type, no classification or offload are possible. With this change, L2 Payload flow type in flow director cannot be used to classify MPLS packet as with previous firmware versions. Meanwhile, the Ethertype filter can be used to classify MPLS packet by using a command in testpmd like:

```
testpmd> ethertype_filter 0 add mac_ignr 00:00:00:00:00:00 ethertype 0x8847 fwd
queue <M>
```

22.8.2 16 Byte RX Descriptor setting on DPDK VF

Currently the VF's RX descriptor mode is decided by PF. There's no PF-VF interface for VF to request the RX descriptor mode, also no interface to notify VF its own RX descriptor mode. For all available versions of the i40e driver, these drivers don't support 16 byte RX descriptor. If the Linux i40e kernel driver is used as host driver, while DPDK i40e PMD is used as the VF driver, DPDK cannot choose 16 byte receive descriptor. The reason is that the RX descriptor is already set to 32 byte by the i40e kernel driver. That is to say, user should keep `CONFIG_RTE_LIBRTE_I40E_16BYTE_RX_DESC=n` in config file. In the future, if the Linux i40e driver supports 16 byte RX descriptor, user should make sure the DPDK VF uses the same RX descriptor mode, 16 byte or 32 byte, as the PF driver.

The same rule for DPDK PF + DPDK VF. The PF and VF should use the same RX descriptor mode. Or the VF RX will not work.

22.8.3 Receive packets with Ethertype 0x88A8

Due to the FW limitation, PF can receive packets with Ethertype 0x88A8 only when floating VEB is disabled.

22.8.4 Incorrect Rx statistics when packet is oversize

When a packet is over maximum frame size, the packet is dropped. However, the Rx statistics, when calling `rte_eth_stats_get` incorrectly shows it as received.

22.8.5 VF & TC max bandwidth setting

The per VF max bandwidth and per TC max bandwidth cannot be enabled in parallel. The behavior is different when handling per VF and per TC max bandwidth setting. When enabling per VF max bandwidth, SW will check if per TC max bandwidth is enabled. If so, return failure. When enabling per TC max bandwidth, SW will check if per VF max bandwidth is enabled. If so, disable per VF max bandwidth and continue with per TC max bandwidth setting.

22.8.6 TC TX scheduling mode setting

There are 2 TX scheduling modes for TCs, round robin and strict priority mode. If a TC is set to strict priority mode, it can consume unlimited bandwidth. It means if APP has set the max bandwidth for that TC, it comes to no effect. It's suggested to set the strict priority mode for a TC that is latency sensitive but no consuming much bandwidth.

22.8.7 VF performance is impacted by PCI extended tag setting

To reach maximum NIC performance in the VF the PCI extended tag must be enabled. The DPDK i40e PF driver will set this feature during initialization, but the kernel PF driver does not. So when running traffic on a VF which is managed by the kernel PF driver, a significant NIC performance downgrade has been observed (for 64 byte packets, there is about 25% line-rate downgrade for a 25GbE device and about 35% for a 40GbE device).

For kernel version ≥ 4.11 , the kernel's PCI driver will enable the extended tag if it detects that the device supports it. So by default, this is not an issue. For kernels ≤ 4.11 or when the PCI extended tag is disabled it can be enabled using the steps below.

1. Get the current value of the PCI configure register:

```
setpci -s <XX:XX.X> a8.w
```

2. Set bit 8:

```
value = value | 0x100
```

3. Set the PCI configure register with new value:

```
setpci -s <XX:XX.X> a8.w=<value>
```

22.8.8 Vlan strip of VF

The VF vlan strip function is only supported in the i40e kernel driver $\geq 2.1.26$.

22.8.9 DCB function

DCB works only when RSS is enabled.

22.8.10 Global configuration warning

I40E PMD will set some global registers to enable some function or set some configure. Then when using different ports of the same NIC with Linux kernel and DPDK, the port with Linux kernel will be impacted by the port with DPDK. For example, register I40E_GL_SWT_L2TAGCTRL is used to control L2 tag, i40e PMD uses I40E_GL_SWT_L2TAGCTRL to set vlan TPID. If setting TPID in port A with DPDK, then the configuration will also impact port B in the NIC with kernel driver, which don't want to use the TPID. So PMD reports warning to clarify what is changed by writing global register.

22.9 High Performance of Small Packets on 40GbE NIC

As there might be firmware fixes for performance enhancement in latest version of firmware image, the firmware update might be needed for getting high performance. Check the Intel support website for the latest firmware updates. Users should consult the release notes specific to a DPDK release to identify the validated firmware version for a NIC using the i40e driver.

22.9.1 Use 16 Bytes RX Descriptor Size

As i40e PMD supports both 16 and 32 bytes RX descriptor sizes, and 16 bytes size can provide helps to high performance of small packets. Configuration of `CONFIG RTE_LIBRTE_I40E_16BYTE_RX_DESC` in config files can be changed to use 16 bytes size RX descriptors.

22.9.2 Input set requirement of each pctype for FDIR

Each PCTYPE can only have one specific FDIR input set at one time. For example, if creating 2 `rte_flow` rules with different input set for one PCTYPE, it will fail and return the info “Conflict with the first rule’s input set”, which means the current rule’s input set conflicts with the first rule’s. Remove the first rule if want to change the input set of the PCTYPE.

22.10 Example of getting best performance with l3fwd example

The following is an example of running the DPDK `l3fwd` sample application to get high performance with a server with Intel Xeon processors and Intel Ethernet CNA XL710.

The example scenario is to get best performance with two Intel Ethernet CNA XL710 40GbE ports. See Fig. 22.1 for the performance test setup.

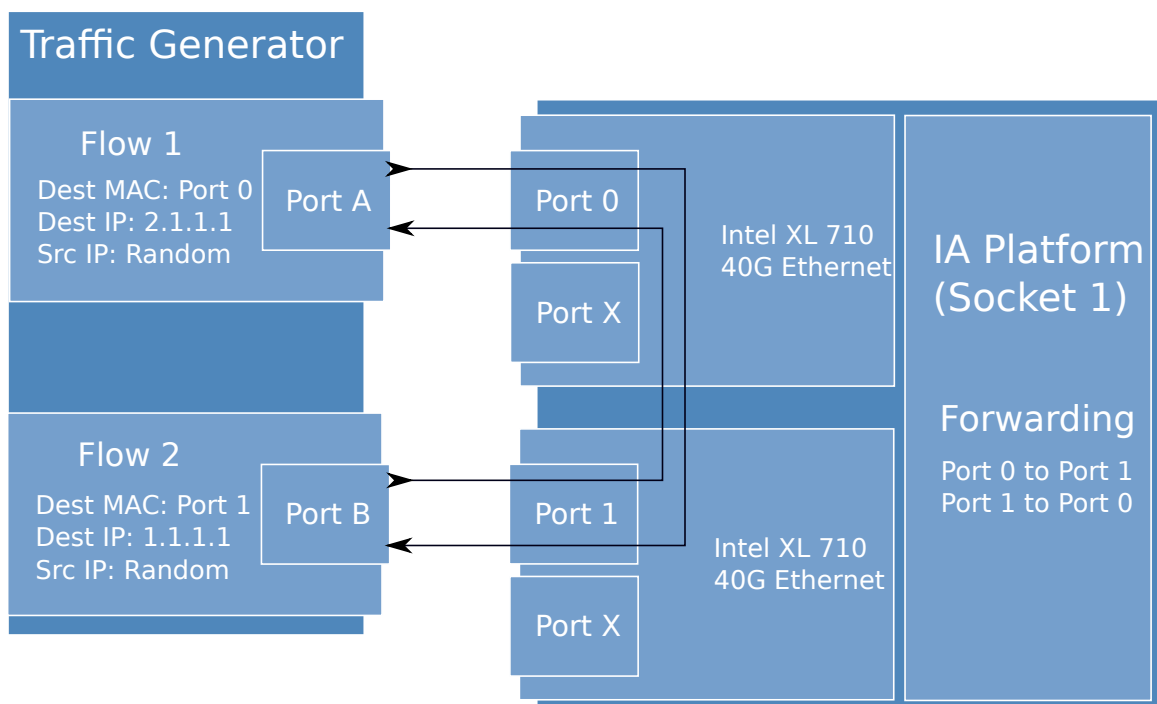


Fig. 22.1: Performance Test Setup

1. Add two Intel Ethernet CNA XL710 to the platform, and use one port per card to get best performance. The reason for using two NICs is to overcome a PCIe v3.0 limitation since it cannot provide 80GbE bandwidth for two 40GbE ports, but two different PCIe v3.0 x8 slot can. Refer to the sample NICs output above, then we can select `82:00.0` and `85:00.0` as test ports:

```
82:00.0 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
85:00.0 Ethernet [0200]: Intel XL710 for 40GbE QSFP+ [8086:1583]
```

2. Connect the ports to the traffic generator. For high speed testing, it's best to use a hardware traffic generator.
3. Check the PCI devices numa node (socket id) and get the cores number on the exact socket id. In this case, 82:00.0 and 85:00.0 are both in socket 1, and the cores on socket 1 in the referenced platform are 18-35 and 54-71. Note: Don't use 2 logical cores on the same core (e.g core18 has 2 logical cores, core18 and core54), instead, use 2 logical cores from different cores (e.g core18 and core19).
4. Bind these two ports to igb_uio.
5. As to Intel Ethernet CNA XL710 40GbE port, we need at least two queue pairs to achieve best performance, then two queues per port will be required, and each queue pair will need a dedicated CPU core for receiving/transmitting packets.
6. The DPDK sample application `l3fwd` will be used for performance testing, with using two ports for bi-directional forwarding. Compile the `l3fwd` sample with the default lpm mode.
7. The command line of running `l3fwd` would be something like the following:

```
./l3fwd -l 18-21 -n 4 -w 82:00.0 -w 85:00.0 \
-- -p 0x3 --config '(0,0,18),(0,1,19),(1,0,20),(1,1,21)'
```

This means that the application uses core 18 for port 0, queue pair 0 forwarding, core 19 for port 0, queue pair 1 forwarding, core 20 for port 1, queue pair 0 forwarding, and core 21 for port 1, queue pair 1 forwarding.

8. Configure the traffic at a traffic generator.
 - Start creating a stream on packet generator.
 - Set the Ethernet II type to 0x0800.

22.10.1 Tx bytes affected by the link status change

For firmware versions prior to 6.01 for X710 series and 3.33 for X722 series, the `tx_bytes` statistics data is affected by the link down event. Each time the link status changes to down, the `tx_bytes` decreases 110 bytes.

ICE POLL MODE DRIVER

The ice PMD (`librte_pmd_ice`) provides poll mode driver support for 10/25 Gbps Intel® Ethernet 810 Series Network Adapters based on the Intel Ethernet Controller E810.

23.1 Prerequisites

- Identifying your adapter using [Intel Support](#) and get the latest NVM/FW images.
- Follow the DPDK Getting Started Guide for Linux to setup the basic DPDK environment.
- To get better performance on Intel platforms, please follow the “How to get best performance with NICs on Intel platforms” section of the Getting Started Guide for Linux.

23.2 Recommended Matching List

It is highly recommended to upgrade the ice kernel driver and firmware and DDP packages to avoid the compatibility issues with ice PMD. Here is the suggested matching list.

DPDK version	Kernel driver version	Firmware version	DDP OS Package	DDP COMMS Package
19.11	0.12.25	1.1.16.39	1.3.4	1.3.10
19.08 (experimental)	0.10.1	1.1.12.7	1.2.0	N/A
19.05 (experimental)	0.9.4	1.1.10.16	1.1.0	N/A

23.3 Pre-Installation Configuration

23.3.1 Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_ICE_PMD` (default `y`)
Toggle compilation of the `librte_pmd_ice` driver.
- `CONFIG_RTE_LIBRTE_ICE_DEBUG_*` (default `n`)
Toggle display of generic debugging messages.

- `CONFIG_RTE_LIBRTE_ICE_16BYTE_RX_DESC` (default `n`)

Toggle to use a 16-byte RX descriptor, by default the RX descriptor is 32 byte.

23.3.2 Runtime Config Options

- Safe Mode Support (default 0)

If driver failed to load OS package, by default driver's initialization failed. But if user intend to use the device without OS package, user can take `devargs` parameter `safe-mode-support`, for example:

```
-w 80:00.0,safe-mode-support=1
```

Then the driver will be initialized successfully and the device will enter Safe Mode. NOTE: In Safe mode, only very limited features are available, features like RSS, checksum, fdir, tunneling ... are all disabled.

- Generic Flow Pipeline Mode Support (default 0)

In pipeline mode, a flow can be set at one specific stage by setting parameter `priority`. Currently, we support two stages: `priority = 0` or `!0`. Flows with `priority 0` located at the first pipeline stage which typically be used as a firewall to drop the packet on a blacklist (we called it permission stage). At this stage, flow rules are created for the device's exact match engine: switch. Flows with `priority !0` located at the second stage, typically packets are classified here and be steered to specific queue or queue group (we called it distribution stage). At this stage, flow rules are created for device's flow director engine. For none-pipeline mode, `priority` is ignored, a flow rule can be created as a flow director rule or a switch rule depends on its pattern/action and the resource allocation situation, all flows are virtually at the same pipeline stage. By default, generic flow API is enabled in none-pipeline mode, user can choose to use pipeline mode by setting `devargs` parameter `pipeline-mode-support`, for example:

```
-w 80:00.0,pipeline-mode-support=1
```

- Flow Mark Support (default 0)

This is a hint to the driver to select the data path that supports flow mark extraction by default. NOTE: This is an experimental devarg, it will be removed when any of below conditions is ready. 1) all data paths support flow mark (currently vPMD does not) 2) a new offload like `RTE_DEV_RX_OFFLOAD_FLOW_MARK` be introduced as a standard way to hint. Example:

```
-w 80:00.0,flow-mark-support=1
```

- Protocol extraction for per queue

Configure the RX queues to do protocol extraction into mbuf for protocol handling acceleration, like checking the TCP SYN packets quickly.

The argument format is:

```
-w 18:00.0,proto_xtr=<queues:protocol>[<queues:protocol>...]
-w 18:00.0,proto_xtr=<protocol>
```

Queues are grouped by (and) within the group. The - character is used as a range separator and , is used as a single number separator. The grouping () can be omitted for single element group. If no queues are specified, PMD will use this protocol extraction type for all queues.

Protocol is : `vlan, ipv4, ipv6, ipv6_flow, tcp`.

```
testpmd -w 18:00.0,proto_xtr='[(1,2-3,8-9):tcp,10-13:vlan]'
```

This setting means queues 1, 2-3, 8-9 are TCP extraction, queues 10-13 are VLAN extraction, other queues run with no protocol extraction.

```
testpmd -w 18:00.0,proto_xtr=vlan,proto_xtr='[(1,2-3,8-9):tcp,10-23:ipv6]'
```

This setting means queues 1, 2-3, 8-9 are TCP extraction, queues 10-23 are IPv6 extraction, other queues use the default VLAN extraction.

The extraction metadata is copied into the registered dynamic mbuf field, and the related dynamic mbuf flags is set.

Table 23.1: Protocol extraction :

vlan

VLAN2			VLAN1		
PCP	D	VID	PCP	D	VID

VLAN1 - single or EVLAN (first for QinQ).

VLAN2 - C-VLAN (second for QinQ).

Table 23.2: Protocol extraction : ipv4

IPHDR2			IPHDR1	
Ver	Hdr Len	ToS	TTL	Protocol

IPHDR1 - IPv4 header word 4, “TTL” and “Protocol” fields.

IPHDR2 - IPv4 header word 0, “Ver”, “Hdr Len” and “Type of Service” fields.

Table 23.3: Protocol extraction : ipv6

IPHDR2			IPHDR1	
Ver	Traffic class	Flow	Next Header	Hop Limit

IPHDR1 - IPv6 header word 3, “Next Header” and “Hop Limit” fields.

IPHDR2 - IPv6 header word 0, “Ver”, “Traffic class” and high 4 bits of “Flow Label” fields.

Table 23.4: Protocol extraction :

ipv6_flow

IPHDR2		IPHDR1	
Ver	Traffic class	Flow Label	

IPHDR1 - IPv6 header word 1, 16 low bits of the “Flow Label” field.

IPHDR2 - IPv6 header word 0, “Ver”, “Traffic class” and high 4 bits of “Flow Label” fields.

Table 23.5: Protocol extraction : tcp

TCPHDR2		TCPHDR1	
Reserved		Offset	Flags

TCPHDR1 - TCP header word 6, “Data Offset” and “Flags” fields.

TCPHDR2 - Reserved

Use `rte_net_ice_dynf_proto_xtr_metadata_get` to access the protocol extraction metadata, and use `RTE_PKT_RX_DYNF_PROTO_XTR_*` to get the metadata type of `struct rte_mbuf::ol_flags`.

The `rte_net_ice_dump_proto_xtr_metadata` routine shows how to access the protocol extraction result in `struct rte_mbuf`.

23.4 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

23.5 Features

23.5.1 Vector PMD

Vector PMD for RX and TX path are selected automatically. The paths are chosen based on 2 conditions.

- **CPU** On the X86 platform, the driver checks if the CPU supports AVX2. If it's supported, AVX2 paths will be chosen. If not, SSE is chosen.
- **Offload features** The supported HW offload features are described in the document `ice_vec.ini`. If any not supported features are used, ICE vector PMD is disabled and the normal paths are chosen.

23.5.2 Malicious driver detection (MDD)

It's not appropriate to send a packet, if this packet's destination MAC address is just this port's MAC address. If SW tries to send such packets, HW will report a MDD event and drop the packets.

The APPs based on DPDK should avoid providing such packets.

23.6 Sample Application Notes

23.6.1 Vlan filter

Vlan filter only works when Promiscuous mode is off.

To start `testpmd`, and add vlan 10 to port 0:

```
./app/testpmd -l 0-15 -n 4 -- -i  
...
```

```
testpmd> rx_vlan add 10 0
```

23.7 Limitations or Known issues

The Intel E810 requires a programmable pipeline package be downloaded by the driver to support normal operations. The E810 has a limited functionality built in to allow PXE boot and other use cases, but the driver must download a package file during the driver initialization stage.

The default DDP package file name is `ice.pkg`. For a specific NIC, the DDP package supposed to be loaded can have a filename: `ice-xxxxxx.pkg`, where 'xxxxxx' is the 64-bit PCIe Device Serial Number of the NIC. For example, if the NIC's device serial number is 00-CC-BB-FF-FF-AA-05-68, the device-specific DDP package filename is `ice-00ccbffffaa0568.pkg` (in hex and all low case). During initialization, the driver searches in the following paths in order: `/lib/firmware/updates/intel/ice/ddp` and `/lib/firmware/intel/ice/ddp`. The corresponding device-specific DDP package will be downloaded first if the file exists. If not, then the driver tries to load the default package. The type of loaded package is stored in `ice_adapter->active_pkg_type`.

A symbolic link to the DDP package file is also ok. The same package file is used by both the kernel driver and the DPDK PMD.

23.7.1 19.02 limitation

Ice code released in 19.02 is for evaluation only.

IFCVF VDPA DRIVER

The IFCVF vDPA (vhost data path acceleration) driver provides support for the Intel FPGA 100G VF (IFCVF). IFCVF's datapath is virtio ring compatible, it works as a HW vhost backend which can send/receive packets to/from virtio directly by DMA. Besides, it supports dirty page logging and device state report/restore, this driver enables its vDPA functionality.

24.1 Pre-Installation Configuration

24.1.1 Config File Options

The following option can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_IFC_PMD` (default `y` for linux)
Toggle compilation of the `librte_pmd_ifc` driver.

24.2 IFCVF vDPA Implementation

IFCVF's vendor ID and device ID are same as that of virtio net pci device, with its specific subsystem vendor ID and device ID. To let the device be probed by IFCVF driver, adding "`vdpa=1`" parameter helps to specify that this device is to be used in vDPA mode, rather than polling mode, virtio pmd will skip when it detects this message. If no this parameter specified, device will not be used as a vDPA device, and it will be driven by virtio pmd.

Different VF devices serve different virtio frontends which are in different VMs, so each VF needs to have its own DMA address translation service. During the driver probe a new container is created for this device, with this container vDPA driver can program DMA remapping table with the VM's memory region information.

The device argument "`sw-live-migration=1`" will configure the driver into SW assisted live migration mode. In this mode, the driver will set up a SW relay thread when LM happens, this thread will help device to log dirty pages. Thus this mode does not require HW to implement a dirty page logging function block, but will consume some percentage of CPU resource depending on the network throughput. If no this parameter specified, driver will rely on device's logging capability.

24.2.1 Key IFCVF vDPA driver ops

- `ifcvf_dev_config`: Enable VF data path with virtio information provided by vhost lib, including IOMMU programming to enable VF DMA to VM's memory, VFIO interrupt setup to route HW

interrupt to virtio driver, create notify relay thread to translate virtio driver's kick to a MMIO write onto HW, HW queues configuration.

This function gets called to set up HW data path backend when virtio driver in VM gets ready.

- `ifcvf_dev_close`: Revoke all the setup in `ifcvf_dev_config`.

This function gets called when virtio driver stops device in VM.

24.2.2 To create a vhost port with IFC VF

- Create a vhost socket and assign a VF's device ID to this socket via vhost API. When QEMU vhost connection gets ready, the assigned VF will get configured automatically.

24.3 Features

Features of the IFCVF driver are:

- Compatibility with virtio 0.95 and 1.0.
- SW assisted vDPA live migration.

24.4 Prerequisites

- Platform with IOMMU feature. IFC VF needs address translation service to Rx/Tx directly with virtio driver in VM.

24.5 Limitations

24.5.1 Dependency on vfio-pci

vDPA driver needs to setup VF MSIX interrupts, each queue's interrupt vector is mapped to a callfd associated with a virtio ring. Currently only vfio-pci allows multiple interrupts, so the IFCVF driver is dependent on vfio-pci.

24.5.2 Live Migration with VIRTIO_NET_F_GUEST_ANNOUNCE

IFC VF doesn't support RARP packet generation, virtio frontend supporting `VIRTIO_NET_F_GUEST_ANNOUNCE` feature can help to do that.

IGB POLL MODE DRIVER

The IGB PMD (`librte_pmd_e1000`) provides poll mode driver support for Intel 1GbE nics.

25.1 Features

Features of the IGB PMD are:

- Multiple queues for TX and RX
- Receiver Side Scaling (RSS)
- MAC/VLAN filtering
- Packet type information
- Double VLAN
- IEEE 1588
- TSO offload
- Checksum offload
- TCP segmentation offload
- Jumbo frames supported

25.2 Limitations or Known issues

25.3 Supported Chipsets and NICs

- Intel 82576EB 10 Gigabit Ethernet Controller
- Intel 82580EB 10 Gigabit Ethernet Controller
- Intel 82580DB 10 Gigabit Ethernet Controller
- Intel Ethernet Controller I210
- Intel Ethernet Controller I350

IPN3KE POLL MODE DRIVER

The ipn3ke PMD (librte_pmd_ipn3ke) provides poll mode driver support for Intel® FPGA PAC(Programmable Acceleration Card) N3000 based on the Intel Ethernet Controller X710/XXV710 and Intel Arria 10 FPGA.

In this card, FPGA is an acceleration bridge between network interface and the Intel Ethernet Controller. Although both FPGA and Ethernet Controllers are connected to CPU with PCIe Gen3x16 Switch, all the packet RX/TX is handled by Intel Ethernet Controller. So from application point of view the data path is still the legacy Intel Ethernet Controller X710/XXV710 PMD. Besides this, users can enable more acceleration features by FPGA IP.

26.1 Prerequisites

- Identifying your adapter using [Intel Support](#) and get the latest NVM/FW images.
- Follow the DPDK Getting Started Guide for Linux to setup the basic DPDK environment.
- To get better performance on Intel platforms, please follow the “How to get best performance with NICs on Intel platforms” section of the Getting Started Guide for Linux.

26.2 Pre-Installation Configuration

26.2.1 Config File Options

The following options can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_IPN3KE_PMD` (default `y`)
Toggle compilation of the `librte_pmd_ipn3ke` driver.

26.2.2 Runtime Config Options

- `AFU name`

AFU name identifies which AFU is used by IPN3KE. The AFU name format is “Port|BDF”, Each FPGA can be divided into four blocks at most. “Port” identifies which FPGA block the AFU bitstream belongs to, but currently only 0 IPN3KE support. “BDF” means FPGA PCIe BDF. For example:

```
--vdev 'ipn3ke_cfg0,afu=0|b3:00.0'
```


IXGBE DRIVER

27.1 Vector PMD for IXGBE

Vector PMD uses Intel® SIMD instructions to optimize packet I/O. It improves load/store bandwidth efficiency of L1 data cache by using a wider SSE/AVX register 1 (1). The wider register gives space to hold multiple packet buffers so as to save instruction number when processing bulk of packets.

There is no change to PMD API. The RX/TX handler are the only two entries for vPMD packet I/O. They are transparently registered at runtime RX/TX execution if all condition checks pass.

1. To date, only an SSE version of IX GBE vPMD is available. To ensure that vPMD is in the binary code, ensure that the option `CONFIG_RTE_IXGBE_INC_VECTOR=y` is in the configure file.

Some constraints apply as pre-conditions for specific optimizations on bulk packet transfers. The following sections explain RX and TX constraints in the vPMD.

27.1.1 RX Constraints

Prerequisites and Pre-conditions

The following prerequisites apply:

- To enable vPMD to work for RX, bulk allocation for Rx must be allowed.

Ensure that the following pre-conditions are satisfied:

- `rxq->rx_free_thresh >= RTE_PMD_IXGBE_RX_MAX_BURST`
- `rxq->rx_free_thresh < rxq->nb_rx_desc`
- `(rxq->nb_rx_desc % rxq->rx_free_thresh) == 0`
- `rxq->nb_rx_desc < (IXGBE_MAX_RING_DESC - RTE_PMD_IXGBE_RX_MAX_BURST)`

These conditions are checked in the code.

Scattered packets are not supported in this mode. If an incoming packet is greater than the maximum acceptable length of one “mbuf” data size (by default, the size is 2 KB), vPMD for RX would be disabled.

By default, `IXGBE_MAX_RING_DESC` is set to 4096 and `RTE_PMD_IXGBE_RX_MAX_BURST` is set to 32.

Feature not Supported by RX Vector PMD

Some features are not supported when trying to increase the throughput in vPMD. They are:

- IEEE1588
- FDIR
- Header split
- RX checksum off load

Other features are supported using optional MACRO configuration. They include:

- HW VLAN strip
- HW extend dual VLAN

To guarantee the constraint, capabilities in `dev_conf.rxmode.offloads` will be checked:

- `DEV_RX_OFFLOAD_VLAN_STRIP`
- `DEV_RX_OFFLOAD_VLAN_EXTEND`
- `DEV_RX_OFFLOAD_CHECKSUM`
- `DEV_RX_OFFLOAD_HEADER_SPLIT`
- `dev_conf`

`fdir_conf->mode` will also be checked.

VF Runtime Options

The following `devargs` options can be enabled at runtime. They must be passed as part of EAL arguments. For example,

```
testpmd -w af:10.0,pflink_fullchk=1 -- -i
```

- `pflink_fullchk` (default 0)

When calling `rte_eth_link_get_nowait()` to get VF link status, this option is used to control how VF synchronizes its status with PF's. If set, VF will not only check the PF's physical link status by reading related register, but also check the mailbox status. We call this behavior as fully checking. And checking mailbox will trigger PF's mailbox interrupt generation. If unset, the application can get the VF's link status quickly by just reading the PF's link status register, this will avoid the whole system's mailbox interrupt generation.

`rte_eth_link_get()` will still use the mailbox method regardless of the `pflink_fullchk` setting.

RX Burst Size

As vPMD is focused on high throughput, it assumes that the RX burst size is equal to or greater than 32 per burst. It returns zero if using `nb_pkt < 32` as the expected packet number in the receive handler.

27.1.2 TX Constraint

Prerequisite

The only prerequisite is related to `tx_rs_thresh`. The `tx_rs_thresh` value must be greater than or equal to `RTE_PMD_IXGBE_TX_MAX_BURST`, but less or equal to `RTE_IXGBE_TX_MAX_FREE_BUF_SZ`. Consequently, by default the `tx_rs_thresh` value is in the range 32 to 64.

Feature not Supported by TX Vector PMD

TX vPMD only works when offloads is set to 0

This means that it does not support any TX offload.

27.2 Application Programming Interface

In DPDK release v16.11 an API for ixgbe specific functions has been added to the ixgbe PMD. The declarations for the API functions are in the header `rte_pmd_ixgbe.h`.

27.3 Sample Application Notes

27.3.1 l3fwd

When running `l3fwd` with vPMD, there is one thing to note. In the configuration, ensure that `DEV_RX_OFFLOAD_CHECKSUM` in `port_conf.rxmode.offloads` is NOT set. Otherwise, by default, RX vPMD is disabled.

27.3.2 load_balancer

As in the case of `l3fwd`, to enable vPMD, do NOT set `DEV_RX_OFFLOAD_CHECKSUM` in `port_conf.rxmode.offloads`. In addition, for improved performance, use `-bsz "(32,32),(64,64),(32,32)"` in `load_balancer` to avoid using the default burst size of 144.

27.4 Limitations or Known issues

27.4.1 Malicious Driver Detection not Supported

The Intel x550 series NICs support a feature called MDD (Malicious Driver Detection) which checks the behavior of the VF driver. If this feature is enabled, the VF must use the advanced context descriptor correctly and set the CC (Check Context) bit. DPDK PF doesn't support MDD, but kernel PF does. We may hit problem in this scenario kernel PF + DPDK VF. If user enables MDD in kernel PF, DPDK VF will not work. Because kernel PF thinks the VF is malicious. But actually it's not. The only reason is the VF doesn't act as MDD required. There's significant performance impact to support MDD. DPDK should check if the advanced context descriptor should be set and set it. And DPDK has to ask the info about the header length from the upper layer, because parsing the packet itself is not acceptable. So, it's

too expensive to support MDD. When using kernel PF + DPDK VF on x550, please make sure to use a kernel PF driver that disables MDD or can disable MDD.

Some kernel drivers already disable MDD by default while some kernels can use the command `insmod ixgbe.ko MDD=0,0` to disable MDD. Each “0” in the command refers to a port. For example, if there are 6 ixgbe ports, the command should be changed to `insmod ixgbe.ko MDD=0,0,0,0,0,0`.

27.4.2 Statistics

The statistics of ixgbe hardware must be polled regularly in order for it to remain consistent. Running a DPDK application without polling the statistics will cause registers on hardware to count to the maximum value, and “stick” at that value.

In order to avoid statistic registers every reaching the maximum value, read the statistics from the hardware using `rte_eth_stats_get()` or `rte_eth_xstats_get()`.

The maximum time between statistics polls that ensures consistent results can be calculated as follows:

```
max_read_interval = UINT_MAX / max_packets_per_second
max_read_interval = 4294967295 / 14880952
max_read_interval = 288.6218096127183 (seconds)
max_read_interval = ~4 mins 48 sec.
```

In order to ensure valid results, it is recommended to poll every 4 minutes.

27.4.3 MTU setting

Although the user can set the MTU separately on PF and VF ports, the ixgbe NIC only supports one global MTU per physical port. So when the user sets different MTUs on PF and VF ports in one physical port, the real MTU for all these PF and VF ports is the largest value set. This behavior is based on the kernel driver behavior.

27.4.4 VF MAC address setting

On ixgbe, the concept of “pool” can be used for different things depending on the mode. In VMDq mode, “pool” means a VMDq pool. In IOV mode, “pool” means a VF.

There is no RTE API to add a VF’s MAC address from the PF. On ixgbe, the `rte_eth_dev_mac_addr_add()` function can be used to add a VF’s MAC address, as a workaround.

27.4.5 X550 does not support legacy interrupt mode

Description

X550 cannot get interrupts if using `uio_pci_generic` module or using legacy interrupt mode of `igb_uio` or `vfio`. Because the errata of X550 states that the Interrupt Status bit is not implemented. The errata is the item #22 from [X550 spec update](#)

Implication

When using `uio_pci_generic` module or using legacy interrupt mode of `igb_uio` or `vfio`, the Interrupt Status bit would be checked if the interrupt is coming. Since the bit is not implemented in X550, the irq cannot be handled correctly and cannot report the event fd to DPDK apps. Then apps cannot get interrupts and `dmesg` will show messages like `irq #No.: `` ``nobody cared`.

Workaround

Do not bind the `uio_pci_generic` module in X550 NICs. Do not bind `igb_uio` with legacy mode in X550 NICs. Before binding `vfio` with legacy mode in X550 NICs, use `modprobe vfio `` ``nointxmask=1` to load `vfio` module if the intx is not shared with other devices.

27.5 Inline crypto processing support

Inline IPsec processing is supported for `RTE_SECURITY_ACTION_TYPE_INLINE_CRYPTO` mode for ESP packets only:

- ESP authentication only: AES-128-GMAC (128-bit key)
- ESP encryption and authentication: AES-128-GCM (128-bit key)

IPsec Security Gateway Sample Application supports inline IPsec processing for ixgbe PMD.

For more details see the IPsec Security Gateway Sample Application and Security library documentation.

27.6 Virtual Function Port Representors

The IXGBE PF PMD supports the creation of VF port representors for the control and monitoring of IXGBE virtual function devices. Each port representor corresponds to a single virtual function of that device. Using the `devargs` option `representor` the user can specify which virtual functions to create port representors for on initialization of the PF PMD by passing the VF IDs of the VFs which are required.:

```
-w DBDF,representor=[0,1,4]
```

Currently hot-plugging of representor ports is not supported so all required representors must be specified on the creation of the PF.

27.7 Supported Chipsets and NICs

- Intel 82599EB 10 Gigabit Ethernet Controller
- Intel 82598EB 10 Gigabit Ethernet Controller
- Intel 82599ES 10 Gigabit Ethernet Controller
- Intel 82599EN 10 Gigabit Ethernet Controller
- Intel Ethernet Controller X540-AT2

- Intel Ethernet Controller X550-BT2
- Intel Ethernet Controller X550-AT2
- Intel Ethernet Controller X550-AT
- Intel Ethernet Converged Network Adapter X520-SR1
- Intel Ethernet Converged Network Adapter X520-SR2
- Intel Ethernet Converged Network Adapter X520-LR1
- Intel Ethernet Converged Network Adapter X520-DA1
- Intel Ethernet Converged Network Adapter X520-DA2
- Intel Ethernet Converged Network Adapter X520-DA4
- Intel Ethernet Converged Network Adapter X520-QDA1
- Intel Ethernet Converged Network Adapter X520-T2
- Intel 10 Gigabit AF DA Dual Port Server Adapter
- Intel 10 Gigabit AT Server Adapter
- Intel 10 Gigabit AT2 Server Adapter
- Intel 10 Gigabit CX4 Dual Port Server Adapter
- Intel 10 Gigabit XF LR Server Adapter
- Intel 10 Gigabit XF SR Dual Port Server Adapter
- Intel 10 Gigabit XF SR Server Adapter
- Intel Ethernet Converged Network Adapter X540-T1
- Intel Ethernet Converged Network Adapter X540-T2
- Intel Ethernet Converged Network Adapter X550-T1
- Intel Ethernet Converged Network Adapter X550-T2

INTEL VIRTUAL FUNCTION DRIVER

Supported Intel® Ethernet Controllers (see the *DPDK Release Notes* for details) support the following modes of operation in a virtualized environment:

- **SR-IOV mode:** Involves direct assignment of part of the port resources to different guest operating systems using the PCI-SIG Single Root I/O Virtualization (SR IOV) standard, also known as “native mode” or “pass-through” mode. In this chapter, this mode is referred to as IOV mode.
- **VMDq mode:** Involves central management of the networking resources by an IO Virtual Machine (IOVM) or a Virtual Machine Monitor (VMM), also known as software switch acceleration mode. In this chapter, this mode is referred to as the Next Generation VMDq mode.

28.1 SR-IOV Mode Utilization in a DPDK Environment

The DPDK uses the SR-IOV feature for hardware-based I/O sharing in IOV mode. Therefore, it is possible to partition SR-IOV capability on Ethernet controller NIC resources logically and expose them to a virtual machine as a separate PCI function called a “Virtual Function”. Refer to [Fig. 28.1](#).

Therefore, a NIC is logically distributed among multiple virtual machines (as shown in [Fig. 28.1](#)), while still having global data in common to share with the Physical Function and other Virtual Functions. The DPDK `fm10kvf`, `i40evf`, `igbvf` or `ixgbev` as a Poll Mode Driver (PMD) serves for the Intel® 82576 Gigabit Ethernet Controller, Intel® Ethernet Controller I350 family, Intel® 82599 10 Gigabit Ethernet Controller NIC, Intel® Fortville 10/40 Gigabit Ethernet Controller NIC’s virtual PCI function, or PCIe host-interface of the Intel Ethernet Switch FM10000 Series. Meanwhile the DPDK Poll Mode Driver (PMD) also supports “Physical Function” of such NIC’s on the host.

The DPDK PF/VF Poll Mode Driver (PMD) supports the Layer 2 switch on Intel® 82576 Gigabit Ethernet Controller, Intel® Ethernet Controller I350 family, Intel® 82599 10 Gigabit Ethernet Controller, and Intel® Fortville 10/40 Gigabit Ethernet Controller NICs so that guest can choose it for inter virtual machine traffic in SR-IOV mode.

For more detail on SR-IOV, please refer to the following documents:

- [SR-IOV provides hardware based I/O sharing](#)
- [PCI-SIG-Single Root I/O Virtualization Support on IA](#)
- [Scalable I/O Virtualized Servers](#)

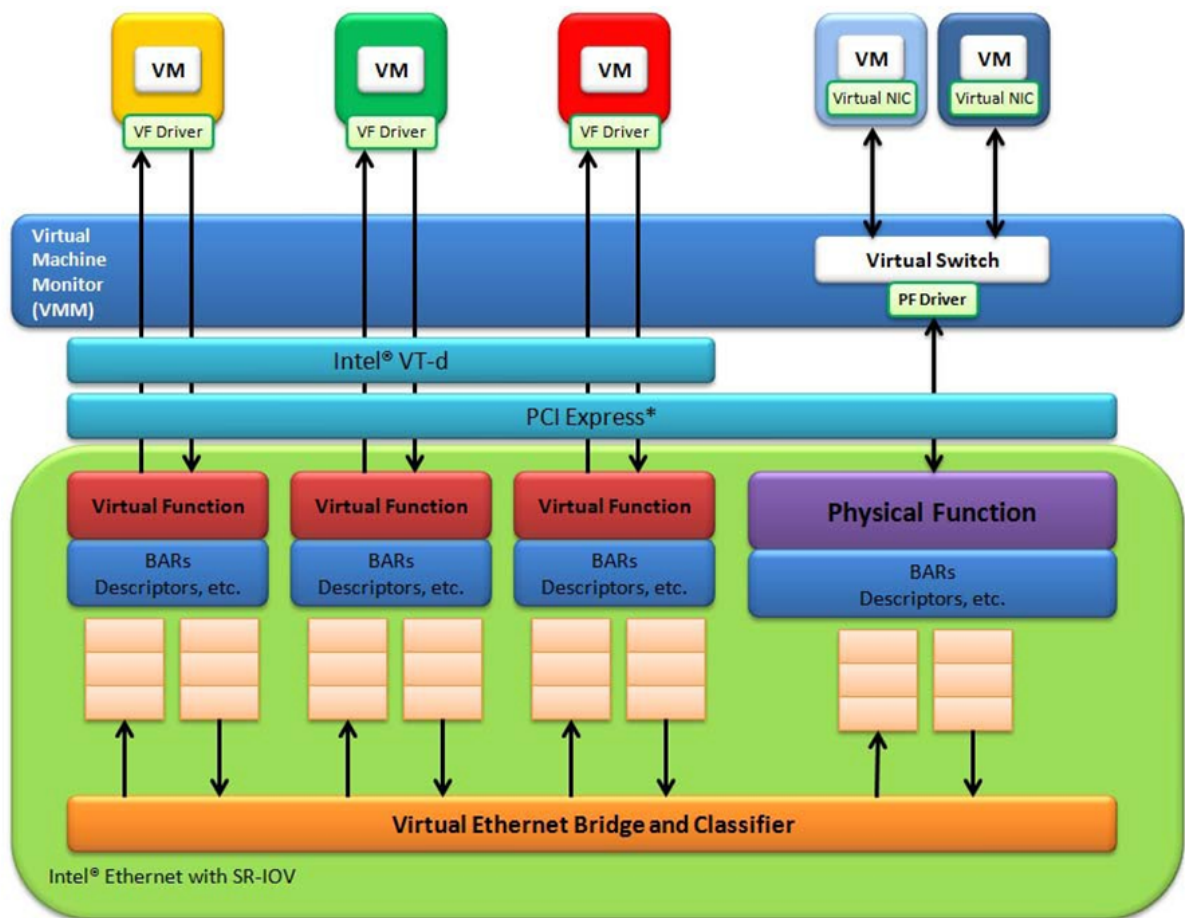


Fig. 28.1: Virtualization for a Single Port NIC in SR-IOV Mode

28.1.1 Physical and Virtual Function Infrastructure

The following describes the Physical Function and Virtual Functions infrastructure for the supported Ethernet Controller NICs.

Virtual Functions operate under the respective Physical Function on the same NIC Port and therefore have no access to the global NIC resources that are shared between other functions for the same NIC port.

A Virtual Function has basic access to the queue resources and control structures of the queues assigned to it. For global resource access, a Virtual Function has to send a request to the Physical Function for that port, and the Physical Function operates on the global resources on behalf of the Virtual Function. For this out-of-band communication, an SR-IOV enabled NIC provides a memory buffer for each Virtual Function, which is called a “Mailbox”.

Intel® Ethernet Adaptive Virtual Function

Adaptive Virtual Function (IAVF) is a SR-IOV Virtual Function with the same device id (8086:1889) on different Intel Ethernet Controller. IAVF Driver is VF driver which supports for all future Intel devices without requiring a VM update. And since this happens to be an adaptive VF driver, every new drop of the VF driver would add more and more advanced features that can be turned on in the VM if the underlying HW device supports those advanced features based on a device agnostic way without ever compromising on the base functionality. IAVF provides generic hardware interface and interface between IAVF driver and a compliant PF driver is specified.

Intel products starting Ethernet Controller 700 Series to support Adaptive Virtual Function.

The way to generate Virtual Function is like normal, and the resource of VF assignment depends on the NIC Infrastructure.

For more detail on SR-IOV, please refer to the following documents:

- [Intel® IAVF HAS](#)

Note: To use DPDK IAVF PMD on Intel® 700 Series Ethernet Controller, the device id (0x1889) need to specified during device assignment in hypervisor. Take qemu for example, the device assignment should carry the IAVF device id (0x1889) like `-device vfio-pci,x-pci-device-id=0x1889,host=03:0a.0`.

The PCIE host-interface of Intel Ethernet Switch FM10000 Series VF infrastructure

In a virtualized environment, the programmer can enable a maximum of *64 Virtual Functions (VF)* globally per PCIE host-interface of the Intel Ethernet Switch FM10000 Series device. Each VF can have a maximum of 16 queue pairs. The Physical Function in host could be only configured by the Linux* fm10k driver (in the case of the Linux Kernel-based Virtual Machine [KVM]), DPDK PMD PF driver doesn't support it yet.

For example,

- Using Linux* fm10k driver:

```
rmmod fm10k (To remove the fm10k module)
insmod fm0k.ko max_vfs=2,2 (To enable two Virtual Functions per port)
```

Virtual Function enumeration is performed in the following sequence by the Linux* pci driver for a dual-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence starting from 0 to 3. However:

- Virtual Functions 0 and 2 belong to Physical Function 0
- Virtual Functions 1 and 3 belong to Physical Function 1

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

Intel® X710/XL710 Gigabit Ethernet Controller VF Infrastructure

In a virtualized environment, the programmer can enable a maximum of *128 Virtual Functions (VF)* globally per Intel® X710/XL710 Gigabit Ethernet Controller NIC device. The number of queue pairs of each VF can be configured by `CONFIG_RTE_LIBRTE_I40E_QUEUE_NUM_PER_VF` in config file. The Physical Function in host could be either configured by the Linux* i40e driver (in the case of the Linux Kernel-based Virtual Machine [KVM]) or by DPDK PMD PF driver. When using both DPDK PMD PF/VF drivers, the whole NIC will be taken over by DPDK based application.

For example,

- Using Linux* i40e driver:

```
rmmod i40e (To remove the i40e module)
insmod i40e.ko max_vfs=2,2 (To enable two Virtual Functions per port)
```

- Using the DPDK PMD PF i40e driver:

Kernel Params: `iommu=pt, intel_iommu=on`

```
modprobe uio
insmod igb_uio
./dpdk-devbind.py -b igb_uio bb:ss.f
echo 2 > /sys/bus/pci/devices/0000\:bb\:ss.f/max_vfs (To enable two VFs on a specific PCI
```

Launch the DPDK testpmd/example or your own host daemon application using the DPDK PMD library.

Virtual Function enumeration is performed in the following sequence by the Linux* pci driver for a dual-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence starting from 0 to 3. However:

- Virtual Functions 0 and 2 belong to Physical Function 0
- Virtual Functions 1 and 3 belong to Physical Function 1

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

For Intel® X710/XL710 Gigabit Ethernet Controller, queues are in pairs. One queue pair means one receive queue and one transmit queue. The default number of queue pairs per VF is 4, and can be 16 in maximum.

Intel® 82599 10 Gigabit Ethernet Controller VF Infrastructure

The programmer can enable a maximum of 63 *Virtual Functions* and there must be *one Physical Function* per Intel® 82599 10 Gigabit Ethernet Controller NIC port. The reason for this is that the device allows for a maximum of 128 queues per port and a virtual/physical function has to have at least one queue pair (RX/TX). The current implementation of the DPDK ixgbevf driver supports a single queue pair (RX/TX) per Virtual Function. The Physical Function in host could be either configured by the Linux* ixgbe driver (in the case of the Linux Kernel-based Virtual Machine [KVM]) or by DPDK PMD PF driver. When using both DPDK PMD PF/VF drivers, the whole NIC will be taken over by DPDK based application.

For example,

- Using Linux* ixgbe driver:

```
rmmod ixgbe (To remove the ixgbe module)
insmod ixgbe max_vfs=2,2 (To enable two Virtual Functions per port)
```

- Using the DPDK PMD PF ixgbe driver:

Kernel Params: iommu=pt, intel_iommu=on

```
modprobe uio
insmod igb_uio
./dpdk-devbind.py -b igb_uio bb:ss.f
echo 2 > /sys/bus/pci/devices/0000\:bb\:ss.f/max_vfs (To enable two VFs on a specific PCI
```

Launch the DPDK testpmd/example or your own host daemon application using the DPDK PMD library.

- Using the DPDK PMD PF ixgbe driver to enable VF RSS:

Same steps as above to install the modules of uio, igb_uio, specify max_vfs for PCI device, and launch the DPDK testpmd/example or your own host daemon application using the DPDK PMD library.

The available queue number (at most 4) per VF depends on the total number of pool, which is determined by the max number of VF at PF initialization stage and the number of queue specified in config:

- If the max number of VFs (max_vfs) is set in the range of 1 to 32:

If the number of Rx queues is specified as 4 (--rxq=4 in testpmd), then there are totally 32 pools (ETH_32_POOLS), and each VF could have 4 Rx queues;

If the number of Rx queues is specified as 2 (--rxq=2 in testpmd), then there are totally 32 pools (ETH_32_POOLS), and each VF could have 2 Rx queues;

- If the max number of VFs (max_vfs) is in the range of 33 to 64:

If the number of Rx queues is specified as 4 (--rxq=4 in testpmd), then error message is expected as rxq is not correct at this case;

If the number of rxq is 2 (--rxq=2 in testpmd), then there is totally 64 pools (ETH_64_POOLS), and each VF have 2 Rx queues;

On host, to enable VF RSS functionality, rx mq mode should be set as ETH_MQ_RX_VMDQ_RSS or ETH_MQ_RX_RSS mode, and SRIOV mode should be activated (max_vfs >= 1). It also needs config VF RSS information like hash function, RSS key, RSS key length.

Note: The limitation for VF RSS on Intel® 82599 10 Gigabit Ethernet Controller is: The hash and key are shared among PF and all VF, the RETA table with 128 entries is also shared among PF and all VF; So it could not to provide a method to query the hash and reta content per VF on guest, while, if possible, please query them on host for the shared RETA information.

Virtual Function enumeration is performed in the following sequence by the Linux* pci driver for a dual-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence starting from 0 to 3. However:

- Virtual Functions 0 and 2 belong to Physical Function 0
- Virtual Functions 1 and 3 belong to Physical Function 1

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

Intel® 82576 Gigabit Ethernet Controller and Intel® Ethernet Controller I350 Family VF Infrastructure

In a virtualized environment, an Intel® 82576 Gigabit Ethernet Controller serves up to eight virtual machines (VMs). The controller has 16 TX and 16 RX queues. They are generally referred to (or thought of) as queue pairs (one TX and one RX queue). This gives the controller 16 queue pairs.

A pool is a group of queue pairs for assignment to the same VF, used for transmit and receive operations. The controller has eight pools, with each pool containing two queue pairs, that is, two TX and two RX queues assigned to each VF.

In a virtualized environment, an Intel® Ethernet Controller I350 family device serves up to eight virtual machines (VMs) per port. The eight queues can be accessed by eight different VMs if configured correctly (the i350 has 4x1GbE ports each with 8T X and 8 RX queues), that means, one Transmit and one Receive queue assigned to each VF.

For example,

- Using Linux* igb driver:

```
rmmod igb (To remove the igb module)
insmod igb max_vfs=2,2 (To enable two Virtual Functions per port)
```

- Using DPDK PMD PF igb driver:

Kernel Params: iommu=pt, intel_iommu=on modprobe uio

```
insmod igb_uio
./dpdk-devbind.py -b igb_uio bb:ss.f
echo 2 > /sys/bus/pci/devices/0000\:bb\:ss.f/max_vfs (To enable two VFs on a specific pci
```

Launch DPDK testpmd/example or your own host daemon application using the DPDK PMD library.

Virtual Function enumeration is performed in the following sequence by the Linux* pci driver for a four-port NIC. When you enable the four Virtual Functions with the above command, the four enabled functions have a Function# represented by (Bus#, Device#, Function#) in sequence, starting from 0 to 7. However:

- Virtual Functions 0 and 4 belong to Physical Function 0
- Virtual Functions 1 and 5 belong to Physical Function 1
- Virtual Functions 2 and 6 belong to Physical Function 2
- Virtual Functions 3 and 7 belong to Physical Function 3

Note: The above is an important consideration to take into account when targeting specific packets to a selected port.

28.1.2 Validated Hypervisors

The validated hypervisor is:

- KVM (Kernel Virtual Machine) with Qemu, version 0.14.0

However, the hypervisor is bypassed to configure the Virtual Function devices using the Mailbox interface, the solution is hypervisor-agnostic. Xen* and VMware* (when SR-IOV is supported) will also be able to support the DPDK with Virtual Function driver support.

28.1.3 Expected Guest Operating System in Virtual Machine

The expected guest operating systems in a virtualized environment are:

- Fedora* 14 (64-bit)
- Ubuntu* 10.04 (64-bit)

For supported kernel versions, refer to the *DPDK Release Notes*.

28.2 Setting Up a KVM Virtual Machine Monitor

The following describes a target environment:

- Host Operating System: Fedora 14
- Hypervisor: KVM (Kernel Virtual Machine) with Qemu version 0.14.0
- Guest Operating System: Fedora 14
- Linux Kernel Version: Refer to the *DPDK Getting Started Guide*
- Target Applications: l2fwd, l3fwd-vf

The setup procedure is as follows:

1. Before booting the Host OS, open **BIOS setup** and enable **Intel® VT features**.
2. While booting the Host OS kernel, pass the `intel_iommu=on` kernel command line argument using GRUB. When using DPDK PF driver on host, pass the `iommu=pt` kernel command line argument in GRUB.

3. Download qemu-kvm-0.14.0 from <http://sourceforge.net/projects/kvm/files/qemu-kvm/> and install it in the Host OS using the following steps:

When using a recent kernel (2.6.25+) with kvm modules included:

```
tar xzf qemu-kvm-release.tar.gz
cd qemu-kvm-release
./configure --prefix=/usr/local/kvm
make
sudo make install
sudo /sbin/modprobe kvm-intel
```

When using an older kernel, or a kernel from a distribution without the kvm modules, you must download (from the same link), compile and install the modules yourself:

```
tar xjf kvm-kmod-release.tar.bz2
cd kvm-kmod-release
./configure
make
sudo make install
sudo /sbin/modprobe kvm-intel
```

qemu-kvm installs in the /usr/local/bin directory.

For more details about KVM configuration and usage, please refer to:

<http://www.linux-kvm.org/page/HOWTO1>.

4. Create a Virtual Machine and install Fedora 14 on the Virtual Machine. This is referred to as the Guest Operating System (Guest OS).
5. Download and install the latest ixgbe driver from:

http://downloadcenter.intel.com/Detail_Desc.aspx?agr=Y&DwnldID=14687

6. In the Host OS

When using Linux kernel ixgbe driver, unload the Linux ixgbe driver and reload it with the max_vfs=2,2 argument:

```
rmmod ixgbe
modprobe ixgbe max_vfs=2,2
```

When using DPDK PMD PF driver, insert DPDK kernel module igb_uio and set the number of VF by sysfs max_vfs:

```
modprobe uio
insmod igb_uio
./dpdk-devbind.py -b igb_uio 02:00.0 02:00.1 0e:00.0 0e:00.1
echo 2 > /sys/bus/pci/devices/0000\:02\:00.0/max_vfs
echo 2 > /sys/bus/pci/devices/0000\:02\:00.1/max_vfs
echo 2 > /sys/bus/pci/devices/0000\:0e\:00.0/max_vfs
echo 2 > /sys/bus/pci/devices/0000\:0e\:00.1/max_vfs
```

Note: You need to explicitly specify number of vfs for each port, for example, in the command above, it creates two vfs for the first two ixgbe ports.

Let say we have a machine with four physical ixgbe ports:

```
0000:02:00.0
0000:02:00.1
```



```
0000:0e:00.0
```

```
0000:0e:00.1
```

The command above creates two vifs for device 0000:02:00.0:

```
ls -alrt /sys/bus/pci/devices/0000\:02\:00.0/virt*
lrwxrwxrwx. 1 root root 0 Apr 13 05:40 /sys/bus/pci/devices/0000:02:00.0/virtfn1 -> ../000
lrwxrwxrwx. 1 root root 0 Apr 13 05:40 /sys/bus/pci/devices/0000:02:00.0/virtfn0 -> ../000
```

It also creates two vifs for device 0000:02:00.1:

```
ls -alrt /sys/bus/pci/devices/0000\:02\:00.1/virt*
lrwxrwxrwx. 1 root root 0 Apr 13 05:51 /sys/bus/pci/devices/0000:02:00.1/virtfn1 -> ../000
lrwxrwxrwx. 1 root root 0 Apr 13 05:51 /sys/bus/pci/devices/0000:02:00.1/virtfn0 -> ../000
```

7. List the PCI devices connected and notice that the Host OS shows two Physical Functions (traditional ports) and four Virtual Functions (two for each port). This is the result of the previous step.
8. Insert the `pci_stub` module to hold the PCI devices that are freed from the default driver using the following command (see http://www.linux-kvm.org/page/How_to_assign_devices_with_VT-d_in_KVM Section 4 for more information):

```
sudo /sbin/modprobe pci-stub
```

Unbind the default driver from the PCI devices representing the Virtual Functions. A script to perform this action is as follows:

```
echo "8086 10ed" > /sys/bus/pci/drivers/pci-stub/new_id
echo 0000:08:10.0 > /sys/bus/pci/devices/0000:08:10.0/driver/unbind
echo 0000:08:10.0 > /sys/bus/pci/drivers/pci-stub/bind
```

where, 0000:08:10.0 belongs to the Virtual Function visible in the Host OS.

9. Now, start the Virtual Machine by running the following command:

```
/usr/local/kvm/bin/qemu-system-x86_64 -m 4096 -smp 4 -boot c -hda lucid.qcow2 -device pci-
```

where:

— `-m` = memory to assign

— `-smp` = number of smp cores

— `-boot` = boot option

— `-hda` = virtual disk image

— `-device` = device to attach

Note: — The `pci-assign,host=08:10.0` value indicates that you want to attach a PCI device to a Virtual Machine and the respective (Bus:Device.Function) numbers should be passed for the Virtual Function to be attached.

— `qemu-kvm-0.14.0` allows a maximum of four PCI devices assigned to a VM, but this is `qemu-kvm` version dependent since `qemu-kvm-0.14.1` allows a maximum of five PCI devices.

— `qemu-system-x86_64` also has a `-cpu` command line option that is used to select the `cpu_model` to emulate in a Virtual Machine. Therefore, it can be used as:

```
/usr/local/kvm/bin/qemu-system-x86_64 -cpu ?
```

```
(to list all available cpu_models)
```

```
/usr/local/kvm/bin/qemu-system-x86_64 -m 4096 -cpu host -smp 4 -boot c -hda lucid.qcow2 -c
```

(to use the same cpu_model equivalent to the host cpu)

For more information, please refer to: <http://wiki.qemu.org/Features/CPUModels>.

10. If use vfio-pci to pass through device instead of pci-assign, steps 8 and 9 need to be updated to bind device to vfio-pci and replace pci-assign with vfio-pci when start virtual machine.

```
sudo /sbin/modprobe vfio-pci
```

```
echo "8086 10ed" > /sys/bus/pci/drivers/vfio-pci/new_id
echo 0000:08:10.0 > /sys/bus/pci/devices/0000:08:10.0/driver/unbind
echo 0000:08:10.0 > /sys/bus/pci/drivers/vfio-pci/bind
```

```
/usr/local/kvm/bin/qemu-system-x86_64 -m 4096 -smp 4 -boot c -hda lucid.qcow2 -device vfio-
```

11. Install and run DPDK host app to take over the Physical Function. Eg.

```
make install T=x86_64-native-linux-gcc
./x86_64-native-linux-gcc/app/testpmd -l 0-3 -n 4 -- -i
```

12. Finally, access the Guest OS using vncviewer with the localhost:5900 port and check the lspci command output in the Guest OS. The virtual functions will be listed as available for use.

13. Configure and install the DPDK with an x86_64-native-linux-gcc configuration on the Guest OS as normal, that is, there is no change to the normal installation procedure.

```
make config T=x86_64-native-linux-gcc O=x86_64-native-linux-gcc
cd x86_64-native-linux-gcc
make
```

Note: If you are unable to compile the DPDK and you are getting “error: CPU you selected does not support x86-64 instruction set”, power off the Guest OS and start the virtual machine with the correct -cpu option in the qemu- system-x86_64 command as shown in step 9. You must select the best x86_64 cpu_model to emulate or you can select host option if available.

Note: Run the DPDK l2fwd sample application in the Guest OS with Hugepages enabled. For the expected benchmark performance, you must pin the cores from the Guest OS to the Host OS (taskset can be used to do this) and you must also look at the PCI Bus layout on the board to ensure you are not running the traffic over the QPI Interface.

Note:

- The Virtual Machine Manager (the Fedora package name is virt-manager) is a utility for virtual machine management that can also be used to create, start, stop and delete virtual machines. If this option is used, step 2 and 6 in the instructions provided will be different.
 - virsh, a command line utility for virtual machine management, can also be used to bind and unbind devices to a virtual machine in Ubuntu. If this option is used, step 6 in the instructions provided will be different.
 - The Virtual Machine Monitor (see Fig. 28.2) is equivalent to a Host OS with KVM installed as described in the instructions.
-

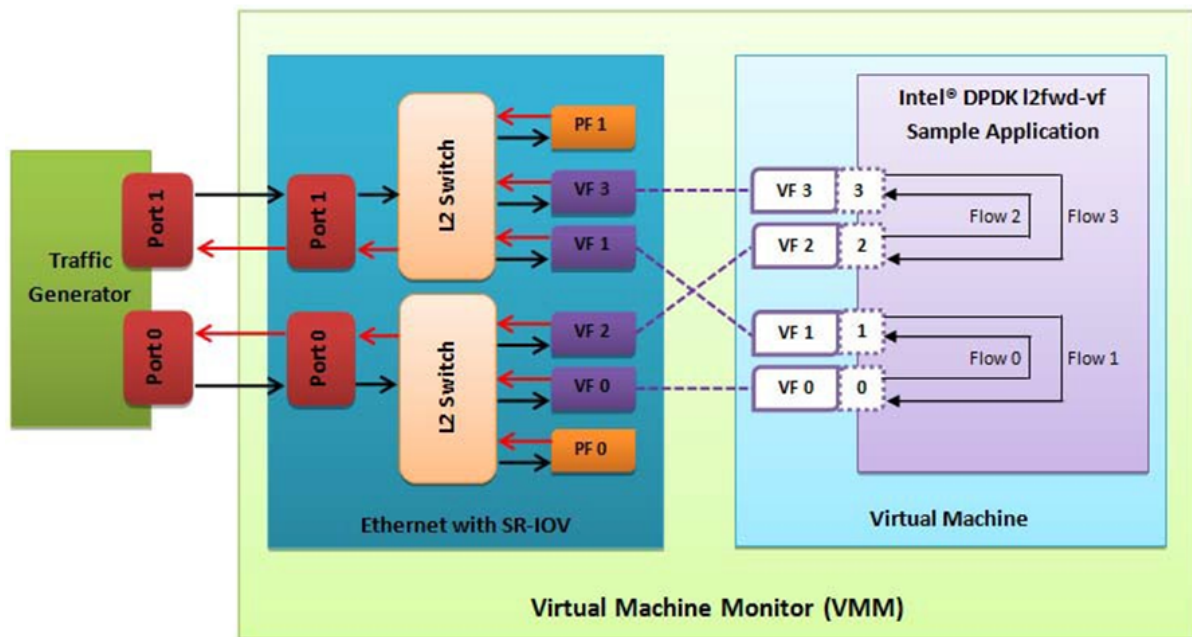


Fig. 28.2: Performance Benchmark Setup

28.3 DPDK SR-IOV PMD PF/VF Driver Usage Model

28.3.1 Fast Host-based Packet Processing

Software Defined Network (SDN) trends are demanding fast host-based packet handling. In a virtualization environment, the DPDK VF PMD driver performs the same throughput result as a non-VT native environment.

With such host instance fast packet processing, lots of services such as filtering, QoS, DPI can be offloaded on the host fast path.

Fig. 28.3 shows the scenario where some VMs directly communicate externally via a VFs, while others connect to a virtual switch and share the same uplink bandwidth.

28.4 SR-IOV (PF/VF) Approach for Inter-VM Communication

Inter-VM data communication is one of the traffic bottle necks in virtualization platforms. SR-IOV device assignment helps a VM to attach the real device, taking advantage of the bridge in the NIC. So VF-to-VF traffic within the same physical port (VM0<->VM1) have hardware acceleration. However, when VF crosses physical ports (VM0<->VM2), there is no such hardware bridge. In this case, the DPDK PMD PF driver provides host forwarding between such VMs.

Fig. 28.4 shows an example. In this case an update of the MAC address lookup tables in both the NIC and host DPDK application is required.

In the NIC, writing the destination of a MAC address belongs to another cross device VM to the PF specific pool. So when a packet comes in, its destination MAC address will match and forward to the host DPDK PMD application.

In the host DPDK application, the behavior is similar to L2 forwarding, that is, the packet is forwarded

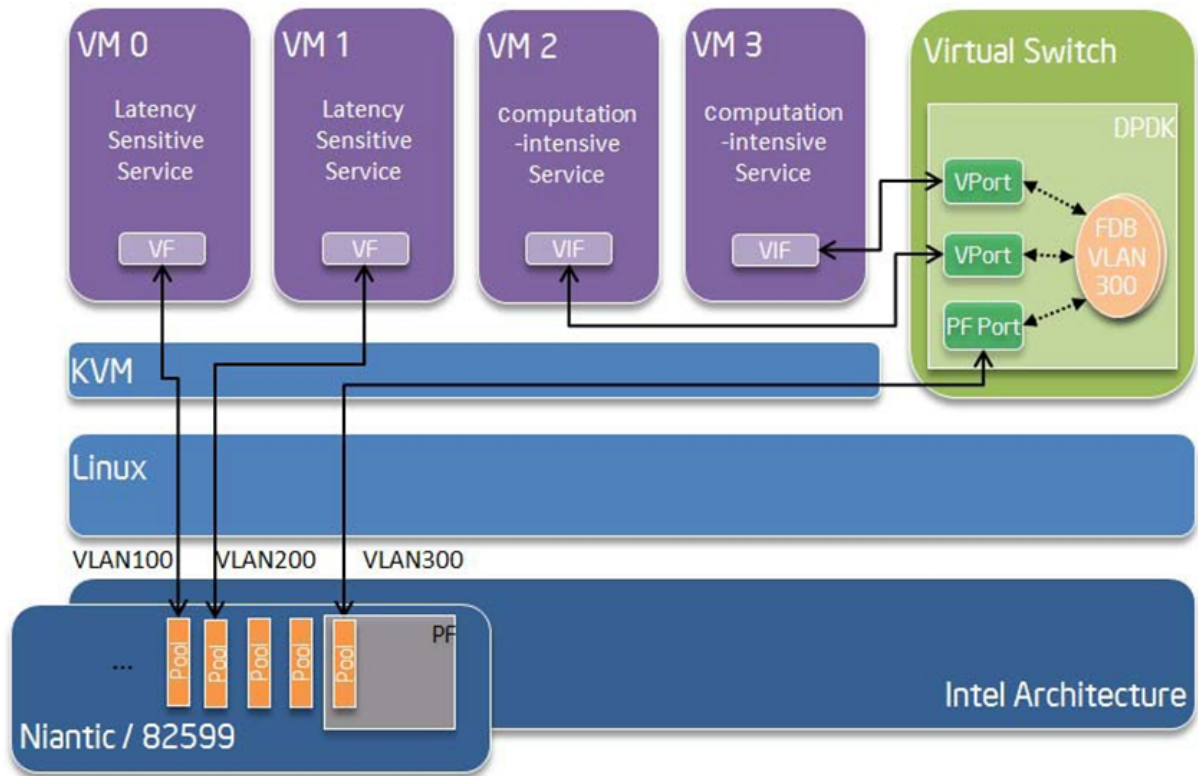


Fig. 28.3: Fast Host-based Packet Processing

to the correct PF pool. The SR-IOV NIC switch forwards the packet to a specific VM according to the MAC destination address which belongs to the destination VF on the VM.

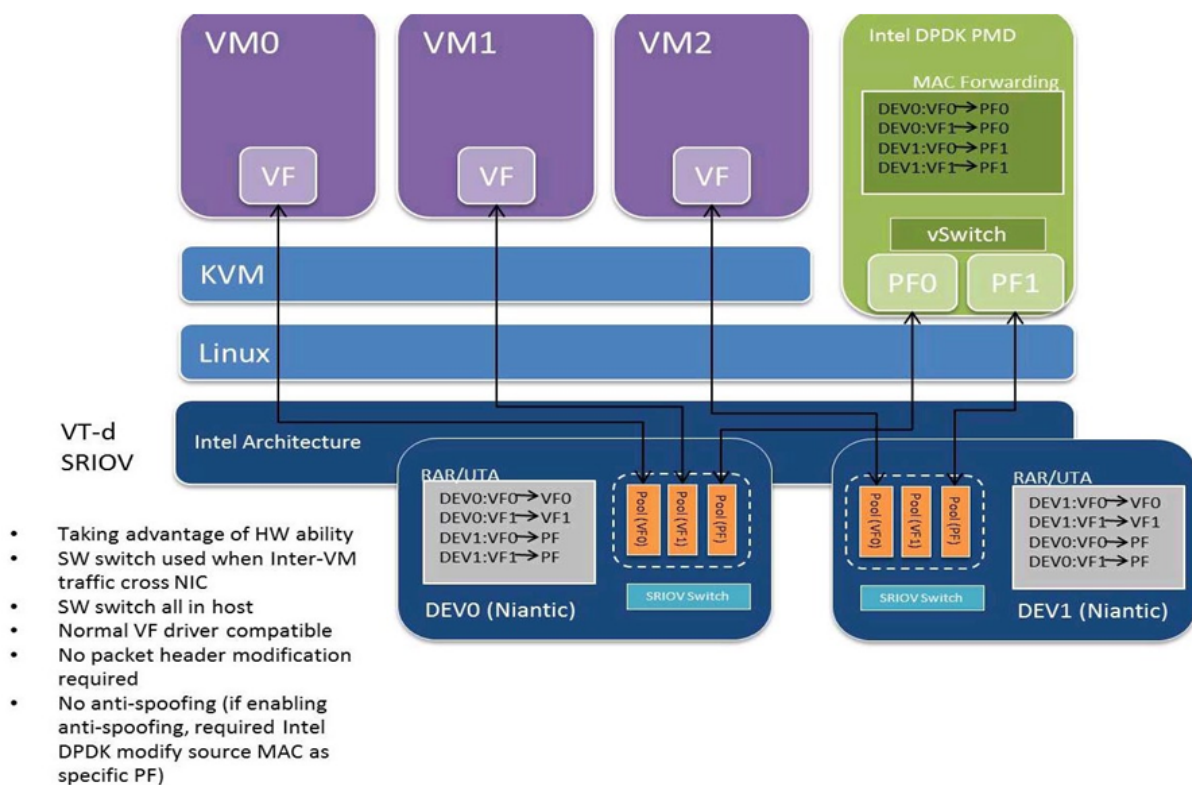


Fig. 28.4: Inter-VM Communication

KNI POLL MODE DRIVER

KNI PMD is wrapper to the `librte_kni` library.

This PMD enables using KNI without having a KNI specific application, any forwarding application can use PMD interface for KNI.

Sending packets to any DPDK controlled interface or sending to the Linux networking stack will be transparent to the DPDK application.

To create a KNI device `net_kni#` device name should be used, and this will create `kni#` Linux virtual network interface.

There is no physical device backend for the virtual KNI device.

Packets sent to the KNI Linux interface will be received by the DPDK application, and DPDK application may forward packets to a physical NIC or to a virtual device (like another KNI interface or PCAP interface).

To forward any traffic from physical NIC to the Linux networking stack, an application should control a physical port and create one virtual KNI port, and forward between two.

Using this PMD requires KNI kernel module be inserted.

29.1 Usage

EAL `--vdev` argument can be used to create KNI device instance, like:

```
testpmd --vdev=net_kni0 --vdev=net_kni1 -- -i
```

Above command will create `kni0` and `kni1` Linux network interfaces, those interfaces can be controlled by standard Linux tools.

When testpmd forwarding starts, any packets sent to `kni0` interface forwarded to the `kni1` interface and vice versa.

There is no hard limit on number of interfaces that can be created.

29.2 Default interface configuration

`librte_kni` can create Linux network interfaces with different features, feature set controlled by a configuration struct, and KNI PMD uses a fixed configuration:

```
Interface name: kni#
force bind kernel thread to a core : NO
mbuf size: (rte_pktmbuf_data_room_size(pktmbuf_pool) - RTE_PKTMBUF_HEADROOM)
mtu: (conf.mbuf_size - RTE_ETHER_HDR_LEN)
```

KNI control path is not supported with the PMD, since there is no physical backend device by default.

29.3 PMD arguments

`no_request_thread`, by default PMD creates a pthread for each KNI interface to handle Linux network interface control commands, like `ifconfig kni0 up`

With `no_request_thread` option, pthread is not created and control commands not handled by PMD.

By default request thread is enabled. And this argument should not be used most of the time, unless this PMD used with customized DPDK application to handle requests itself.

Argument usage:

```
testpmd --vdev "net_kni0,no_request_thread=1" -- -i
```

29.4 PMD log messages

If KNI kernel module (`rte_kni.ko`) not inserted, following error log printed:

```
"KNI: KNI subsystem has not been initialized. Invoke rte_kni_init() first"
```

29.5 PMD testing

It is possible to test PMD quickly using KNI kernel module loopback feature:

- Insert KNI kernel module with loopback support:

```
insmod build/kmod/rte_kni.ko lo_mode=lo_mode_fifo_skb
```

- Start `testpmd` with no physical device but two KNI virtual devices:

```
./testpmd --vdev net_kni0 --vdev net_kni1 -- -i
...
Configuring Port 0 (socket 0)
KNI: pci: 00:00:00 c580:b8
Port 0: 1A:4A:5B:7C:A2:8C
Configuring Port 1 (socket 0)
KNI: pci: 00:00:00 600:b9
Port 1: AE:95:21:07:93:DD
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

- Observe Linux interfaces

```
$ ifconfig kni0 && ifconfig kni1
kni0: flags=4098<BROADCAST,MULTICAST> mtu 1500
    ether ae:8e:79:8e:9b:c8 txqueuelen 1000 (Ethernet)
```

```

RX packets 0  bytes 0 (0.0 B)
RX errors 0  dropped 0  overruns 0  frame 0
TX packets 0  bytes 0 (0.0 B)
TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

```

```

kn1l: flags=4098<BROADCAST,MULTICAST>  mtu 1500
      ether 9e:76:43:53:3e:9b  txqueuelen 1000  (Ethernet)
RX packets 0  bytes 0 (0.0 B)
RX errors 0  dropped 0  overruns 0  frame 0
TX packets 0  bytes 0 (0.0 B)
TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

```

- Start forwarding with tx_first:

```
testpmd> start tx_first
```

- Quit and check forwarding stats:

```

testpmd> quit
Telling cores to stop...
Waiting for lcores to finish...

```

```

----- Forward statistics for port 0 -----
RX-packets: 35637905      RX-dropped: 0      RX-total: 35637905
TX-packets: 35637947      TX-dropped: 0      TX-total: 35637947
-----

```

```

----- Forward statistics for port 1 -----
RX-packets: 35637915      RX-dropped: 0      RX-total: 35637915
TX-packets: 35637937      TX-dropped: 0      TX-total: 35637937
-----

```

```

+++++++ Accumulated forward statistics for all ports+++++++
RX-packets: 71275820      RX-dropped: 0      RX-total: 71275820
TX-packets: 71275884      TX-dropped: 0      TX-total: 71275884
+++++++

```


LIQUIDIO VF POLL MODE DRIVER

The LiquidIO VF PMD library (`librte_pmd_lio`) provides poll mode driver support for Cavium LiquidIO® II server adapter VFs. PF management and VF creation can be done using kernel driver.

More information can be found at [Cavium Official Website](#).

30.1 Supported LiquidIO Adapters

- LiquidIO II CN2350 210SV/225SV
- LiquidIO II CN2350 210SVPT
- LiquidIO II CN2360 210SV/225SV
- LiquidIO II CN2360 210SVPT

30.2 Pre-Installation Configuration

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_LIO_PMD` (default `y`)
Toggle compilation of LiquidIO PMD.
- `CONFIG_RTE_LIBRTE_LIO_DEBUG_RX` (default `n`)
Toggle display of receive fast path run-time messages.
- `CONFIG_RTE_LIBRTE_LIO_DEBUG_TX` (default `n`)
Toggle display of transmit fast path run-time messages.
- `CONFIG_RTE_LIBRTE_LIO_DEBUG_MBOX` (default `n`)
Toggle display of mailbox messages.
- `CONFIG_RTE_LIBRTE_LIO_DEBUG_REGS` (default `n`)
Toggle display of register reads and writes.

30.3 SR-IOV: Prerequisites and Sample Application Notes

This section provides instructions to configure SR-IOV with Linux OS.

1. Verify SR-IOV and ARI capabilities are enabled on the adapter using `lspci`:

```
lspci -s <slot> -vvv
```

Example output:

```
[...]
Capabilities: [148 v1] Alternative Routing-ID Interpretation (ARI)
[...]
Capabilities: [178 v1] Single Root I/O Virtualization (SR-IOV)
[...]
Kernel driver in use: LiquidIO
```

2. Load the kernel module:

```
modprobe liquidio
```

3. Bring up the PF ports:

```
ifconfig p4p1 up
ifconfig p4p2 up
```

4. Change PF MTU if required:

```
ifconfig p4p1 mtu 9000
ifconfig p4p2 mtu 9000
```

5. Create VF device(s):

Echo number of VFs to be created into "sriov_numvfs" sysfs entry of the parent PF.

```
echo 1 > /sys/bus/pci/devices/0000:03:00.0/sriov_numvfs
echo 1 > /sys/bus/pci/devices/0000:03:00.1/sriov_numvfs
```

6. Assign VF MAC address:

Assign MAC address to the VF using `iproute2` utility. The syntax is:

```
ip link set <PF iface> vf <VF id> mac <macaddr>
```

Example output:

```
ip link set p4p1 vf 0 mac F2:A8:1B:5E:B4:66
```

7. Assign VF(s) to VM.

The VF devices may be passed through to the guest VM using `qemu` or `virt-manager` or `virsh` etc.

Example `qemu` guest launch command:

```
./qemu-system-x86_64 -name lio-vm -machine accel=kvm \
-cpu host -m 4096 -smp 4 \
-drive file=<disk_file>,if=none,id=disk1,format=<type> \
-device virtio-blk-pci,scsi=off,drive=disk1,id=virtio-disk1,bootindex=1 \
-device vfio-pci,host=03:00.3 -device vfio-pci,host=03:08.3
```

8. Running `testpmd`

Refer to the document *compiling and testing a PMD for a NIC* to run `testpmd` application.

Note: Use `igb_uio` instead of `vfio-pci` in VM.

Example output:

```
[...]
EAL: PCI device 0000:03:00.3 on NUMA socket 0
EAL:  probe driver: 177d:9712 net_liovf
EAL:  using IOMMU type 1 (Type 1)
PMD: net_liovf[03:00.3]INFO: DEVICE : CN23XX VF
EAL: PCI device 0000:03:08.3 on NUMA socket 0
EAL:  probe driver: 177d:9712 net_liovf
PMD: net_liovf[03:08.3]INFO: DEVICE : CN23XX VF
Interactive-mode selected
USER1: create a new mbuf pool <mbuf_pool_socket_0>: n=171456, size=2176, socket=0
Configuring Port 0 (socket 0)
PMD: net_liovf[03:00.3]INFO: Starting port 0
Port 0: F2:A8:1B:5E:B4:66
Configuring Port 1 (socket 0)
PMD: net_liovf[03:08.3]INFO: Starting port 1
Port 1: 32:76:CC:EE:56:D7
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

9. Enabling VF promiscuous mode

One VF per PF can be marked as trusted for promiscuous mode.

```
ip link set dev <PF iface> vf <VF id> trust on
```

30.4 Limitations

30.4.1 VF MTU

VF MTU is limited by PF MTU. Raise PF value before configuring VF for larger packet size.

30.4.2 VLAN offload

Tx VLAN insertion is not supported and consequently VLAN offload feature is marked partial.

30.4.3 Ring size

Number of descriptors for Rx/Tx ring should be in the range 128 to 512.

30.4.4 CRC stripping

LiquidIO adapters strip ethernet FCS of every packet coming to the host interface.

MEMIF POLL MODE DRIVER

Shared memory packet interface (memif) PMD allows for DPDK and any other client using memif (DPDK, VPP, libmemif) to communicate using shared memory. Memif is Linux only.

The created device transmits packets in a raw format. It can be used with Ethernet mode, IP mode, or Punt/Inject. At this moment, only Ethernet mode is supported in DPDK memif implementation.

Memif works in two roles: master and slave. Slave connects to master over an existing socket. It is also a producer of shared memory file and initializes the shared memory. Each interface can be connected to one peer interface at same time. The peer interface is identified by id parameter. Master creates the socket and listens for any slave connection requests. The socket may already exist on the system. Be sure to remove any such sockets, if you are creating a master interface, or you will see an “Address already in use” error. Function `rte_pmd_memif_remove()`, which removes memif interface, will also remove a listener socket, if it is not being used by any other interface.

The method to enable one or more interfaces is to use the `--vdev=net_memif0` option on the DPDK application command line. Each `--vdev=net_memif1` option given will create an interface named `net_memif0`, `net_memif1`, and so on. Memif uses unix domain socket to transmit control messages. Each memif has a unique id per socket. This id is used to identify peer interface. If you are connecting multiple interfaces using same socket, be sure to specify unique ids `id=0`, `id=1`, etc. Note that if you assign a socket to a master interface it becomes a listener socket. Listener socket can not be used by a slave interface on same client.

Table 31.1: Memif configuration options

Option	Description	Default	Valid value
<code>id=0</code>	Used to identify peer interface	0	uint32_t
<code>role=master</code>	Set memif role	slave	master slave
<code>bsize=1024</code>	Size of single packet buffer	2048	uint16_t
<code>rsize=11</code>	Log2 of ring size. If rsize is 10, actual ring size is 1024	10	1-14
<code>socket=/tmp/memif.sock</code>	Socket filename	/tmp/memif.sock	string len 108
<code>mac=01:23:45:ab:cd:ef</code>	Mac address	01:ab:23:cd:45:ef	
<code>secret=abc123</code>	Secret is an optional security option, which if specified, must be matched by peer		string len 24
<code>zero-copy=yes</code>	Enable/disable zero-copy slave mode. Only relevant to slave, requires ‘-single-file-segments’ eal argument	no	yes no

Connection establishment

In order to create memif connection, two memif interfaces, each in separate process, are needed. One interface in `master` role and other in `slave` role. It is not possible to connect two interfaces in a single process. Each interface can be connected to one interface at same time, identified by matching `id` parameter.

Memif driver uses unix domain socket to exchange required information between memif interfaces. Socket file path is specified at interface creation see *Memif configuration options* table above. If socket is used by `master` interface, it's marked as listener socket (in scope of current process) and listens to connection requests from other processes. One socket can be used by multiple interfaces. One process can have `slave` and `master` interfaces at the same time, provided each role is assigned unique socket.

For detailed information on memif control messages, see: `net/memif/memif.h`.

Slave interface attempts to make a connection on assigned socket. Process listening on this socket will extract the connection request and create a new connected socket (control channel). Then it sends the 'hello' message (`MEMIF_MSG_TYPE_HELLO`), containing configuration boundaries. Slave interface adjusts its configuration accordingly, and sends 'init' message (`MEMIF_MSG_TYPE_INIT`). This message among others contains interface `id`. Driver uses this `id` to find master interface, and assigns the control channel to this interface. If such interface is found, 'ack' message (`MEMIF_MSG_TYPE_ACK`) is sent. Slave interface sends 'add region' message (`MEMIF_MSG_TYPE_ADD_REGION`) for every region allocated. Master responds to each of these messages with 'ack' message. Same behavior applies to rings. Slave sends 'add ring' message (`MEMIF_MSG_TYPE_ADD_RING`) for every initialized ring. Master again responds to each message with 'ack' message. To finalize the connection, slave interface sends 'connect' message (`MEMIF_MSG_TYPE_CONNECT`). Upon receiving this message master maps regions to its address space, initializes rings and responds with 'connected' message (`MEMIF_MSG_TYPE_CONNECTED`). Disconnect (`MEMIF_MSG_TYPE_DISCONNECT`) can be sent by both master and slave interfaces at any time, due to driver error or if the interface is being deleted.

Files

- `net/memif/memif.h` - *control messages definitions*
- `net/memif/memif_socket.h`
- `net/memif/memif_socket.c`

31.1 Shared memory

Shared memory format

Slave is producer and master is consumer. Memory regions, are mapped shared memory files, created by memif slave and provided to master at connection establishment. Regions contain rings and buffers. Rings and buffers can also be separated into multiple regions. For no-zero-copy, rings and buffers are stored inside single memory region to reduce the number of opened files.

region `n` (no-zero-copy):

Rings		Buffers		
S2M rings	M2S rings	packet buffer 0	.	$\text{pb}((1 \ll \text{pmd->run.log2_ring_size}) * (\text{s2m} + \text{m2s})) - 1$

S2M OR M2S Rings:

ring 0	ring 1	ring <code>num_s2m_rings - 1</code>
--------	--------	-------------------------------------

ring 0:

Descriptor format

[illegible]

Bits	Name	Functionality
0	MEMIF_DESC_FLAG_NEXT	Continued buffer. When set, the packet is divided into multiple buffers. May not be contiguous.

- net/memif/memif.h - *descriptor and ring definitions*
- net/memif/rte_eth_memif.c - *eth_memif_rx() eth_memif_tx()*

143

Rings	
S2M rings	M2S rings

region n:

Buffers
memseg

Buffers are dequeued and enqueued as needed. Offset descriptor field is calculated at tx. Only single file segments mode (EAL option `--single-file-segments`) is supported, as calculating offset from multiple segments is too expensive.

31.2.1 Example: testpmd

In this example we run two instances of testpmd application and transmit packets over memif.

First create master interface:

```
#./build/app/testpmd -l 0-1 --proc-type=primary --file-prefix=pmd1 --vdev=net_memif,role=master
```

Now create slave interface (master must be already running so the slave will connect):

```
#./build/app/testpmd -l 2-3 --proc-type=primary --file-prefix=pmd2 --vdev=net_memif -- -i
```

You can also enable zero-copy on slave interface:

```
#./build/app/testpmd -l 2-3 --proc-type=primary --file-prefix=pmd2 --vdev=net_memif,zero-copy=y
```

Start forwarding packets:

```
Slave:
testpmd> start
```

```
Master:
testpmd> start tx_first
```

Show status:

```
testpmd> show port stats 0
```

For more details on testpmd please refer to `../testpmd_app_ug/index`.

31.2.2 Example: testpmd and VPP

For information on how to get and run VPP please see <https://wiki.fd.io/view/VPP>.

Start VPP in interactive mode (should be by default). Create memif master interface in VPP:

```
vpp# create interface memif id 0 master no-zero-copy
vpp# set interface state memif0/0 up
vpp# set interface ip address memif0/0 192.168.1.1/24
```

To see socket filename use show memif command:

```
vpp# show memif
sockets
id listener filename
0 yes (1) /run/vpp/memif.sock
...
```

Now create memif interface by running testpmd with these command line options:

```
#./testpmd --vdev=net_memif,socket=/run/vpp/memif.sock -- -i
```

Testpmd should now create memif slave interface and try to connect to master. In testpmd set forward option to icmpecho and start forwarding:

```
testpmd> set fwd icmpecho
testpmd> start
```

Send ping from VPP:

```
vpp# ping 192.168.1.2
64 bytes from 192.168.1.2: icmp_seq=2 ttl=254 time=36.2918 ms
64 bytes from 192.168.1.2: icmp_seq=3 ttl=254 time=23.3927 ms
64 bytes from 192.168.1.2: icmp_seq=4 ttl=254 time=24.2975 ms
64 bytes from 192.168.1.2: icmp_seq=5 ttl=254 time=17.7049 ms
```


MLX4 POLL MODE DRIVER LIBRARY

The MLX4 poll mode driver library (**librte_pmd_mlx4**) implements support for **Mellanox ConnectX-3** and **Mellanox ConnectX-3 Pro** 10/40 Gbps adapters as well as their virtual functions (VF) in SR-IOV context.

Information and documentation about this family of adapters can be found on the [Mellanox website](#). Help is also provided by the [Mellanox community](#).

There is also a [section dedicated to this poll mode driver](#).

Note: Due to external dependencies, this driver is disabled by default. It must be enabled manually by setting `CONFIG_RTE_LIBRTE_MLX4_PMD=y` and recompiling DPDK.

32.1 Implementation details

Most Mellanox ConnectX-3 devices provide two ports but expose a single PCI bus address, thus unlike most drivers, `librte_pmd_mlx4` registers itself as a PCI driver that allocates one Ethernet device per detected port.

For this reason, one cannot white/blacklist a single port without also white/blacklisting the others on the same device.

Besides its dependency on `libibverbs` (that implies `libmlx4` and associated kernel support), `librte_pmd_mlx4` relies heavily on system calls for control operations such as querying/updating the MTU and flow control parameters.

For security reasons and robustness, this driver only deals with virtual memory addresses. The way resources allocations are handled by the kernel combined with hardware specifications that allow it to handle virtual memory addresses directly ensure that DPDK applications cannot access random physical memory (or memory that does not belong to the current process).

This capability allows the PMD to coexist with kernel network interfaces which remain functional, although they stop receiving unicast packets as long as they share the same MAC address.

The `flow_isolated_mode` is supported.

Compiling `librte_pmd_mlx4` causes DPDK to be linked against `libibverbs`.

32.2 Configuration

32.2.1 Compilation options

These options can be modified in the `.config` file.

- `CONFIG_RTE_LIBRTE_MLX4_PMD` (default **n**)

Toggle compilation of `librte_pmd_mlx4` itself.

- `CONFIG_RTE_IBVERBS_LINK_DLOPEN` (default **n**)

Build PMD with additional code to make it loadable without hard dependencies on **libibverbs** nor **libmlx4**, which may not be installed on the target system.

In this mode, their presence is still required for it to run properly, however their absence won't prevent a DPDK application from starting (with `CONFIG_RTE_BUILD_SHARED_LIB` disabled) and they won't show up as missing with `ldd(1)`.

It works by moving these dependencies to a purpose-built rdma-core “glue” plug-in which must either be installed in a directory whose name is based on `CONFIG_RTE_EAL_PMD_PATH` suffixed with `-glue` if set, or in a standard location for the dynamic linker (e.g. `/lib`) if left to the default empty string (`""`).

This option has no performance impact.

- `CONFIG_RTE_IBVERBS_LINK_STATIC` (default **n**)

Embed static flavor of the dependencies **libibverbs** and **libmlx4** in the PMD shared library or the executable static binary.

- `CONFIG_RTE_LIBRTE_MLX4_DEBUG` (default **n**)

Toggle debugging code and stricter compilation flags. Enabling this option adds additional run-time checks and debugging messages at the cost of lower performance.

This option is available in meson:

- `ibverbs_link` can be `static`, `shared`, or `dlopen`.

32.2.2 Environment variables

- `MLX4_GLUE_PATH`

A list of directories in which to search for the rdma-core “glue” plug-in, separated by colons or semi-colons.

Only matters when compiled with `CONFIG_RTE_IBVERBS_LINK_DLOPEN` enabled and most useful when `CONFIG_RTE_EAL_PMD_PATH` is also set, since `LD_LIBRARY_PATH` has no effect in this case.

32.2.3 Run-time configuration

- `librte_pmd_mlx4` brings kernel network interfaces up during initialization because it is affected by their state. Forcing them down prevents packets reception.
- **ethtool** operations on related kernel interfaces also affect the PMD.

- `port` parameter [int]

This parameter provides a physical port to probe and can be specified multiple times for additional ports. All ports are probed by default if left unspecified.

- `mr_ext_memseg_en` parameter [int]

A nonzero value enables extending memseg when registering DMA memory. If enabled, the number of entries in MR (Memory Region) lookup table on datapath is minimized and it benefits performance. On the other hand, it worsens memory utilization because registered memory is pinned by kernel driver. Even if a page in the extended chunk is freed, that doesn't become reusable until the entire memory is freed.

Enabled by default.

32.2.4 Kernel module parameters

The `mlx4_core` kernel module has several parameters that affect the behavior and/or the performance of `librte_pmd_mlx4`. Some of them are described below.

- `num_vfs` (integer or triplet, optionally prefixed by device address strings)

Create the given number of VFs on the specified devices.

- `log_num_mgm_entry_size` (integer)

Device-managed flow steering (DMFS) is required by DPDK applications. It is enabled by using a negative value, the last four bits of which have a special meaning.

- `-1`: force device-managed flow steering (DMFS).
- `-7`: configure optimized steering mode to improve performance with the following limitation: VLAN filtering is not supported with this mode. This is the recommended mode in case VLAN filter is not needed.

32.3 Limitations

- For secondary process:
 - Forked secondary process not supported.
 - External memory unregistered in EAL memseg list cannot be used for DMA unless such memory has been registered by `mlx4_mr_update_ext_mp()` in primary process and remapped to the same virtual address in secondary process. If the external memory is registered by primary process but has different virtual address in secondary process, unexpected error may happen.
- CRC stripping is supported by default and always reported as “true”. The ability to enable/disable CRC stripping requires OFED version 4.3-1.5.0.0 and above or rdma-core version v18 and above.
- TSO (Transmit Segmentation Offload) is supported in OFED version 4.4 and above.

32.4 Prerequisites

This driver relies on external libraries and kernel drivers for resources allocations and initialization. The following dependencies are not part of DPDK and must be installed separately:

- **libibverbs** (provided by rdma-core package)

User space verbs framework used by librte_pmd_mlx4. This library provides a generic interface between the kernel and low-level user space drivers such as libmlx4.

It allows slow and privileged operations (context initialization, hardware resources allocations) to be managed by the kernel and fast operations to never leave user space.

- **libmlx4** (provided by rdma-core package)

Low-level user space driver library for Mellanox ConnectX-3 devices, it is automatically loaded by libibverbs.

This library basically implements send/receive calls to the hardware queues.

- **Kernel modules**

They provide the kernel-side verbs API and low level device drivers that manage actual hardware initialization and resources sharing with user space processes.

Unlike most other PMDs, these modules must remain loaded and bound to their devices:

- `mlx4_core`: hardware driver managing Mellanox ConnectX-3 devices.
- `mlx4_en`: Ethernet device driver that provides kernel network interfaces.
- `mlx4_ib`: InfiniBand device driver.
- `ib_uverbs`: user space driver for verbs (entry point for libibverbs).

- **Firmware update**

Mellanox OFED releases include firmware updates for ConnectX-3 adapters.

Because each release provides new features, these updates must be applied to match the kernel modules and libraries they come with.

Note: Both libraries are BSD and GPL licensed. Linux kernel modules are GPL licensed.

Depending on system constraints and user preferences either RDMA core library with a recent enough Linux kernel release (recommended) or Mellanox OFED, which provides compatibility with older releases.

32.4.1 Current RDMA core package and Linux kernel (recommended)

- Minimal Linux kernel version: 4.14.
- Minimal RDMA core version: v15 (see [RDMA core installation documentation](#)).
- Starting with rdma-core v21, static libraries can be built:

```
cd build
CFLAGS=-fPIC cmake -DIN_PLACE=1 -DENABLE_STATIC=1 -GNinja ..
ninja
```

If rdma-core libraries are built but not installed, DPDK makefile can link them, thanks to these environment variables:

- EXTRA_CFLAGS=-I/path/to/rdma-core/build/include
- EXTRA_LDFLAGS=-L/path/to/rdma-core/build/lib
- PKG_CONFIG_PATH=/path/to/rdma-core/build/lib/pkgconfig

32.4.2 Mellanox OFED as a fallback

- Mellanox OFED version: **4.4, 4.5, 4.6**.
- firmware version: **2.42.5000** and above.

Note: Several versions of Mellanox OFED are available. Installing the version this DPDK release was developed and tested against is strongly recommended. Please check the [prerequisites](#).

Installing Mellanox OFED

1. Download latest Mellanox OFED.
2. Install the required libraries and kernel modules either by installing only the required set, or by installing the entire Mellanox OFED:

For bare metal use:

```
./mlnxofedinstall --dpdk --upstream-libs
```

For SR-IOV hypervisors use:

```
./mlnxofedinstall --dpdk --upstream-libs --enable-sriov --hypervisor
```

For SR-IOV virtual machine use:

```
./mlnxofedinstall --dpdk --upstream-libs --guest
```

3. Verify the firmware is the correct one:

```
ibv_devinfo
```
4. Set all ports links to Ethernet, follow instructions on the screen:

```
connectx_port_config
```

5. Continue with [section 2 of the Quick Start Guide](#).

32.5 Quick Start Guide

1. Set all ports links to Ethernet:

```
PCI=<NIC PCI address>
echo eth > "/sys/bus/pci/devices/$PCI/mlx4_port0"
echo eth > "/sys/bus/pci/devices/$PCI/mlx4_port1"
```

Note: If using Mellanox OFED one can permanently set the port link to Ethernet using `connectx_port_config` tool provided by it. *Mellanox OFED as a fallback:*

2. In case of bare metal or hypervisor, configure optimized steering mode by adding the following line to `/etc/modprobe.d/mlx4_core.conf`:

```
options mlx4_core log_num_mgm_entry_size=-7
```

Note: If VLAN filtering is used, set `log_num_mgm_entry_size=-1`. Performance degradation can occur on this case.

3. Restart the driver:

```
/etc/init.d/openibd restart
```

or:

```
service openibd restart
```

4. Compile DPDK and you are ready to go. See instructions on Development Kit Build System

32.6 Performance tuning

1. Verify the optimized steering mode is configured:

```
cat /sys/module/mlx4_core/parameters/log_num_mgm_entry_size
```

2. Use the CPU near local NUMA node to which the PCIe adapter is connected, for better performance. For VMs, verify that the right CPU and NUMA node are pinned according to the above. Run:

```
lstopo-no-graphics
```

to identify the NUMA node to which the PCIe adapter is connected.

3. If more than one adapter is used, and root complex capabilities allow to put both adapters on the same NUMA node without PCI bandwidth degradation, it is recommended to locate both adapters on the same NUMA node. This in order to forward packets from one to the other without NUMA performance penalty.

4. Disable pause frames:

```
ethtool -A <netdev> rx off tx off
```

5. Verify IO non-posted prefetch is disabled by default. This can be checked via the BIOS configuration. Please contact your server provider for more information about the settings.

Note: On some machines, depends on the machine integrator, it is beneficial to set the PCI max read request parameter to 1K. This can be done in the following way:

To query the read request size use:

```
setpci -s <NIC PCI address> 68.w
```

If the output is different than 3XXX, set it by:

```
setpci -s <NIC PCI address> 68.w=3XXX
```

The XXX can be different on different systems. Make sure to configure according to the setpci output.

6. To minimize overhead of searching Memory Regions:

- ‘-socket-mem’ is recommended to pin memory by predictable amount.
- Configure per-lcore cache when creating Mempools for packet buffer.
- Refrain from dynamically allocating/freeing memory in run-time.

32.7 Usage example

This section demonstrates how to launch **testpmd** with Mellanox ConnectX-3 devices managed by **librte_pmd_mlx4**.

1. Load the kernel modules:

```
modprobe -a ib_uverbs mlx4_en mlx4_core mlx4_ib
```

Alternatively if MLNX_OFED is fully installed, the following script can be run:

```
/etc/init.d/openibd restart
```

Note: User space I/O kernel modules (uio and igb_uio) are not used and do not have to be loaded.

2. Make sure Ethernet interfaces are in working order and linked to kernel verbs. Related sysfs entries should be present:

```
ls -d /sys/class/net/*/device/infiniband_verbs/uverbs* | cut -d / -f 5
```

Example output:

```
eth2
eth3
eth4
eth5
```

3. Optionally, retrieve their PCI bus addresses for whitelisting:

```
{
    for intf in eth2 eth3 eth4 eth5;
    do
        (cd "/sys/class/net/${intf}/device/" && pwd -P);
    done;
} |
sed -n 's,.*\/\(.*\),-w \1,p'
```

Example output:

```
-w 0000:83:00.0
-w 0000:83:00.0
-w 0000:84:00.0
-w 0000:84:00.0
```

Note: There are only two distinct PCI bus addresses because the Mellanox ConnectX-3 adapters installed on this system are dual port.

4. Request huge pages:

```
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages/nr_hugepages
```

5. Start testpmd with basic parameters:

```
testpmd -l 8-15 -n 4 -w 0000:83:00.0 -w 0000:84:00.0 -- --rxq=2 --txq=2 -i
```

Example output:

```
[...]
EAL: PCI device 0000:83:00.0 on NUMA socket 1
EAL: probe driver: 15b3:1007 librte_pmd_mlx4
PMD: librte_pmd_mlx4: PCI information matches, using device "mlx4_0" (VF: false)
PMD: librte_pmd_mlx4: 2 port(s) detected
PMD: librte_pmd_mlx4: port 1 MAC address is 00:02:c9:b5:b7:50
PMD: librte_pmd_mlx4: port 2 MAC address is 00:02:c9:b5:b7:51
EAL: PCI device 0000:84:00.0 on NUMA socket 1
EAL: probe driver: 15b3:1007 librte_pmd_mlx4
PMD: librte_pmd_mlx4: PCI information matches, using device "mlx4_1" (VF: false)
PMD: librte_pmd_mlx4: 2 port(s) detected
PMD: librte_pmd_mlx4: port 1 MAC address is 00:02:c9:b5:ba:b0
PMD: librte_pmd_mlx4: port 2 MAC address is 00:02:c9:b5:ba:b1
Interactive-mode selected
Configuring Port 0 (socket 0)
PMD: librte_pmd_mlx4: 0x867d60: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx4: 0x867d60: RX queues number update: 0 -> 2
Port 0: 00:02:C9:B5:B7:50
Configuring Port 1 (socket 0)
PMD: librte_pmd_mlx4: 0x867da0: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx4: 0x867da0: RX queues number update: 0 -> 2
Port 1: 00:02:C9:B5:B7:51
Configuring Port 2 (socket 0)
PMD: librte_pmd_mlx4: 0x867de0: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx4: 0x867de0: RX queues number update: 0 -> 2
Port 2: 00:02:C9:B5:BA:B0
Configuring Port 3 (socket 0)
PMD: librte_pmd_mlx4: 0x867e20: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx4: 0x867e20: RX queues number update: 0 -> 2
Port 3: 00:02:C9:B5:BA:B1
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 40000 Mbps - full-duplex
Port 2 Link Up - speed 10000 Mbps - full-duplex
Port 3 Link Up - speed 40000 Mbps - full-duplex
Done
testpmd>
```


MLX5 POLL MODE DRIVER

The MLX5 poll mode driver library (`librte_pmd_mlx5`) provides support for **Mellanox ConnectX-4**, **Mellanox ConnectX-4 Lx**, **Mellanox ConnectX-5**, **Mellanox ConnectX-6**, **Mellanox ConnectX-6 Dx** and **Mellanox BlueField** families of 10/25/40/50/100/200 Gb/s adapters as well as their virtual functions (VF) in SR-IOV context.

Information and documentation about these adapters can be found on the [Mellanox website](#). Help is also provided by the [Mellanox community](#).

There is also a [section dedicated to this poll mode driver](#).

Note: Due to external dependencies, this driver is disabled in default configuration of the “make” build. It can be enabled with `CONFIG_RTE_LIBRTE_MLX5_PMD=y` or by using “meson” build system which will detect dependencies.

33.1 Design

Besides its dependency on `libibverbs` (that implies `libmlx5` and associated kernel support), `librte_pmd_mlx5` relies heavily on system calls for control operations such as querying/updating the MTU and flow control parameters.

For security reasons and robustness, this driver only deals with virtual memory addresses. The way resources allocations are handled by the kernel, combined with hardware specifications that allow to handle virtual memory addresses directly, ensure that DPDK applications cannot access random physical memory (or memory that does not belong to the current process).

This capability allows the PMD to coexist with kernel network interfaces which remain functional, although they stop receiving unicast packets as long as they share the same MAC address. This means legacy linux control tools (for example: `ethtool`, `ifconfig` and more) can operate on the same network interfaces that owned by the DPDK application.

The PMD can use `libibverbs` and `libmlx5` to access the device firmware or directly the hardware components. There are different levels of objects and bypassing abilities to get the best performances:

- Verbs is a complete high-level generic API
- Direct Verbs is a device-specific API
- DevX allows to access firmware objects
- Direct Rules manages flow steering at low-level hardware layer

Enabling `librte_pmd_mlx5` causes DPDK applications to be linked against `libibverbs`.

33.2 Features

- Multi arch support: x86_64, POWER8, ARMv8, i686.
- Multiple TX and RX queues.
- Support for scattered TX and RX frames.
- IPv4, IPv6, TCPv4, TCPv6, UDPv4 and UDPv6 RSS on any number of queues.
- Several RSS hash keys, one for each flow type.
- Default RSS operation with no hash key specification.
- Configurable RETA table.
- Link flow control (pause frame).
- Support for multiple MAC addresses.
- VLAN filtering.
- RX VLAN stripping.
- TX VLAN insertion.
- RX CRC stripping configuration.
- Promiscuous mode on PF and VF.
- Multicast promiscuous mode on PF and VF.
- Hardware checksum offloads.
- Flow director (RTE_FDIR_MODE_PERFECT, RTE_FDIR_MODE_PERFECT_MAC_VLAN and RTE_ETH_FDIR_REJECT).
- Flow API, including flow_isolated_mode.
- Multiple process.
- KVM and VMware ESX SR-IOV modes are supported.
- RSS hash result is supported.
- Hardware TSO for generic IP or UDP tunnel, including VXLAN and GRE.
- Hardware checksum Tx offload for generic IP or UDP tunnel, including VXLAN and GRE.
- RX interrupts.
- Statistics query including Basic, Extended and per queue.
- Rx HW timestamp.
- Tunnel types: VXLAN, L3 VXLAN, VXLAN-GPE, GRE, MPLSoGRE, MPLSoUDP, IP-in-IP, Geneve.
- Tunnel HW offloads: packet type, inner/outer RSS, IP and UDP checksum verification.
- NIC HW offloads: encapsulation (vxlan, gre, mplsoudp, mplsogre), NAT, routing, TTL increment/decrement, count, drop, mark. For details please see *Supported hardware offloads*.
- Flow insertion rate of more then million flows per second, when using Direct Rules.
- Support for multiple rte_flow groups.

- Hardware LRO.

33.3 Limitations

- For secondary process:
 - Forked secondary process not supported.
 - External memory unregistered in EAL memseg list cannot be used for DMA unless such memory has been registered by `mlx5_mr_update_ext_mp()` in primary process and remapped to the same virtual address in secondary process. If the external memory is registered by primary process but has different virtual address in secondary process, unexpected error may happen.

- When using Verbs flow engine (`dv_flow_en = 0`), flow pattern without any specific VLAN will match for VLAN packets as well:

When VLAN spec is not specified in the pattern, the matching rule will be created with VLAN as a wild card. Meaning, the flow rule:

```
flow create 0 ingress pattern eth / vlan vid is 3 / ipv4 / end ...
```

Will only match vlan packets with vid=3. and the flow rule:

```
flow create 0 ingress pattern eth / ipv4 / end ...
```

Will match any ipv4 packet (VLAN included).

- When using DV flow engine (`dv_flow_en = 1`), flow pattern without VLAN item will match untagged packets only. The flow rule:

```
flow create 0 ingress pattern eth / ipv4 / end ...
```

Will match untagged packets only. The flow rule:

```
flow create 0 ingress pattern eth / vlan / ipv4 / end ...
```

Will match tagged packets only, with any VLAN ID value. The flow rule:

```
flow create 0 ingress pattern eth / vlan vid is 3 / ipv4 / end ...
```

Will only match tagged packets with VLAN ID 3.

- VLAN pop offload command:
 - Flow rules having a VLAN pop offload command as one of their actions and are lacking a match on VLAN as one of their items are not supported.
 - The command is not supported on egress traffic.
- VLAN push offload is not supported on ingress traffic.
- VLAN set PCP offload is not supported on existing headers.
- A multi segment packet must have not more segments than reported by `dev_infos_get()` in `tx_desc_lim.nb_seg_max` field. This value depends on maximal supported Tx descriptor size and `txq_inline_min` settings and may be from 2 (worst case forced by maximal inline settings) to 58.
- Flows with a VXLAN Network Identifier equal (or ends to be equal) to 0 are not supported.
- VXLAN TSO and checksum offloads are not supported on VM.

- L3 VXLAN and VXLAN-GPE tunnels cannot be supported together with MPLSoGRE and MPLSoUDP.
- Match on Geneve header supports the following fields only:
 - VNI
 - OAM
 - protocol type
 - options length Currently, the only supported options length value is 0.
- VF: flow rules created on VF devices can only match traffic targeted at the configured MAC addresses (see `rte_eth_dev_mac_addr_add()`).

Note: MAC addresses not already present in the bridge table of the associated kernel network device will be added and cleaned up by the PMD when closing the device. In case of ungraceful program termination, some entries may remain present and should be removed manually by other means.

- When Multi-Packet Rx queue is configured (`mprq_en`), a Rx packet can be externally attached to a user-provided mbuf with having `EXT_ATTACHED_MBUF` in `ol_flags`. As the mempool for the external buffer is managed by PMD, all the Rx mbufs must be freed before the device is closed. Otherwise, the mempool of the external buffers will be freed by PMD and the application which still holds the external buffers may be corrupted.
- If Multi-Packet Rx queue is configured (`mprq_en`) and Rx CQE compression is enabled (`rxq_cqe_comp_en`) at the same time, RSS hash result is not fully supported. Some Rx packets may not have `PKT_RX_RSS_HASH`.
- IPv6 Multicast messages are not supported on VM, while promiscuous mode and allmulticast mode are both set to off. To receive IPv6 Multicast messages on VM, explicitly set the relevant MAC address using `rte_eth_dev_mac_addr_add()` API.
- The amount of descriptors in Tx queue may be limited by data inline settings. Inline data require the more descriptor building blocks and overall block amount may exceed the hardware supported limits. The application should reduce the requested Tx size or adjust data inline settings with `txq_inline_max` and `txq_inline_mpw` devargs keys.
- E-Switch decapsulation Flow:
 - can be applied to PF port only.
 - must specify VF port action (packet redirection from PF to VF).
 - optionally may specify tunnel inner source and destination MAC addresses.
- E-Switch encapsulation Flow:
 - can be applied to VF ports only.
 - must specify PF port action (packet redirection from VF to PF).
- ICMP/ICMP6 code/type matching, IP-in-IP and MPLS flow matching are all mutually exclusive features which cannot be supported together (see [Firmware configuration](#)).
- LRO:
 - Requires DevX and DV flow to be enabled.

- KEEP_CRC offload cannot be supported with LRO.
- The first mbuf length, without head-room, must be big enough to include the TCP header (122B).
- Rx queue with LRO offload enabled, receiving a non-LRO packet, can forward it with size limited to max LRO size, not to max RX packet length.

33.4 Statistics

MLX5 supports various methods to report statistics:

Port statistics can be queried using `rte_eth_stats_get()`. The received and sent statistics are through SW only and counts the number of packets received or sent successfully by the PMD. The missed counter is the amount of packets that could not be delivered to SW because a queue was full. Packets not received due to congestion in the bus or on the NIC can be queried via the `rx_discards_phy_xstats` counter.

Extended statistics can be queried using `rte_eth_xstats_get()`. The extended statistics expose a wider set of counters counted by the device. The extended port statistics counts the number of packets received or sent successfully by the port. As Mellanox NICs are using the Bifurcated Linux Driver those counters counts also packet received or sent by the Linux kernel. The counters with `_phy` suffix counts the total events on the physical port, therefore not valid for VF.

Finally per-flow statistics can be queried using `rte_flow_query` when attaching a count action for specific flow. The flow counter counts the number of packets received successfully by the port and match the specific flow.

33.5 Configuration

33.5.1 Compilation options

These options can be modified in the `.config` file.

- `CONFIG_RTE_LIBRTE_MLX5_PMD` (default **n**)

Toggle compilation of `librte_pmd_mlx5` itself.

- `CONFIG_RTE_IBVERBS_LINK_DLOPEN` (default **n**)

Build PMD with additional code to make it loadable without hard dependencies on **libibverbs** nor **libmlx5**, which may not be installed on the target system.

In this mode, their presence is still required for it to run properly, however their absence won't prevent a DPDK application from starting (with `CONFIG_RTE_BUILD_SHARED_LIB` disabled) and they won't show up as missing with `ldd(1)`.

It works by moving these dependencies to a purpose-built rdma-core “glue” plug-in which must either be installed in a directory whose name is based on `CONFIG_RTE_EAL_PMD_PATH` suffixed with `-glue` if set, or in a standard location for the dynamic linker (e.g. `/lib`) if left to the default empty string (`" "`).

This option has no performance impact.

- `CONFIG_RTE_IBVERBS_LINK_STATIC` (default **n**)

Embed static flavor of the dependencies **libibverbs** and **libmlx5** in the PMD shared library or the executable static binary.

- `CONFIG_RTE_LIBRTE_MLX5_DEBUG` (default **n**)

Toggle debugging code and stricter compilation flags. Enabling this option adds additional run-time checks and debugging messages at the cost of lower performance.

Note: For BlueField, target should be set to `arm64-bluefield-linux-gcc`. This will enable `CONFIG_RTE_LIBRTE_MLX5_PMD` and set `RTE_CACHE_LINE_SIZE` to 64. Default armv8a configuration of make build and meson build set it to 128 then brings performance degradation.

This option is available in meson:

- `ibverbs_link` can be `static`, `shared`, or `dlopen`.

33.5.2 Environment variables

- `MLX5_GLUE_PATH`

A list of directories in which to search for the rdma-core “glue” plug-in, separated by colons or semi-colons.

Only matters when compiled with `CONFIG_RTE_IBVERBS_LINK_DLOPEN` enabled and most useful when `CONFIG_RTE_EAL_PMD_PATH` is also set, since `LD_LIBRARY_PATH` has no effect in this case.

- `MLX5_SHUT_UP_BF`

Configures HW Tx doorbell register as IO-mapped.

By default, the HW Tx doorbell is configured as a write-combining register. The register would be flushed to HW usually when the write-combining buffer becomes full, but it depends on CPU design.

Except for vectorized Tx burst routines, a write memory barrier is enforced after updating the register so that the update can be immediately visible to HW.

When vectorized Tx burst is called, the barrier is set only if the burst size is not aligned to `MLX5_VPMD_TX_MAX_BURST`. However, setting this environmental variable will bring better latency even though the maximum throughput can slightly decline.

33.5.3 Run-time configuration

- `librte_pmd_mlx5` brings kernel network interfaces up during initialization because it is affected by their state. Forcing them down prevents packets reception.
- **ethtool** operations on related kernel interfaces also affect the PMD.
- `rxq_cqe_comp_en` parameter [int]

A nonzero value enables the compression of CQE on RX side. This feature allows to save PCI bandwidth and improve performance. Enabled by default.

Supported on:

- x86_64 with ConnectX-4, ConnectX-4 Lx, ConnectX-5, ConnectX-6, ConnectX-6 Dx and BlueField.
- POWER9 and ARMv8 with ConnectX-4 Lx, ConnectX-5, ConnectX-6, ConnectX-6 Dx and BlueField.

- `rxq_cqe_pad_en` parameter [int]

A nonzero value enables 128B padding of CQE on RX side. The size of CQE is aligned with the size of a cacheline of the core. If cacheline size is 128B, the CQE size is configured to be 128B even though the device writes only 64B data on the cacheline. This is to avoid unnecessary cache invalidation by device's two consecutive writes on to one cacheline. However in some architecture, it is more beneficial to update entire cacheline with padding the rest 64B rather than striding because read-modify-write could drop performance a lot. On the other hand, writing extra data will consume more PCIe bandwidth and could also drop the maximum throughput. It is recommended to empirically set this parameter. Disabled by default.

Supported on:

- CPU having 128B cacheline with ConnectX-5 and BlueField.

- `rxq_pkt_pad_en` parameter [int]

A nonzero value enables padding Rx packet to the size of cacheline on PCI transaction. This feature would waste PCI bandwidth but could improve performance by avoiding partial cacheline write which may cause costly read-modify-copy in memory transaction on some architectures. Disabled by default.

Supported on:

- x86_64 with ConnectX-4, ConnectX-4 Lx, ConnectX-5, ConnectX-6, ConnectX-6 Dx and BlueField.
- POWER8 and ARMv8 with ConnectX-4 Lx, ConnectX-5, ConnectX-6, ConnectX-6 Dx and BlueField.

- `mprq_en` parameter [int]

A nonzero value enables configuring Multi-Packet Rx queues. Rx queue is configured as Multi-Packet RQ if the total number of Rx queues is `rxqs_min_mprq` or more. Disabled by default.

Multi-Packet Rx Queue (MPRQ a.k.a Striding RQ) can further save PCIe bandwidth by posting a single large buffer for multiple packets. Instead of posting a buffers per a packet, one large buffer is posted in order to receive multiple packets on the buffer. A MPRQ buffer consists of multiple fixed-size strides and each stride receives one packet. MPRQ can improve throughput for small-packet traffic.

When MPRQ is enabled, `max_rx_pkt_len` can be larger than the size of user-provided mbuf even if `DEV_RX_OFFLOAD_SCATTER` isn't enabled. PMD will configure large stride size enough to accommodate `max_rx_pkt_len` as long as device allows. Note that this can waste system memory compared to enabling Rx scatter and multi-segment packet.

- `mprq_log_stride_num` parameter [int]

Log 2 of the number of strides for Multi-Packet Rx queue. Configuring more strides can reduce PCIe traffic further. If configured value is not in the range of device capability, the default value will be set with a warning message. The default value is 4 which is 16 strides per a buffer, valid only if `mprq_en` is set.

The size of Rx queue should be bigger than the number of strides.

- `mprq_log_stride_size` parameter [int]

Log 2 of the size of a stride for Multi-Packet Rx queue. Configuring a smaller stride size can save some memory and reduce probability of a depletion of all available strides due to unreleased packets by an application. If configured value is not in the range of device capability, the default value will be set with a warning message. The default value is 11 which is 2048 bytes per a stride, valid only if `mprq_en` is set. With `mprq_log_stride_size` set it is possible for a packet to span across multiple strides. This mode allows support of jumbo frames (9K) with MPRQ. The memcopy of some packets (or part of a packet if Rx scatter is configured) may be required in case there is no space left for a head room at the end of a stride which incurs some performance penalty.

- `mprq_max_memcpy_len` parameter [int]

The maximum length of packet to memcopy in case of Multi-Packet Rx queue. Rx packet is mem-copied to a user-provided mbuf if the size of Rx packet is less than or equal to this parameter. Otherwise, PMD will attach the Rx packet to the mbuf by external buffer attachment - `rte_pktmbuf_attach_extbuf()`. A mempool for external buffers will be allocated and managed by PMD. If Rx packet is externally attached, `ol_flags` field of the mbuf will have `EXT_ATTACHED_MBUF` and this flag must be preserved. `RTE_MBUF_HAS_EXTBUF()` checks the flag. The default value is 128, valid only if `mprq_en` is set.

- `rxqs_min_mprq` parameter [int]

Configure Rx queues as Multi-Packet RQ if the total number of Rx queues is greater or equal to this value. The default value is 12, valid only if `mprq_en` is set.

- `txq_inline` parameter [int]

Amount of data to be inlined during TX operations. This parameter is deprecated and converted to the new parameter `txq_inline_max` providing partial compatibility.

- `txqs_min_inline` parameter [int]

Enable inline data send only when the number of TX queues is greater or equal to this value.

This option should be used in combination with `txq_inline_max` and `txq_inline_mpw` below and does not affect `txq_inline_min` settings above.

If this option is not specified the default value 16 is used for BlueField and 8 for other platforms

The data inlining consumes the CPU cycles, so this option is intended to auto enable inline data if we have enough Tx queues, which means we have enough CPU cores and PCI bandwidth is getting more critical and CPU is not supposed to be bottleneck anymore.

The copying data into WQE improves latency and can improve PPS performance when PCI back pressure is detected and may be useful for scenarios involving heavy traffic on many queues.

Because additional software logic is necessary to handle this mode, this option should be used with care, as it may lower performance when back pressure is not expected.

If inline data are enabled it may affect the maximal size of Tx queue in descriptors because the inline data increase the descriptor size and queue size limits supported by hardware may be exceeded.

- `txq_inline_min` parameter [int]

Minimal amount of data to be inlined into WQE during Tx operations. NICs may require this minimal data amount to operate correctly. The exact value may depend on NIC operation mode, requested offloads, etc. It is strongly recommended to omit this parameter and use the default values. Anyway, applications using this parameter should take into consideration that specifying an inconsistent value may prevent the NIC from sending packets.

If `txq_inline_min` key is present the specified value (may be aligned by the driver in order not to exceed the limits and provide better descriptor space utilization) will be used by the driver and it is guaranteed that requested amount of data bytes are inlined into the WQE beside other inline settings. This key also may update `txq_inline_max` value (default or specified explicitly in devargs) to reserve the space for inline data.

If `txq_inline_min` key is not present, the value may be queried by the driver from the NIC via DevX if this feature is available. If there is no DevX enabled/supported the value 18 (supposing L2 header including VLAN) is set for ConnectX-4 and ConnectX-4 Lx, and 0 is set by default for ConnectX-5 and newer NICs. If packet is shorter the `txq_inline_min` value, the entire packet is inlined.

For ConnectX-4 NIC, driver does not allow specifying value below 18 (minimal L2 header, including VLAN), error will be raised.

For ConnectX-4 Lx NIC, it is allowed to specify values below 18, but it is not recommended and may prevent NIC from sending packets over some configurations.

Please, note, this minimal data inlining disengages eMPW feature (Enhanced Multi-Packet Write), because last one does not support partial packet inlining. This is not very critical due to minimal data inlining is mostly required by ConnectX-4 and ConnectX-4 Lx, these NICs do not support eMPW feature.

- `txq_inline_max` parameter [int]

Specifies the maximal packet length to be completely inlined into WQE Ethernet Segment for ordinary SEND method. If packet is larger than specified value, the packet data won't be copied by the driver at all, data buffer is addressed with a pointer. If packet length is less or equal all packet data will be copied into WQE. This may improve PCI bandwidth utilization for short packets significantly but requires the extra CPU cycles.

The data inline feature is controlled by number of Tx queues, if number of Tx queues is larger than `txqs_min_inline` key parameter, the inline feature is engaged, if there are not enough Tx queues (which means not enough CPU cores and CPU resources are scarce), data inline is not performed by the driver. Assigning `txqs_min_inline` with zero always enables the data inline.

The default `txq_inline_max` value is 290. The specified value may be adjusted by the driver in order not to exceed the limit (930 bytes) and to provide better WQE space filling without gaps, the adjustment is reflected in the debug log. Also, the default value (290) may be decreased in run-time if the large transmit queue size is requested and hardware does not support enough descriptor amount, in this case warning is emitted. If `txq_inline_max` key is specified and requested inline settings can not be satisfied then error will be raised.

- `txq_inline_mpw` parameter [int]

Specifies the maximal packet length to be completely inlined into WQE for Enhanced MPW method. If packet is large the specified value, the packet data won't be copied, and data buffer is addressed with pointer. If packet length is less or equal, all packet data will be copied into WQE. This may improve PCI bandwidth utilization for short packets significantly but requires the extra CPU cycles.

The data inline feature is controlled by number of TX queues, if number of Tx queues is larger than `txqs_min_inline` key parameter, the inline feature is engaged, if there are not enough Tx queues (which means not enough CPU cores and CPU resources are scarce), data inline is not performed by the driver. Assigning `txqs_min_inline` with zero always enables the data inline.

The default `txq_inline_mpw` value is 268. The specified value may be adjusted by the driver in order not to exceed the limit (930 bytes) and to provide better WQE space filling without gaps, the adjustment is reflected in the debug log. Due to multiple packets may be included to the same WQE with Enhanced Multi Packet Write Method and overall WQE size is limited it is not recommended to specify large values for the `txq_inline_mpw`. Also, the default value (268) may be decreased in run-time if the large transmit queue size is requested and hardware does not support enough descriptor amount, in this case warning is emitted. If `txq_inline_mpw` key is specified and requested inline settings can not be satisfied then error will be raised.

- `txqs_max_vec` parameter [int]

Enable vectorized Tx only when the number of TX queues is less than or equal to this value. This parameter is deprecated and ignored, kept for compatibility issue to not prevent driver from probing.

- `txq_mpw_hdr_dseg_en` parameter [int]

A nonzero value enables including two pointers in the first block of TX descriptor. The parameter is deprecated and ignored, kept for compatibility issue.

- `txq_max_inline_len` parameter [int]

Maximum size of packet to be inlined. This limits the size of packet to be inlined. If the size of a packet is larger than configured value, the packet isn't inlined even though there's enough space remained in the descriptor. Instead, the packet is included with pointer. This parameter is deprecated and converted directly to `txq_inline_mpw` providing full compatibility. Valid only if eMPW feature is engaged.

- `txq_mpw_en` parameter [int]

A nonzero value enables Enhanced Multi-Packet Write (eMPW) for ConnectX-5, ConnectX-6, ConnectX-6 Dx and BlueField. eMPW allows the TX burst function to pack up multiple packets in a single descriptor session in order to save PCI bandwidth and improve performance at the cost of a slightly higher CPU usage. When `txq_inline_mpw` is set along with `txq_mpw_en`, TX burst function copies entire packet data on to TX descriptor instead of including pointer of packet.

The Enhanced Multi-Packet Write feature is enabled by default if NIC supports it, can be disabled by explicit specifying 0 value for `txq_mpw_en` option. Also, if minimal data inlining is requested by non-zero `txq_inline_min` option or reported by the NIC, the eMPW feature is disengaged.

- `tx_db_nc` parameter [int]

The rdma core library can map doorbell register in two ways, depending on the environment variable "MLX5_SHUT_UP_BF":

- As regular cached memory (usually with write combining attribute), if the variable is either missing or set to zero.
- As non-cached memory, if the variable is present and set to not "0" value.

The type of mapping may slightly affect the Tx performance, the optimal choice is strongly relied on the host architecture and should be deduced practically.

If `tx_db_nc` is set to zero, the doorbell is forced to be mapped to regular memory (with write combining), the PMD will perform the extra write memory barrier after writing to doorbell, it might increase the needed CPU clocks per packet to send, but latency might be improved.

If `tx_db_nc` is set to one, the doorbell is forced to be mapped to non cached memory, the PMD will not perform the extra write memory barrier after writing to doorbell, on some architectures it might improve the performance.

If `tx_db_nc` is set to two, the doorbell is forced to be mapped to regular memory, the PMD will use heuristics to decide whether write memory barrier should be performed. For bursts with size multiple of recommended one (64 pkts) it is supposed the next burst is coming and no need to issue the extra memory barrier (it is supposed to be issued in the next coming burst, at least after descriptor writing). It might increase latency (on some hosts till next packets transmit) and should be used with care.

If `tx_db_nc` is omitted or set to zero, the preset (if any) environment variable “`MLX5_SHUT_UP_BF`” value is used. If there is no “`MLX5_SHUT_UP_BF`”, the default `tx_db_nc` value is zero for ARM64 hosts and one for others.

- `tx_vec_en` parameter [int]

A nonzero value enables Tx vector on ConnectX-5, ConnectX-6, ConnectX-6 Dx and BlueField NICs if the number of global Tx queues on the port is less than `txqs_max_vec`. The parameter is deprecated and ignored.

- `rx_vec_en` parameter [int]

A nonzero value enables Rx vector if the port is not configured in multi-segment otherwise this parameter is ignored.

Enabled by default.

- `vf_nl_en` parameter [int]

A nonzero value enables Netlink requests from the VF to add/remove MAC addresses or/and enable/disable promiscuous/all multicast on the Netdevice. Otherwise the relevant configuration must be run with Linux `iproute2` tools. This is a prerequisite to receive this kind of traffic.

Enabled by default, valid only on VF devices ignored otherwise.

- `l3_vxlan_en` parameter [int]

A nonzero value allows L3 VXLAN and VXLAN-GPE flow creation. To enable L3 VXLAN or VXLAN-GPE, users has to configure firmware and enable this parameter. This is a prerequisite to receive this kind of traffic.

Disabled by default.

- `dv_xmeta_en` parameter [int]

A nonzero value enables extensive flow metadata support if device is capable and driver supports it. This can enable extensive support of MARK and META item of `rte_flow`. The newly introduced `SET_TAG` and `SET_META` actions do not depend on `dv_xmeta_en`.

There are some possible configurations, depending on parameter value:

- 0, this is default value, defines the legacy mode, the MARK and META related actions and items operate only within NIC Tx and NIC Rx steering domains, no MARK and META information crosses the domain boundaries. The MARK item is 24 bits wide, the META item is 32 bits wide and match supported on egress only.

- 1, this engages extensive metadata mode, the MARK and META related actions and items operate within all supported steering domains, including FDB, MARK and META information may cross the domain boundaries. The MARK item is 24 bits wide, the META item width depends on kernel and firmware configurations and might be 0, 16 or 32 bits. Within NIC Tx domain META data width is 32 bits for compatibility, the actual width of data transferred to the FDB domain depends on kernel configuration and may be vary. The actual supported width can be retrieved in runtime by series of `rte_flow_validate()` trials.
- 2, this engages extensive metadata mode, the MARK and META related actions and items operate within all supported steering domains, including FDB, MARK and META information may cross the domain boundaries. The META item is 32 bits wide, the MARK item width depends on kernel and firmware configurations and might be 0, 16 or 24 bits. The actual supported width can be retrieved in runtime by series of `rte_flow_validate()` trials.

Mode	MARK	META	META Tx	FDB/Through
0	24 bits	32 bits	32 bits	no
1	24 bits	vary 0-32	32 bits	yes
2	vary 0-32	32 bits	32 bits	yes

If there is no E-Switch configuration the `dv_xmeta_en` parameter is ignored and the device is configured to operate in legacy mode (0).

Disabled by default (set to 0).

The Direct Verbs/Rules (engaged with `dv_flow_en = 1`) supports all of the extensive metadata features. The legacy Verbs supports FLAG and MARK metadata actions over NIC Rx steering domain only.

- `dv_flow_en` parameter [int]

A nonzero value enables the DV flow steering assuming it is supported by the driver (RDMA Core library version is `rdma-core-24.0` or higher).

Enabled by default if supported.

- `dv_esw_en` parameter [int]

A nonzero value enables E-Switch using Direct Rules.

Enabled by default if supported.

- `mr_ext_memseg_en` parameter [int]

A nonzero value enables extending memseg when registering DMA memory. If enabled, the number of entries in MR (Memory Region) lookup table on datapath is minimized and it benefits performance. On the other hand, it worsens memory utilization because registered memory is pinned by kernel driver. Even if a page in the extended chunk is freed, that doesn't become reusable until the entire memory is freed.

Enabled by default.

- `representor` parameter [list]

This parameter can be used to instantiate DPDK Ethernet devices from existing port (or VF) representors configured on the device.

It is a standard parameter whose format is described in `ether-net_device_standard_device_arguments`.

For instance, to probe port representors 0 through 2:

```
representor=[0-2]
```

- `max_dump_files_num` parameter [int]

The maximum number of files per PMD entity that may be created for debug information. The files will be created in `/var/log` directory or in current directory.

set to 128 by default.

- `lro_timeout_usec` parameter [int]

The maximum allowed duration of an LRO session, in micro-seconds. PMD will set the nearest value supported by HW, which is not bigger than the input `lro_timeout_usec` value. If this parameter is not specified, by default PMD will set the smallest value supported by HW.

33.5.4 Firmware configuration

Firmware features can be configured as key/value pairs.

The command to set a value is:

```
mlxconfig -d <device> set <key>=<value>
```

The command to query a value is:

```
mlxconfig -d <device> query | grep <key>
```

The device name for the command `mlxconfig` can be either the PCI address, or the mst device name found with:

```
mst status
```

Below are some firmware configurations listed.

- link type:

```
LINK_TYPE_P1
LINK_TYPE_P2
value: 1=Infiniband 2=Ethernet 3=VPI (auto-sense)
```

- enable SR-IOV:

```
SRIOV_EN=1
```

- maximum number of SR-IOV virtual functions:

```
NUM_OF_VFS=<max>
```

- enable DevX (required by Direct Rules and other features):

```
UCTX_EN=1
```

- aggressive CQE zipping:

```
CQE_COMPRESSION=1
```

- L3 VXLAN and VXLAN-GPE destination UDP port:

```
IP_OVER_VXLAN_EN=1
IP_OVER_VXLAN_PORT=<udp dport>
```

- enable IP-in-IP tunnel flow matching:

```
FLEX_PARSER_PROFILE_ENABLE=0
```

- enable MPLS flow matching:

```
FLEX_PARSER_PROFILE_ENABLE=1
```

- enable ICMP/ICMP6 code/type fields matching:

```
FLEX_PARSER_PROFILE_ENABLE=2
```

- enable Geneve flow matching:

```
FLEX_PARSER_PROFILE_ENABLE=0
```

33.6 Prerequisites

This driver relies on external libraries and kernel drivers for resources allocations and initialization. The following dependencies are not part of DPDK and must be installed separately:

- **libibverbs**

User space Verbs framework used by `librte_pmd_mlx5`. This library provides a generic interface between the kernel and low-level user space drivers such as `libmlx5`.

It allows slow and privileged operations (context initialization, hardware resources allocations) to be managed by the kernel and fast operations to never leave user space.

- **libmlx5**

Low-level user space driver library for Mellanox ConnectX-4/ConnectX-5/ConnectX-6/BlueField devices, it is automatically loaded by `libibverbs`.

This library basically implements send/receive calls to the hardware queues.

- **Kernel modules**

They provide the kernel-side Verbs API and low level device drivers that manage actual hardware initialization and resources sharing with user space processes.

Unlike most other PMDs, these modules must remain loaded and bound to their devices:

- `mlx5_core`: hardware driver managing Mellanox ConnectX-4/ConnectX-5/ConnectX-6/BlueField devices and related Ethernet kernel network devices.
- `mlx5_ib`: InfiniBand device driver.
- `ib_uverbs`: user space driver for Verbs (entry point for `libibverbs`).

- **Firmware update**

Mellanox OFED/EN releases include firmware updates for ConnectX-4/ConnectX-5/ConnectX-6/BlueField adapters.

Because each release provides new features, these updates must be applied to match the kernel modules and libraries they come with.

Note: Both libraries are BSD and GPL licensed. Linux kernel modules are GPL licensed.

33.6.1 Installation

Either RDMA Core library with a recent enough Linux kernel release (recommended) or Mellanox OFED/EN, which provides compatibility with older releases.

RDMA Core with Linux Kernel

- Minimal kernel version : v4.14 or the most recent 4.14-rc (see [Linux installation documentation](#))
- Minimal rdma-core version: v15+ commit 0c5f5765213a (“Merge pull request #227 from yishaih/tm”) (see [RDMA Core installation documentation](#))
- When building for i686 use:
 - rdma-core version 18.0 or above built with 32bit support.
 - Kernel version 4.14.41 or above.
- Starting with rdma-core v21, static libraries can be built:

```
cd build
CFLAGS=-fPIC cmake -DIN_PLACE=1 -DENABLE_STATIC=1 -GNinja ..
ninja
```

If rdma-core libraries are built but not installed, DPDK makefile can link them, thanks to these environment variables:

- EXTRA_CFLAGS=-I/path/to/rdma-core/build/include
- EXTRA_LDFLAGS=-L/path/to/rdma-core/build/lib
- PKG_CONFIG_PATH=/path/to/rdma-core/build/lib/pkgconfig

Mellanox OFED/EN

- Mellanox OFED version: ** 4.5, 4.6** / Mellanox EN version: **4.5, 4.6**
- firmware version:
 - ConnectX-4: **12.21.1000** and above.
 - ConnectX-4 Lx: **14.21.1000** and above.
 - ConnectX-5: **16.21.1000** and above.
 - ConnectX-5 Ex: **16.21.1000** and above.
 - ConnectX-6: **20.99.5374** and above.
 - ConnectX-6 Dx: **22.27.0090** and above.
 - BlueField: **18.25.1010** and above.

While these libraries and kernel modules are available on OpenFabrics Alliance’s [website](#) and provided by package managers on most distributions, this PMD requires Ethernet extensions that may not be supported at the moment (this is a work in progress).

[Mellanox OFED](#) and [Mellanox EN](#) include the necessary support and should be used in the meantime. For DPDK, only libibverbs, libmlx5, mlnx-ofed-kernel packages and firmware updates are required from that distribution.

Note: Several versions of Mellanox OFED/EN are available. Installing the version this DPDK release was developed and tested against is strongly recommended. Please check the [prerequisites](#).

33.7 Supported NICs

The following Mellanox device families are supported by the same mlx5 driver:

- ConnectX-4
- ConnectX-4 Lx
- ConnectX-5
- ConnectX-5 Ex
- ConnectX-6
- ConnectX-6 Dx
- BlueField

Below are detailed device names:

- Mellanox® ConnectX®-4 10G MCX4111A-XCAT (1x10G)
- Mellanox® ConnectX®-4 10G MCX412A-XCAT (2x10G)
- Mellanox® ConnectX®-4 25G MCX4111A-ACAT (1x25G)
- Mellanox® ConnectX®-4 25G MCX412A-ACAT (2x25G)
- Mellanox® ConnectX®-4 40G MCX413A-BCAT (1x40G)
- Mellanox® ConnectX®-4 40G MCX4131A-BCAT (1x40G)
- Mellanox® ConnectX®-4 40G MCX415A-BCAT (1x40G)
- Mellanox® ConnectX®-4 50G MCX413A-GCAT (1x50G)
- Mellanox® ConnectX®-4 50G MCX4131A-GCAT (1x50G)
- Mellanox® ConnectX®-4 50G MCX414A-BCAT (2x50G)
- Mellanox® ConnectX®-4 50G MCX415A-GCAT (1x50G)
- Mellanox® ConnectX®-4 50G MCX416A-BCAT (2x50G)
- Mellanox® ConnectX®-4 50G MCX416A-GCAT (2x50G)
- Mellanox® ConnectX®-4 50G MCX415A-CCAT (1x100G)
- Mellanox® ConnectX®-4 100G MCX416A-CCAT (2x100G)
- Mellanox® ConnectX®-4 Lx 10G MCX4111A-XCAT (1x10G)
- Mellanox® ConnectX®-4 Lx 10G MCX4121A-XCAT (2x10G)
- Mellanox® ConnectX®-4 Lx 25G MCX4111A-ACAT (1x25G)
- Mellanox® ConnectX®-4 Lx 25G MCX4121A-ACAT (2x25G)
- Mellanox® ConnectX®-4 Lx 40G MCX4131A-BCAT (1x40G)
- Mellanox® ConnectX®-5 100G MCX556A-ECAT (2x100G)
- Mellanox® ConnectX®-5 Ex EN 100G MCX516A-CDAT (2x100G)
- Mellanox® ConnectX®-6 200G MCX654106A-HCAT (2x200G)
- Mellanox® ConnectX®-6 Dx EN 100G MCX623106AN-CDAT (2x100G)

- Mellanox® ConnectX®-6 Dx EN 200G MCX623105AN-VDAT (1x200G)

33.8 Quick Start Guide on OFED/EN

1. Download latest Mellanox OFED/EN. For more info check the [prerequisites](#).
2. Install the required libraries and kernel modules either by installing only the required set, or by installing the entire Mellanox OFED/EN:

```
./mlnxofedinstall --upstream-libs --dpdk
```

3. Verify the firmware is the correct one:

```
ibv_devinfo
```

4. Verify all ports links are set to Ethernet:

```
mlxconfig -d <mst device> query | grep LINK_TYPE
LINK_TYPE_P1          ETH (2)
LINK_TYPE_P2          ETH (2)
```

Link types may have to be configured to Ethernet:

```
mlxconfig -d <mst device> set LINK_TYPE_P1/2=1/2/3

* LINK_TYPE_P1=<1|2|3> , 1=Infiniband 2=Ethernet 3=VPI (auto-sense)
```

For hypervisors, verify SR-IOV is enabled on the NIC:

```
mlxconfig -d <mst device> query | grep SRIOV_EN
SRIOV_EN          True (1)
```

If needed, configure SR-IOV:

```
mlxconfig -d <mst device> set SRIOV_EN=1 NUM_OF_VFS=16
mlxfwreset -d <mst device> reset
```

5. Restart the driver:

```
/etc/init.d/openibd restart
```

or:

```
service openibd restart
```

If link type was changed, firmware must be reset as well:

```
mlxfwreset -d <mst device> reset
```

For hypervisors, after reset write the sysfs number of virtual functions needed for the PF.

To dynamically instantiate a given number of virtual functions (VFs):

```
echo [num_vfs] > /sys/class/infiniband/mlx5_0/device/sriov_numvfs
```

6. Compile DPDK and you are ready to go. See instructions on Development Kit Build System

33.9 Enable switchdev mode

Switchdev mode is a mode in E-Switch, that binds between representor and VF. Representor is a port in DPDK that is connected to a VF in such a way that assuming there are no offload flows, each packet that is sent from the VF will be received by the corresponding representor. While each packet that is sent to a representor will be received by the VF. This is very useful in case of SRIOV mode, where the first

packet that is sent by the VF will be received by the DPDK application which will decide if this flow should be offloaded to the E-Switch. After offloading the flow packet that the VF that are matching the flow will not be received any more by the DPDK application.

1. Enable SRIOV mode:

```
mlxconfig -d <mst device> set SRIOV_EN=true
```

2. Configure the max number of VFs:

```
mlxconfig -d <mst device> set NUM_OF_VFS=<num of vfs>
```

3. Reset the FW:

```
mlxfwreset -d <mst device> reset
```

3. Configure the actual number of VFs:

```
echo <num of vfs > /sys/class/net/<net device>/device/sriov_numvfs
```

4. Unbind the device (can be rebind after the switchdev mode):

```
echo -n "<device pci address" > /sys/bus/pci/drivers/mlx5_core/unbind
```

5. Enable switchdev mode:

```
echo switchdev > /sys/class/net/<net device>/compat/devlink/mode
```

33.10 Performance tuning

1. Configure aggressive CQE Zipping for maximum performance:

```
mlxconfig -d <mst device> s CQE_COMPRESSION=1
```

To set it back to the default CQE Zipping mode use:

```
mlxconfig -d <mst device> s CQE_COMPRESSION=0
```

2. In case of virtualization:

- Make sure that hypervisor kernel is 3.16 or newer.
- Configure boot with `iommu=pt`.
- Use 1G huge pages.
- Make sure to allocate a VM on huge pages.
- Make sure to set CPU pinning.

3. Use the CPU near local NUMA node to which the PCIe adapter is connected, for better performance. For VMs, verify that the right CPU and NUMA node are pinned according to the above. Run:

```
lstopo-no-graphics
```

to identify the NUMA node to which the PCIe adapter is connected.

4. If more than one adapter is used, and root complex capabilities allow to put both adapters on the same NUMA node without PCI bandwidth degradation, it is recommended to locate both adapters on the same NUMA node. This in order to forward packets from one to the other without NUMA performance penalty.
5. Disable pause frames:

```
ethtool -A <netdev> rx off tx off
```

6. Verify IO non-posted prefetch is disabled by default. This can be checked via the BIOS configuration. Please contact your server provider for more information about the settings.

Note: On some machines, depends on the machine integrator, it is beneficial to set the PCI max read request parameter to 1K. This can be done in the following way:

To query the read request size use:

```
setpci -s <NIC PCI address> 68.w
```

If the output is different than 3XXX, set it by:

```
setpci -s <NIC PCI address> 68.w=3XXX
```

The XXX can be different on different systems. Make sure to configure according to the setpci output.

7. To minimize overhead of searching Memory Regions:

- ‘-socket-mem’ is recommended to pin memory by predictable amount.
- Configure per-core cache when creating Mempools for packet buffer.
- Refrain from dynamically allocating/freeing memory in run-time.

33.11 Supported hardware offloads

Table 33.1: Minimal SW/HW versions for queue offloads

Offload	DPDK	Linux	rdma-core	OFED	firmware	hardware
common base	17.11	4.14	16	4.2-1	12.21.1000	ConnectX-4
checksums	17.11	4.14	16	4.2-1	12.21.1000	ConnectX-4
Rx timestamp	17.11	4.14	16	4.2-1	12.21.1000	ConnectX-4
TSO	17.11	4.14	16	4.2-1	12.21.1000	ConnectX-4
LRO	19.08	N/A	N/A	4.6-4	16.25.6406	ConnectX-5

Table 33.2: Minimal SW/HW versions for rte_flow offloads

Offload	with E-Switch	with NIC
Count	DPDK 19.05 OFED 4.6 rdma-core 24 ConnectX-5	DPDK 19.02 OFED 4.6 rdma-core 23 ConnectX-5
Drop	DPDK 19.05 OFED 4.6 rdma-core 24 ConnectX-5	DPDK 18.11 OFED 4.5 rdma-core 23 ConnectX-4
Queue / RSS	N/A	DPDK 18.11 OFED 4.5 rdma-core 23 ConnectX-4
Encapsulation (VXLAN / NVGRE / RAW)	DPDK 19.05 OFED 4.7-1 rdma-core 24 ConnectX-5	DPDK 19.02 OFED 4.6 rdma-core 23 ConnectX-5
Encapsulation GENEVE	DPDK 19.11 OFED 4.7-3 rdma-core 27 ConnectX-5	DPDK 19.11 OFED 4.7-3 rdma-core 27 ConnectX-5
Header rewrite (set_ipv4_src / set_ipv4_dst / set_ipv6_src / set_ipv6_dst / set_tp_src / set_tp_dst / dec_ttl / set_ttl / set_mac_src / set_mac_dst)	DPDK 19.05 OFED 4.7-1 rdma-core 24 ConnectX-5	DPDK 19.02 OFED 4.7-1 rdma-core 24 ConnectX-5
33.11. Supported hardware offloads (of_set_vlan_vid)	DPDK 19.11 OFED 4.7-1 ConnectX-5	DPDK 19.11 OFED 4.7-1 ConnectX-5

33.12 Notes for metadata

MARK and META items are interrelated with datapath - they might move from/to the applications in mbuf fields. Hence, zero value for these items has the special meaning - it means “no metadata are provided”, not zero values are treated by applications and PMD as valid ones.

Moreover in the flow engine domain the value zero is acceptable to match and set, and we should allow to specify zero values as `rte_flow` parameters for the META and MARK items and actions. In the same time zero mask has no meaning and should be rejected on validation stage.

33.13 Notes for testpmd

Compared to `librte_pmd_mlx4` that implements a single RSS configuration per port, `librte_pmd_mlx5` supports per-protocol RSS configuration.

Since `testpmd` defaults to IP RSS mode and there is currently no command-line parameter to enable additional protocols (UDP and TCP as well as IP), the following commands must be entered from its CLI to get the same behavior as `librte_pmd_mlx4`:

```
> port stop all
> port config all rss all
> port start all
```

33.14 Usage example

This section demonstrates how to launch **testpmd** with Mellanox ConnectX-4/ConnectX-5/ConnectX-6/BlueField devices managed by `librte_pmd_mlx5`.

1. Load the kernel modules:

```
modprobe -a ib_uverbs mlx5_core mlx5_ib
```

Alternatively if `MLNX_OFED/MLNX_EN` is fully installed, the following script can be run:

```
/etc/init.d/openibd restart
```

Note: User space I/O kernel modules (`uio` and `igb_uio`) are not used and do not have to be loaded.

2. Make sure Ethernet interfaces are in working order and linked to kernel verbs. Related `sysfs` entries should be present:

```
ls -d /sys/class/net/*/device/infiniband_verbs/uverbs* | cut -d / -f 5
```

Example output:

```
eth30
eth31
eth32
eth33
```

3. Optionally, retrieve their PCI bus addresses for whitelisting:

```
{
    for intf in eth2 eth3 eth4 eth5;
    do
        (cd "/sys/class/net/${intf}/device/" && pwd -P);
    done
}
```

```
done;
} |
sed -n 's,.*\/(.*)\,-w \1,p'
```

Example output:

```
-w 0000:05:00.1
-w 0000:06:00.0
-w 0000:06:00.1
-w 0000:05:00.0
```

4. Request huge pages:

```
echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages/nr_hugepages
```

5. Start testpmd with basic parameters:

```
testpmd -l 8-15 -n 4 -w 05:00.0 -w 05:00.1 -w 06:00.0 -w 06:00.1 -- --rxq=2 --txq=2 -i
```

Example output:

```
[...]
EAL: PCI device 0000:05:00.0 on NUMA socket 0
EAL: probe driver: 15b3:1013 librte_pmd_mlx5
PMD: librte_pmd_mlx5: PCI information matches, using device "mlx5_0" (VF: false)
PMD: librte_pmd_mlx5: 1 port(s) detected
PMD: librte_pmd_mlx5: port 1 MAC address is e4:1d:2d:e7:0c:fe
EAL: PCI device 0000:05:00.1 on NUMA socket 0
EAL: probe driver: 15b3:1013 librte_pmd_mlx5
PMD: librte_pmd_mlx5: PCI information matches, using device "mlx5_1" (VF: false)
PMD: librte_pmd_mlx5: 1 port(s) detected
PMD: librte_pmd_mlx5: port 1 MAC address is e4:1d:2d:e7:0c:ff
EAL: PCI device 0000:06:00.0 on NUMA socket 0
EAL: probe driver: 15b3:1013 librte_pmd_mlx5
PMD: librte_pmd_mlx5: PCI information matches, using device "mlx5_2" (VF: false)
PMD: librte_pmd_mlx5: 1 port(s) detected
PMD: librte_pmd_mlx5: port 1 MAC address is e4:1d:2d:e7:0c:fa
EAL: PCI device 0000:06:00.1 on NUMA socket 0
EAL: probe driver: 15b3:1013 librte_pmd_mlx5
PMD: librte_pmd_mlx5: PCI information matches, using device "mlx5_3" (VF: false)
PMD: librte_pmd_mlx5: 1 port(s) detected
PMD: librte_pmd_mlx5: port 1 MAC address is e4:1d:2d:e7:0c:fb
Interactive-mode selected
Configuring Port 0 (socket 0)
PMD: librte_pmd_mlx5: 0x8cba80: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx5: 0x8cba80: RX queues number update: 0 -> 2
Port 0: E4:1D:2D:E7:0C:FE
Configuring Port 1 (socket 0)
PMD: librte_pmd_mlx5: 0x8ccac8: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx5: 0x8ccac8: RX queues number update: 0 -> 2
Port 1: E4:1D:2D:E7:0C:FF
Configuring Port 2 (socket 0)
PMD: librte_pmd_mlx5: 0x8cdb10: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx5: 0x8cdb10: RX queues number update: 0 -> 2
Port 2: E4:1D:2D:E7:0C:FA
Configuring Port 3 (socket 0)
PMD: librte_pmd_mlx5: 0x8ceb58: TX queues number update: 0 -> 2
PMD: librte_pmd_mlx5: 0x8ceb58: RX queues number update: 0 -> 2
Port 3: E4:1D:2D:E7:0C:FB
Checking link statuses...
Port 0 Link Up - speed 40000 Mbps - full-duplex
Port 1 Link Up - speed 40000 Mbps - full-duplex
Port 2 Link Up - speed 10000 Mbps - full-duplex
Port 3 Link Up - speed 10000 Mbps - full-duplex
Done
```

testpmd>

MVNETA POLL MODE DRIVER

The MVNETA PMD (librte_pmd_mvnet) provides poll mode driver support for the Marvell NETA 1/2.5 Gbps adapter.

Detailed information about SoCs that use PPv2 can be obtained here:

- <https://www.marvell.com/embedded-processors/armada-3700/>

Note: Due to external dependencies, this driver is disabled by default. It must be enabled manually by setting relevant configuration option manually. Please refer to *Config File Options* section for further details.

34.1 Features

Features of the MVNETA PMD are:

- Start/stop
- tx/rx_queue_setup
- tx/rx_burst
- Speed capabilities
- Jumbo frame
- MTU update
- Promiscuous mode
- Unicast MAC filter
- Link status
- CRC offload
- L3 checksum offload
- L4 checksum offload
- Packet type parsing
- Basic stats

34.2 Limitations

- Flushing vlans added for filtering is not possible due to MUSDK missing functionality. Current workaround is to reset board so that NETA has a chance to start in a sane state.

34.3 Prerequisites

- Custom Linux Kernel sources

```
git clone https://github.com/MarvellEmbeddedProcessors/linux-marvell.git -b linux-4.4.120-
```

- MUSDK (Marvell User-Space SDK) sources

```
git clone https://github.com/MarvellEmbeddedProcessors/musdk-marvell.git -b musdk-armada-1
```

MUSDK is a light-weight library that provides direct access to Marvell's NETA. Alternatively pre-built MUSDK library can be requested from [Marvell Extranet](#). Once approval has been granted, library can be found by typing `musdk` in the search box.

MUSDK must be configured with the following features:

```
--enable-pp2=no --enable-neta
```

- DPDK environment

Follow the DPDK Getting Started Guide for Linux to setup DPDK environment.

34.4 Pre-Installation Configuration

34.4.1 Config File Options

The following options can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_MVNETA_PMD` (default `n`)

Toggle compilation of the `librte_pmd_mvnet` driver.

34.4.2 Runtime options

The following `devargs` options can be enabled at runtime. They must be passed as part of EAL arguments.

- `iface` (mandatory, with no default value)

The name of port (owned by MUSDK) that should be enabled in DPDK. This options can be repeated resulting in a list of ports to be enabled. For instance below will enable `eth0` and `eth1` ports.

```
./testpmd --vdev=net_mvnet,iface=eth0,iface=eth1 \  
-c 3 -- -i --p 3 -a
```

34.5 Building DPDK

Driver needs precompiled MUSDK library during compilation.

```
export CROSS_COMPILE=<toolchain>/bin/aarch64-linux-gnu-
./bootstrap
./configure --host=aarch64-linux-gnu --enable-pp2=no --enable-neta
make install
```

MUSDK will be installed to *usr/local* under current directory. For the detailed build instructions please consult `doc/musdk_get_started.txt`.

Before the DPDK build process the environmental variable `LIBMUSDK_PATH` with the path to the MUSDK installation directory needs to be exported.

```
export LIBMUSDK_PATH=<musdk>/usr/local
export CROSS=aarch64-linux-gnu-
make config T=arm64-armv8a-linux-gcc
sed -ri 's, (MVNETA_PMD=)n,\ly, ' build/.config
make
```

34.6 Usage Example

MVNETA PMD requires extra out of tree kernel modules to function properly. *musdk_uio* and *mv_neta_uio* sources are part of the MUSDK. Please consult `doc/musdk_get_started.txt` for the detailed build instructions.

```
insmod musdk_uio.ko
insmod mv_neta_uio.ko
```

Additionally interfaces used by DPDK application need to be put up:

```
ip link set eth0 up
ip link set eth1 up
```

In order to run `testpmd` example application following command can be used:

```
./testpmd --vdev=net_mvneta,iface=eth0,iface=eth1 -c 3 -- \
-i --p 3 -a --txd 256 --rxq 128 --rxq=1 --txq=1 --nb-cores=1
```

In order to run `l2fwd` example application following command can be used:

```
./l2fwd --vdev=net_mvneta,iface=eth0,iface=eth1 -c 3 -- -T 1 -p 3
```

MVPP2 POLL MODE DRIVER

The MVPP2 PMD (`librte_pmd_mvpp2`) provides poll mode driver support for the Marvell PPv2 (Packet Processor v2) 1/10 Gbps adapter.

Detailed information about SoCs that use PPv2 can be obtained here:

- <https://www.marvell.com/embedded-processors/armada-70xx/>
- <https://www.marvell.com/embedded-processors/armada-80xx/>

Note: Due to external dependencies, this driver is disabled by default. It must be enabled manually by setting relevant configuration option manually. Please refer to *Config File Options* section for further details.

35.1 Features

Features of the MVPP2 PMD are:

- Speed capabilities
- Link status
- Tx Queue start/stop
- MTU update
- Jumbo frame
- Promiscuous mode
- Allmulticast mode
- Unicast MAC filter
- Multicast MAC filter
- RSS hash
- VLAN filter
- CRC offload
- L3 checksum offload
- L4 checksum offload

- Packet type parsing
- Basic stats
- *Extended stats*
- RX flow control
- Scattered TX frames
- *QoS*
- *Flow API*
- *Traffic metering and policing*
- *Traffic Management API*

35.2 Limitations

- Number of lcores is limited to 9 by MUSDK internal design. If more lcores need to be allocated, locking will have to be considered. Number of available lcores can be changed via `MRVL_MUSDK_HIFS_RESERVED` define in `mrvl_ethdev.c` source file.
- Flushing vlans added for filtering is not possible due to MUSDK missing functionality. Current workaround is to reset board so that PPv2 has a chance to start in a sane state.
- MUSDK architecture does not support changing configuration in run time. All necessary configurations should be done before first `dev_start()`.
- RX queue start/stop is not supported.
- Current implementation does not support replacement of buffers in the HW buffer pool at run time, so it is responsibility of the application to ensure that MTU does not exceed the configured buffer size.
- Configuring TX flow control currently is not supported.
- In current implementation, mechanism for acknowledging transmitted packets (`tx_done_cleanup`) is not supported.
- Running more than one DPDK-MUSDK application simultaneously is not supported.

35.3 Prerequisites

- Custom Linux Kernel sources

```
git clone https://github.com/MarvellEmbeddedProcessors/linux-marvell.git -b linux-4.4.120-
```

- Out of tree `mvpp2x_sysfs` kernel module sources

```
git clone https://github.com/MarvellEmbeddedProcessors/mvpp2x-marvell.git -b mvpp2x-armada-
```

- MUSDK (Marvell User-Space SDK) sources

```
git clone https://github.com/MarvellEmbeddedProcessors/musdk-marvell.git -b musdk-armada-
```

MUSDK is a light-weight library that provides direct access to Marvell's PPv2 (Packet Processor v2). Alternatively prebuilt MUSDK library can be requested from [Marvell Extranet](#). Once approval has been granted, library can be found by typing `musdk` in the search box.

To get better understanding of the library one can consult documentation available in the `doc` top level directory of the MUSDK sources.

- DPDK environment

Follow the DPDK Getting Started Guide for Linux to setup DPDK environment.

35.4 Config File Options

The following options can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_MVPP2_PMD` (default `n`)

Toggle compilation of the `librte mvpp2` driver.

Note: When MVPP2 PMD is enabled `CONFIG_RTE_LIBRTE_MVNETA_PMD` must be disabled

35.5 Building DPDK

Driver needs precompiled MUSDK library during compilation.

```
export CROSS_COMPILE=<toolchain>/bin/aarch64-linux-gnu-  
./bootstrap  
./configure --host=aarch64-linux-gnu  
make install
```

MUSDK will be installed to `usr/local` under current directory. For the detailed build instructions please consult `doc/musdk_get_started.txt`.

Before the DPDK build process the environmental variable `LIBMUSDK_PATH` with the path to the MUSDK installation directory needs to be exported.

For additional instructions regarding DPDK cross compilation please refer to Cross compile DPDK for ARM64.

```
export LIBMUSDK_PATH=<musdk>/usr/local  
export CROSS=<toolchain>/bin/aarch64-linux-gnu-  
export RTE_KERNELDIR=<kernel-dir>  
export RTE_TARGET=arm64-armv8a-linux-gcc  
  
make config T=arm64-armv8a-linux-gcc  
sed -i "s/MVNETA_PMD=y/MVNETA_PMD=n/" build/.config  
sed -i "s/MVPP2_PMD=n/MVPP2_PMD=y/" build/.config  
make
```

35.6 Usage Example

MVPP2 PMD requires extra out of tree kernel modules to function properly. `musdk_cma` sources are part of the MUSDK. Please consult `doc/musdk_get_started.txt` for the detailed build instructions. For `mvpp2x_sysfs` please consult `Documentation/pp22_sysfs.txt` for the detailed build instructions.

```
insmod musdk_cma.ko
insmod mvpp2x_sysfs.ko
```

Additionally interfaces used by DPDK application need to be put up:

```
ip link set eth0 up
ip link set eth2 up
```

In order to run testpmd example application following command can be used:

```
./testpmd --vdev=eth_mvpp2,iface=eth0,iface=eth2 -c 7 -- \
--burst=128 --txd=2048 --rxd=1024 --rxq=2 --txq=2 --nb-cores=2 \
-i -a --rss-udp
```

35.7 Extended stats

MVPP2 PMD supports the following extended statistics:

- `rx_bytes`: number of RX bytes
- `rx_packets`: number of RX packets
- `rx_unicast_packets`: number of RX unicast packets
- `rx_errors`: number of RX MAC errors
- `rx_fullq_dropped`: number of RX packets dropped due to full RX queue
- `rx_bm_dropped`: number of RX packets dropped due to no available buffers in the HW pool
- `rx_early_dropped`: number of RX packets that were early dropped
- `rx_fifo_dropped`: number of RX packets dropped due to RX fifo overrun
- `rx_cls_dropped`: number of RX packets dropped by classifier
- `tx_bytes`: number of TX bytes
- `tx_packets`: number of TX packets
- `tx_unicast_packets`: number of TX unicast packets
- `tx_errors`: number of TX MAC errors

35.8 QoS Configuration

QoS configuration is done through external configuration file. Path to the file must be given as *cfg* in driver's vdev parameter list.

35.8.1 Configuration syntax

```
[policer <policer_id>]
token_unit = <token_unit>
color = <color_mode>
cir = <cir>
ebs = <ebs>
cbs = <cbs>

[port <portnum> default]
```

```

default_tc = <default_tc>
mapping_priority = <mapping_priority>

rate_limit_enable = <rate_limit_enable>
rate_limit = <rate_limit>
burst_size = <burst_size>

default_policer = <policer_id>

[port <portnum> tc <traffic_class>]
rxq = <rx_queue_list>
pcp = <pcp_list>
dscp = <dscp_list>
default_color = <default_color>

[port <portnum> tc <traffic_class>]
rxq = <rx_queue_list>
pcp = <pcp_list>
dscp = <dscp_list>

[port <portnum> txq <txqnum>]
sched_mode = <sched_mode>
wrr_weight = <wrr_weight>

rate_limit_enable = <rate_limit_enable>
rate_limit = <rate_limit>
burst_size = <burst_size>

```

Where:

- <portnum>: DPDK Port number (0..n).
- <default_tc>: Default traffic class (e.g. 0)
- <mapping_priority>: QoS priority for mapping (*ip*, *vlan*, *ip/vlan* or *vlan/ip*).
- <traffic_class>: Traffic Class to be configured.
- <rx_queue_list>: List of DPDK RX queues (e.g. 0 1 3-4)
- <pcp_list>: List of PCP values to handle in particular TC (e.g. 0 1 3-4 7).
- <dscp_list>: List of DSCP values to handle in particular TC (e.g. 0-12 32-48 63).
- <default_policer>: Id of the policer configuration section to be used as default.
- <policer_id>: Id of the policer configuration section (0..31).
- <token_unit>: Policer token unit (*bytes* or *packets*).
- <color_mode>: Policer color mode (*aware* or *blind*).
- <cir>: Committed information rate in unit of kilo bits per second (data rate) or packets per second.
- <cbs>: Committed burst size in unit of kilo bytes or number of packets.
- <ebs>: Excess burst size in unit of kilo bytes or number of packets.
- <default_color>: Default color for specific tc.
- <rate_limit_enable>: Enables per port or per txq rate limiting (*0/1* to disable/enable).
- <rate_limit>: Committed information rate, in kilo bits per second.
- <burst_size>: Committed burst size, in kilo bytes.

- <sched_mode>: Egress scheduler mode (*wrr* or *sp*).
- <wrr_weight>: Txq weight.

Setting PCP/DSCP values for the default TC is not required. All PCP/DSCP values not assigned explicitly to particular TC will be handled by the default TC.

Configuration file example

```
[policer 0]
token_unit = bytes
color = blind
cir = 100000
ebs = 64
cbs = 64

[port 0 default]
default_tc = 0
mapping_priority = ip

rate_limit_enable = 1
rate_limit = 1000
burst_size = 2000

[port 0 tc 0]
rxq = 0 1

[port 0 txq 0]
sched_mode = wrr
wrr_weight = 10

[port 0 txq 1]
sched_mode = wrr
wrr_weight = 100

[port 0 txq 2]
sched_mode = sp

[port 0 tc 1]
rxq = 2
pcp = 5 6 7
dscp = 26-38

[port 1 default]
default_tc = 0
mapping_priority = vlan/ip

default_policer = 0

[port 1 tc 0]
rxq = 0
dscp = 10

[port 1 tc 1]
rxq = 1
dscp = 11-20

[port 1 tc 2]
rxq = 2
dscp = 30

[port 1 txq 0]
```



```
rate_limit_enable = 1
rate_limit = 10000
burst_size = 2000
```

Usage example

```
./testpmd --vdev=eth_mvpp2,iface=eth0,iface=eth2,cfg=/home/user/mrvl.conf \
-c 7 -- -i -a --disable-hw-vlan-strip --rxq=3 --txq=3
```

35.9 Flow API

PPv2 offers packet classification capabilities via classifier engine which can be configured via generic flow API offered by DPDK.

The `flow_isolated_mode` is supported.

For an additional description please refer to DPDK `../prog_guide/rte_flow`.

35.9.1 Supported flow actions

Following flow action items are supported by the driver:

- DROP
- QUEUE

35.9.2 Supported flow items

Following flow items and their respective fields are supported by the driver:

- ETH
 - source MAC
 - destination MAC
 - ethertype
- VLAN
 - PCP
 - VID
- IPV4
 - DSCP
 - protocol
 - source address
 - destination address
- IPV6
 - flow label

- next header
- source address
- destination address
- UDP
 - source port
 - destination port
- TCP
 - source port
 - destination port

35.9.3 Classifier match engine

Classifier has an internal match engine which can be configured to operate in either exact or maskable mode.

Mode is selected upon creation of the first unique flow rule as follows:

- maskable, if key size is up to 8 bytes.
- exact, otherwise, i.e for keys bigger than 8 bytes.

Where the key size equals the number of bytes of all fields specified in the flow items.

Table 35.1: Examples of key size calculation

Flow pattern	Key size in bytes	Used engine
ETH (destination MAC) / VLAN (VID)	$6 + 2 = 8$	Maskable
VLAN (VID) / IPV4 (source address)	$2 + 4 = 6$	Maskable
TCP (source port, destination port)	$2 + 2 = 4$	Maskable
VLAN (priority) / IPV4 (source address)	$1 + 4 = 5$	Maskable
IPV4 (destination address) / UDP (source port, destination port)	$6 + 2 + 2 = 10$	Exact
VLAN (VID) / IPV6 (flow label, destination address)	$2 + 3 + 16 = 21$	Exact
IPV4 (DSCP, source address, destination address)	$1 + 4 + 4 = 9$	Exact
IPV6 (flow label, source address, destination address)	$3 + 16 + 16 = 35$	Exact

From the user perspective maskable mode means that masks specified via flow rules are respected. In case of exact match mode, masks which do not provide exact matching (all bits masked) are ignored.

If the flow matches more than one classifier rule the first (with the lowest index) matched takes precedence.

35.9.4 Flow rules usage example

Before proceeding run testpmd user application:

```
./testpmd --vdev=eth_mvpp2,iface=eth0,iface=eth2 -c 3 -- -i --p 3 -a --disable-hw-vlan-strip
```

Example #1

```
testpmd> flow create 0 ingress pattern eth src is 10:11:12:13:14:15 / end actions drop / end
```

In this case key size is 6 bytes thus maskable type is selected. Testpmd will set mask to ff:ff:ff:ff:ff:ff i.e traffic explicitly matching above rule will be dropped.

Example #2

```
testpmd> flow create 0 ingress pattern ipv4 src spec 10.10.10.0 src mask 255.255.255.0 / tcp src port 16 / end actions drop / end
```

In this case key size is 8 bytes thus maskable type is selected. Flows which have IPv4 source addresses ranging from 10.10.10.0 to 10.10.10.255 and tcp source port set to 16 will be dropped.

Example #3

```
testpmd> flow create 0 ingress pattern vlan vid spec 0x10 vid mask 0x10 / ipv4 src spec 10.10.1.1 / end actions drop / end
```

In this case key size is 10 bytes thus exact type is selected. Even though each item has partial mask set, masks will be ignored. As a result only flows with VID set to 16 and IPv4 source and destination addresses set to 10.10.1.1 and 11.11.11.1 respectively will be dropped.

35.9.5 Limitations

Following limitations need to be taken into account while creating flow rules:

- For IPv4 exact match type the key size must be up to 12 bytes.
- For IPv6 exact match type the key size must be up to 36 bytes.
- Following fields cannot be partially masked (all masks are treated as if they were exact):
 - ETH: ethertype
 - VLAN: PCP, VID
 - IPv4: protocol
 - IPv6: next header
 - TCP/UDP: source port, destination port
- Only one classifier table can be created thus all rules in the table have to match table format. Table format is set during creation of the first unique flow rule.
- Up to 5 fields can be specified per flow rule.
- Up to 20 flow rules can be added.

For additional information about classifier please consult `doc/musdk_cls_user_guide.txt`.

35.10 Traffic metering and policing

MVPP2 PMD supports DPDK traffic metering and policing that allows the following:

1. Meter ingress traffic.

2. Do policing.
3. Gather statistics.

For an additional description please refer to DPDK Traffic Metering and Policing API.

The policer objects defined by this feature can work with the default policer defined via config file as described in [QoS Support](#).

35.10.1 Limitations

The following capabilities are not supported:

- MTR object meter DSCP table update
- MTR object policer action update
- MTR object enabled statistics

35.10.2 Usage example

1. Run testpmd user app:

```
./testpmd --vdev=eth_mvpp2,iface=eth0,iface=eth2 -c 6 -- -i -p 3 -a --txd 1024 --rxd 1024
```

2. Create meter profile:

```
testpmd> add port meter profile 0 0 srtcm_rfc2697 2000 256 256
```

3. Create meter:

```
testpmd> create port meter 0 0 0 yes d d d 0 1 0
```

4. Create flow rule witch meter attached:

```
testpmd> flow create 0 ingress pattern ipv4 src is 10.10.10.1 / end actions meter mtr_id 0
```

For a detailed usage description please refer to “Traffic Metering and Policing” section in DPDK Testpmd Runtime Functions.

35.11 Traffic Management API

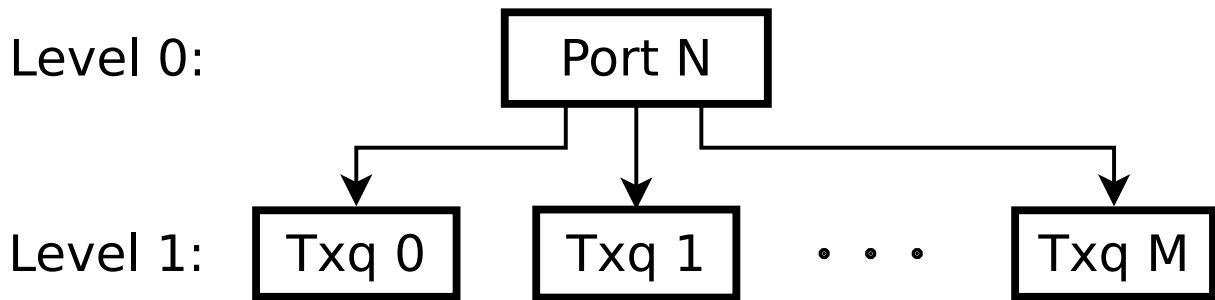
MVPP2 PMD supports generic DPDK Traffic Management API which allows to configure the following features:

1. Hierarchical scheduling
2. Traffic shaping
3. Congestion management
4. Packet marking

Internally TM is represented by a hierarchy (tree) of nodes. Node which has a parent is called a leaf whereas node without parent is called a non-leaf (root). MVPP2 PMD supports two level hierarchy where level 0 represents ports and level 1 represents tx queues of a given port.

Nodes hold following types of settings:

- for egress scheduler configuration: weight



- for egress rate limiter: private shaper
- bitmask indicating which statistics counters will be read

Hierarchy is always constructed from the top, i.e first a root node is added then some number of leaf nodes. Number of leaf nodes cannot exceed number of configured tx queues.

After hierarchy is complete it can be committed.

For an additional description please refer to DPDK Traffic Management API.

35.11.1 Limitations

The following capabilities are not supported:

- Traffic manager WRED profile and WRED context
- Traffic manager shared shaper update
- Traffic manager packet marking
- Maximum number of levels in hierarchy is 2
- Currently dynamic change of a hierarchy is not supported

35.11.2 Usage example

For a detailed usage description please refer to “Traffic Management” section in DPDK Testpmd Runtime Functions.

1. Run testpmd as follows:

```
./testpmd --vdev=net_mrvl,iface=eth0,iface=eth2,cfg=./qos_config -c 7 -- \
-i -p 3 --disable-hw-vlan-strip --rxq 3 --txq 3 --txd 1024 --rxd 1024
```

2. Stop all ports:

```
testpmd> port stop all
```

3. Add shaper profile:

```
testpmd> add port tm node shaper profile 0 0 900000 70000 0
```

Parameters have following meaning:

```

0          - Id of a port.
0          - Id of a new shaper profile.
900000     - Shaper rate in bytes/s.
70000      - Bucket size in bytes.
0          - Packet length adjustment - ignored.
```

4. Add non-leaf node for port 0:

```
testpmd> add port tm nonleaf node 0 3 -1 0 0 0 0 0 1 3 0
```

Parameters have following meaning:

- 0 - Id of a port
- 3 - Id of a new node.
- 1 - Indicate that root does not have a parent.
- 0 - Priority of the node.
- 0 - Weight of the node.
- 0 - Id of a level. Since this is a root 0 is passed.
- 0 - Id of the shaper profile.
- 0 - Number of SP priorities.
- 3 - Enable statistics for both number of transmitted packets and bytes.
- 0 - Number of shared shapers.

5. Add leaf node for tx queue 0:

```
testpmd> add port tm leaf node 0 0 3 0 30 1 -1 0 0 1 0
```

Parameters have following meaning:

- 0 - Id of a port.
- 0 - Id of a new node.
- 3 - Id of the parent node.
- 0 - Priority of a node.
- 30 - WRR weight.
- 1 - Id of a level. Since this is a leaf node 1 is passed.
- 1 - Id of a shaper. -1 indicates that shaper is not attached.
- 0 - Congestion management is not supported.
- 0 - Congestion management is not supported.
- 1 - Enable statistics counter for number of transmitted packets.
- 0 - Number of shared shapers.

6. Add leaf node for tx queue 1:

```
testpmd> add port tm leaf node 0 1 3 0 60 1 -1 0 0 1 0
```

Parameters have following meaning:

- 0 - Id of a port.
- 1 - Id of a new node.
- 3 - Id of the parent node.
- 0 - Priority of a node.
- 60 - WRR weight.
- 1 - Id of a level. Since this is a leaf node 1 is passed.
- 1 - Id of a shaper. -1 indicates that shaper is not attached.
- 0 - Congestion management is not supported.
- 0 - Congestion management is not supported.
- 1 - Enable statistics counter for number of transmitted packets.
- 0 - Number of shared shapers.

7. Add leaf node for tx queue 2:

```
testpmd> add port tm leaf node 0 2 3 0 99 1 -1 0 0 1 0
```

Parameters have following meaning:

- 0 - Id of a port.
- 2 - Id of a new node.
- 3 - Id of the parent node.
- 0 - Priority of a node.
- 99 - WRR weight.
- 1 - Id of a level. Since this is a leaf node 1 is passed.
- 1 - Id of a shaper. -1 indicates that shaper is not attached.
- 0 - Congestion management is not supported.

- 0 - Congestion management is not supported.
- 1 - Enable statistics counter for number of transmitted packets.
- 0 - Number of shared shapers.

8. Commit hierarchy:

```
testpmd> port tm hierarchy commit 0 no
```

Parameters have following meaning:

- 0 - Id of a port.
- no - Do not flush TM hierarchy if commit fails.

9. Start all ports

```
testpmd> port start all
```

10. Enable forwarding

```
testpmd> start
```

NETVSC POLL MODE DRIVER

The Netvsc Poll Mode driver (PMD) provides support for the paravirtualized network device for Microsoft Hyper-V. It can be used with Window Server 2008/2012/2016, Windows 10. The device offers multi-queue support (if kernel and host support it), checksum and segmentation offloads.

36.1 Features and Limitations of Hyper-V PMD

In this release, the hyper PMD driver provides the basic functionality of packet reception and transmission.

- It supports merge-able buffers per packet when receiving packets and scattered buffer per packet when transmitting packets. The packet size supported is from 64 to 65536.
- The PMD supports multicast packets and promiscuous mode subject to restrictions on the host. In order to this to work, the guest network configuration on Hyper-V must be configured to allow MAC address spoofing.
- The device has only a single MAC address. Hyper-V driver does not support MAC or VLAN filtering because the Hyper-V host does not support it.
- VLAN tags are always stripped and presented in mbuf tci field.
- The Hyper-V driver does not use or support interrupts. Link state change callback is done via change events in the packet ring.
- The maximum number of queues is limited by the host (currently 64). When used with 4.16 kernel only a single queue is available.
- This driver supports SR-IOV network acceleration. If SR-IOV is enabled then the driver will transparently manage the interface, and send and receive packets using the VF path. The VDEV_NETVSC and FAILSAFE drivers are *not* used when using netvsc PMD.

36.2 Installation

The Netvsc PMD is a standalone driver, similar to virtio and vmxnet3. Using Netvsc PMD requires that the associated VMBUS device be bound to the userspace I/O device driver for Hyper-V (uio_hv_generic). By default, all netvsc devices will be bound to the Linux kernel driver; in order to use netvsc PMD the device must first be overridden.

The first step is to identify the network device to override. VMBUS uses Universal Unique Identifiers (UUID) to identify devices on the bus similar to how PCI uses Domain:Bus:Function. The UUID asso-

ciated with a Linux kernel network device can be determined by looking at the sysfs information. To find the UUID for eth1 and store it in a shell variable:

```
DEV_UUID=$(basename $(readlink /sys/class/net/eth1/device))
```

There are several possible ways to assign the uio device driver for a device. The easiest way (but only on 4.18 or later) is to use the [driverctl Device Driver control utility](#) to override the normal kernel device.

```
driverctl -b vmbus set-override $DEV_UUID uio_hv_generic
```

Any settings done with driverctl are by default persistent and will be reapplied on reboot.

On older kernels, the same effect can be had by manual sysfs bind and unbind operations:

```
NET_UUID="f8615163-df3e-46c5-913f-f2d2f965ed0e"
modprobe uio_hv_generic
echo $NET_UUID > /sys/bus/vmbus/drivers/uio_hv_generic/new_id
echo $DEV_UUID > /sys/bus/vmbus/drivers/hv_netvsc/unbind
echo $DEV_UUID > /sys/bus/vmbus/drivers/uio_hv_generic/bind
```

Note: The dpdk-devbind.py script can not be used since it only handles PCI devices.

36.3 Prerequisites

The following prerequisites apply:

- Linux kernel support for UIO on vmbus is done with the uio_hv_generic driver. Full support of multiple queues requires the 4.17 kernel. It is possible to use the netvsc PMD with 4.16 kernel but it is limited to a single queue.

36.4 Netvsc PMD arguments

The user can specify below argument in devargs.

1. latency:

A netvsc device uses a mailbox page to indicate to the host that there is something in the transmit queue. The host scans this page at a periodic interval. This parameter allows adjusting the value that is used by the host. Smaller values improve transmit latency, and larger values save CPU cycles. This parameter is in microseconds. If the value is too large or too small it will be ignored by the host. (Default: 50)

NFB POLL MODE DRIVER LIBRARY

The NFB poll mode driver library implements support for the Netcope FPGA Boards (**NFB-40G2**, **NFB-100G2**, **NFB-200G2QL**) and Silicom **FB2CGG3** card, FPGA-based programmable NICs. The NFB PMD uses interface provided by the libnfb library to communicate with these cards over the nfb layer.

More information about the [NFB cards](#) and used technology ([Netcope Development Kit](#)) can be found on the [Netcope Technologies website](#).

Note: This driver has external dependencies. Therefore it is disabled in default configuration files. It can be enabled by setting `CONFIG_RTE_LIBRTE_NFB_PMD=y` and recompiling.

Note: Currently the driver is supported only on x86_64 architectures. Only x86_64 versions of the external libraries are provided.

37.1 Prerequisites

This PMD requires kernel modules which are responsible for initialization and allocation of resources needed for nfb layer function. Communication between PMD and kernel modules is mediated by libnfb library. These kernel modules and library are not part of DPDK and must be installed separately:

- **libnfb library**

The library provides API for initialization of nfb transfers, receiving and transmitting data segments.

- **Kernel modules**

- nfb

Kernel modules manage initialization of hardware, allocation and sharing of resources for user space applications.

Dependencies can be found here: [Netcope common](#).

37.1.1 Versions of the packages

The minimum version of the provided packages:

- for DPDK from 19.05

37.2 Configuration

These configuration options can be modified before compilation in the `.config` file:

- `CONFIG_RTE_LIBRTE_NFB_PMD` default value: **n**

Value **y** enables compilation of nfb PMD.

Timestamps

The PMD supports hardware timestamps of frame receipt on physical network interface. In order to use the timestamps, the hardware timestamping unit must be enabled (follow the documentation of the NFB products) and the device argument `timestamp=1` must be used.

```
$RTE_TARGET/app/testpmd -w b3:00.0,timestamp=1 <other EAL params> -- <testpmd params>
```

When the timestamps are enabled with the *devarg*, a timestamp validity flag is set in the Mbufs containing received frames and timestamp is inserted into the *rte_mbuf* struct.

The timestamp is an *uint64_t* field. Its lower 32 bits represent *seconds* portion of the timestamp (number of seconds elapsed since 1.1.1970 00:00:00 UTC) and its higher 32 bits represent *nanosecond* portion of the timestamp (number of nanoseconds elapsed since the beginning of the second in the *seconds* portion).

37.3 Using the NFB PMD

Kernel modules have to be loaded before running the DPDK application.

37.4 NFB card architecture

The NFB cards are multi-port multi-queue cards, where (generally) data from any Ethernet port may be sent to any queue. They are represented in DPDK as a single port.

NFB-200G2QL card employs an add-on cable which allows to connect it to two physical PCI-E slots at the same time (see the diagram below). This is done to allow 200 Gbps of traffic to be transferred through the PCI-E bus (note that a single PCI-E 3.0 x16 slot provides only 125 Gbps theoretical throughput).

Although each slot may be connected to a different CPU and therefore to a different NUMA node, the card is represented as a single port in DPDK. To work with data from the individual queues on the right NUMA node, connection of NUMA nodes on first and last queue (each NUMA node has half of the queues) need to be checked.

37.5 Limitations

Driver is usable only on Linux architecture, namely on CentOS.

Since a card is always represented as a single port, but can be connected to two NUMA nodes, there is need for manual check where master/slave is connected.

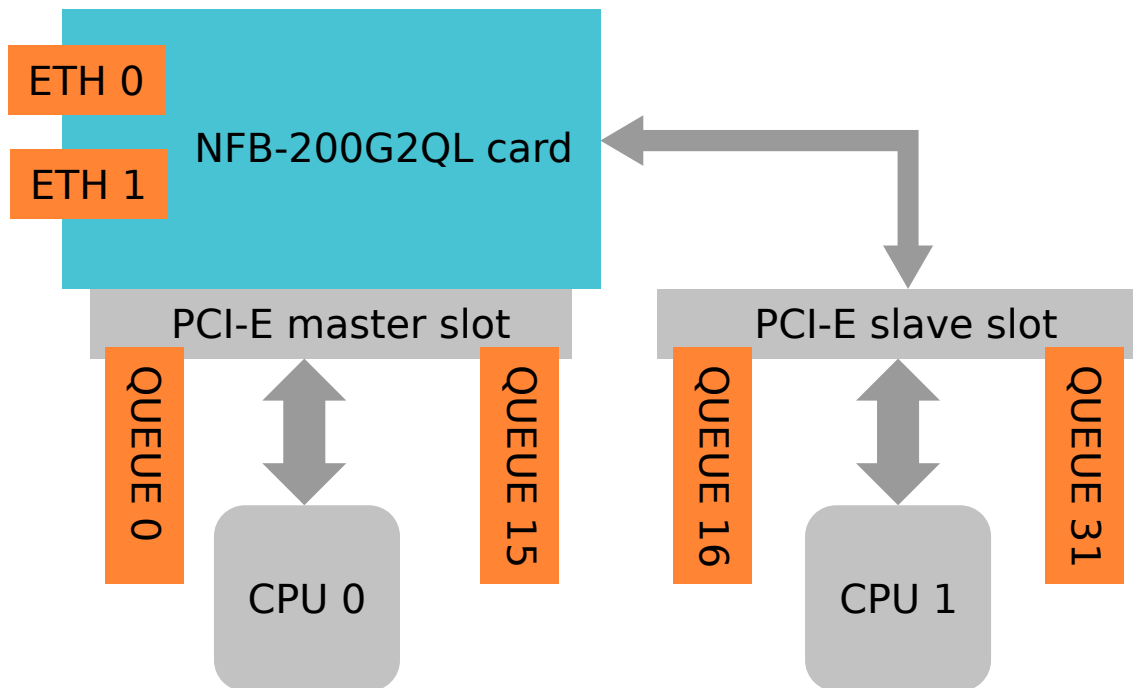


Fig. 37.1: NFB-200G2QL high-level diagram

37.6 Example of usage

Read packets from 0. and 1. receive queue and write them to 0. and 1. transmit queue:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 2 \
-- --port-topology=chained --rxq=2 --txq=2 --nb-cores=2 -i -a
```

Example output:

```
[...]
EAL: PCI device 0000:06:00.0 on NUMA socket -1
EAL: probe driver: 1b26:clcl net_nfb
PMD: Initializing NFB device (0000:06:00.0)
PMD: Available DMA queues RX: 8 TX: 8
PMD: NFB device (0000:06:00.0) successfully initialized
Interactive-mode selected
Auto-start selected
Configuring Port 0 (socket 0)
Port 0: 00:11:17:00:00:00
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Done
Start automatic packet forwarding
io packet forwarding - CRC stripping disabled - packets/burst=32
nb forwarding cores=2 - nb forwarding ports=1
RX queues=2 - RX desc=128 - RX free threshold=0
RX threshold registers: pthresh=0 hthresh=0 wthresh=0
TX queues=2 - TX desc=512 - TX free threshold=0
TX threshold registers: pthresh=0 hthresh=0 wthresh=0
TX RS bit threshold=0 - TXQ flags=0x0
testpmd>
```

NFP POLL MODE DRIVER LIBRARY

Netronome's sixth generation of flow processors pack 216 programmable cores and over 100 hardware accelerators that uniquely combine packet, flow, security and content processing in a single device that scales up to 400-Gb/s.

This document explains how to use DPDK with the Netronome Poll Mode Driver (PMD) supporting Netronome's Network Flow Processor 6xxx (NFP-6xxx) and Netronome's Flow Processor 4xxx (NFP-4xxx).

NFP is a SRIOV capable device and the PMD driver supports the physical function (PF) and the virtual functions (VFs).

38.1 Dependencies

Before using the Netronome's DPDK PMD some NFP configuration, which is not related to DPDK, is required. The system requires installation of **Netronome's BSP (Board Support Package)** along with a specific NFP firmware application. Netronome's NSP ABI version should be 0.20 or higher.

If you have a NFP device you should already have the code and documentation for this configuration. Contact support@netronome.com to obtain the latest available firmware.

The NFP Linux netdev kernel driver for VFs has been a part of the vanilla kernel since kernel version 4.5, and support for the PF since kernel version 4.11. Support for older kernels can be obtained on Github at <https://github.com/Netronome/nfp-driv-kmods> along with the build instructions.

NFP PMD needs to be used along with UIO `igb_uio` or VFIO (`vfio-pci`) Linux kernel driver.

38.2 Building the software

Netronome's PMD code is provided in the **drivers/net/nfp** directory. Although NFP PMD has Netronome's BSP dependencies, it is possible to compile it along with other DPDK PMDs even if no BSP was installed previously. Of course, a DPDK app will require such a BSP installed for using the NFP PMD, along with a specific NFP firmware application.

Default PMD configuration is at the **common_linux configuration** file:

- **CONFIG_RTE_LIBRTE_NFP_PMD=y**

Once the DPDK is built all the DPDK apps and examples include support for the NFP PMD.

38.3 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

38.4 Using the PF

NFP PMD supports using the NFP PF as another DPDK port, but it does not have any functionality for controlling VFs. In fact, it is not possible to use the PMD with the VFs if the PF is being used by DPDK, that is, with the NFP PF bound to `igb_uio` or `vfio-pci` kernel drivers. Future DPDK versions will have a PMD able to work with the PF and VFs at the same time and with the PF implementing VF management along with other PF-only functionalities/offloads.

The PMD PF has extra work to do which will delay the DPDK app initialization like uploading the firmware and configure the Link state properly when starting or stopping a PF port. Since DPDK 18.05 the firmware upload happens when a PF is initialized, which was not always true with older DPDK versions.

Depending on the Netronome product installed in the system, firmware files should be available under `/lib/firmware/netronome`. DPDK PMD supporting the PF looks for a firmware file in this order:

1. First try to find a firmware image specific for this device using the NFP serial number:

```
serial-00-15-4d-12-20-65-10-ff.nffw
```

2. Then try the PCI name:

```
pci-0000:04:00.0.nffw
```

3. Finally try the card type and media:

```
nic_AMDA0099-0001_2x25.nffw
```

Netronome's software packages install firmware files under `/lib/firmware/netronome` to support all the Netronome's SmartNICs and different firmware applications. This is usually done using file names based on SmartNIC type and media and with a directory per firmware application. Options 1 and 2 for firmware filenames allow more than one SmartNIC, same type of SmartNIC or different ones, and to upload a different firmware to each SmartNIC.

38.5 PF multiport support

Some NFP cards support several physical ports with just one single PCI device. The DPDK core is designed with a 1:1 relationship between PCI devices and DPDK ports, so NFP PMD PF support requires handling the multiport case specifically. During NFP PF initialization, the PMD will extract the information about the number of PF ports from the firmware and will create as many DPDK ports as needed.

Because the unusual relationship between a single PCI device and several DPDK ports, there are some limitations when using more than one PF DPDK port: there is no support for RX interrupts and it is not possible either to use those PF ports with the device hotplug functionality.

38.6 PF multiprocess support

Due to how the driver needs to access the NFP through a CPP interface, which implies to use specific registers inside the chip, the number of secondary processes with PF ports is limited to only one.

This limitation will be solved in future versions but having basic multiprocess support is important for allowing development and debugging through the PF using a secondary process which will create a CPP bridge for user space tools accessing the NFP.

38.7 System configuration

1. **Enable SR-IOV on the NFP device:** The current NFP PMD supports the PF and the VFs on a NFP device. However, it is not possible to work with both at the same time because the VFs require the PF being bound to the NFP PF Linux netdev driver. Make sure you are working with a kernel with NFP PF support or get the drivers from the above Github repository and follow the instructions for building and installing it.

VFs need to be enabled before they can be used with the PMD. Before enabling the VFs it is useful to obtain information about the current NFP PCI device detected by the system:

```
lspci -d19ee:
```

Now, for example, configure two virtual functions on a NFP-6xxx device whose PCI system identity is “0000:03:00.0”:

```
echo 2 > /sys/bus/pci/devices/0000:03:00.0/sriov_numvfs
```

The result of this command may be shown using `lspci` again:

```
lspci -d19ee: -k
```

Two new PCI devices should appear in the output of the above command. The `-k` option shows the device driver, if any, that devices are bound to. Depending on the modules loaded at this point the new PCI devices may be bound to `nfp_netvf` driver.

OCTEON TX POLL MODE DRIVER

The OCTEON TX ETHDEV PMD (`librte_pmd_octeontx`) provides poll mode ethdev driver support for the inbuilt network device found in the **Cavium OCTEON TX** SoC family as well as their virtual functions (VF) in SR-IOV context.

More information can be found at [Cavium, Inc Official Website](#).

39.1 Features

Features of the OCTEON TX Ethdev PMD are:

- Packet type information
- Promiscuous mode
- Port hardware statistics
- Jumbo frames
- Link state information
- SR-IOV VF
- Multiple queues for TX
- Lock-free Tx queue
- HW offloaded *ethdev Rx queue* to *eventdev event queue* packet injection

39.2 Supported OCTEON TX SoCs

- CN83xx

39.3 Unsupported features

The features supported by the device and not yet supported by this PMD include:

- Receive Side Scaling (RSS)
- Scattered and gather for TX and RX
- Ingress classification support

- Egress hierarchical scheduling, traffic shaping, and marking

39.4 Prerequisites

See `../platform/octeontx` for setup information.

39.5 Pre-Installation Configuration

39.5.1 Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_OCTEONTX_PMD` (default `y`)

Toggle compilation of the `librte_pmd_octeontx` driver.

39.5.2 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

To compile the OCTEON TX PMD for Linux arm64 gcc target, run the following make command:

```
cd <DPDK-source-directory>
make config T=arm64-thunderx-linux-gcc install
```

1. Running testpmd:

Follow instructions available in the document *compiling and testing a PMD for a NIC* to run `testpmd`.

Example output:

```
./arm64-thunderx-linux-gcc/app/testpmd -c 700 \
    --base-virtaddr=0x100000000000 \
    --mbuf-pool-ops-name="octeontx_fpvf" \
    --vdev='event_octeontx' \
    --vdev='eth_octeontx,nr_port=2' \
    -- --rxq=1 --txq=1 --nb-core=2 \
    --total-num-mbufs=16384 -i

.....
EAL: Detected 24 lcore(s)
EAL: Probing VFIO support...
EAL: VFIO support initialized
.....
EAL: PCI device 0000:07:00.1 on NUMA socket 0
EAL:   probe driver: 177d:a04b octeontx_ssov
.....
EAL: PCI device 0001:02:00.7 on NUMA socket 0
EAL:   probe driver: 177d:a0dd octeontx_pkivf
.....
EAL: PCI device 0001:03:01.0 on NUMA socket 0
EAL:   probe driver: 177d:a049 octeontx_pkovf
.....
PMD: octeontx_probe(): created ethdev eth_octeontx for port 0
PMD: octeontx_probe(): created ethdev eth_octeontx for port 1
.....
```

```
Configuring Port 0 (socket 0)
Port 0: 00:0F:B7:11:94:46
Configuring Port 1 (socket 0)
Port 1: 00:0F:B7:11:94:47
.....
Checking link statuses...
Port 0 Link Up - speed 40000 Mbps - full-duplex
Port 1 Link Up - speed 40000 Mbps - full-duplex
Done
testpmd>
```

39.6 Initialization

The OCTEON TX ethdev pmd is exposed as a vdev device which consists of a set of PKI and PKO PCIe VF devices. On EAL initialization, PKI/PKO PCIe VF devices will be probed and then the vdev device can be created from the application code, or from the EAL command line based on the number of probed/bound PKI/PKO PCIe VF device to DPDK by

- Invoking `rte_vdev_init("eth_octeontx")` from the application
- Using `--vdev="eth_octeontx"` in the EAL options, which will call `rte_vdev_init()` internally

39.6.1 Device arguments

Each ethdev port is mapped to a physical port(LMAC), Application can specify the number of interesting ports with `nr_ports` argument.

39.6.2 Dependency

`eth_octeontx` pmd is depend on `event_octeontx` eventdev device and `octeontx_fpvf` external mempool handler.

Example:

```
./your_dpdk_application --mbuf-pool-ops-name="octeontx_fpvf" \
--vdev='event_octeontx' \
--vdev="eth_octeontx,nr_port=2"
```

39.7 Limitations

39.7.1 `octeontx_fpvf` external mempool handler dependency

The OCTEON TX SoC family NIC has inbuilt HW assisted external mempool manager. This driver will only work with `octeontx_fpvf` external mempool handler as it is the most performance effective way for packet allocation and Tx buffer recycling on OCTEON TX SoC platform.

39.7.2 CRC stripping

The OCTEON TX SoC family NICs strip the CRC for every packets coming into the host interface irrespective of the offload configuration.

39.7.3 Maximum packet length

The OCTEON TX SoC family NICs support a maximum of a 32K jumbo frame. The value is fixed and cannot be changed. So, even when the `rxmode.max_rx_pkt_len` member of `struct rte_eth_conf` is set to a value lower than 32k, frames up to 32k bytes can still reach the host interface.

39.7.4 Maximum mempool size

The maximum mempool size supplied to Rx queue setup should be less than 128K. When running `testpmd` on OCTEON TX the application can limit the number of mbufs by using the option `--total-num-mbufs=131072`.

OCTEON TX2 POLL MODE DRIVER

The OCTEON TX2 ETHDEV PMD (`librte_pmd_octeontx2`) provides poll mode ethdev driver support for the inbuilt network device found in **Marvell OCTEON TX2** SoC family as well as for their virtual functions (VF) in SR-IOV context.

More information can be found at [Marvell Official Website](#).

40.1 Features

Features of the OCTEON TX2 Ethdev PMD are:

- Packet type information
- Promiscuous mode
- Jumbo frames
- SR-IOV VF
- Lock-free Tx queue
- Multiple queues for TX and RX
- Receiver Side Scaling (RSS)
- MAC/VLAN filtering
- Multicast MAC filtering
- Generic flow API
- Inner and Outer Checksum offload
- VLAN/QinQ stripping and insertion
- Port hardware statistics
- Link state information
- Link flow control
- MTU update
- Scatter-Gather IO support
- Vector Poll mode driver
- Debug utilities - Context dump and error interrupt support

- IEEE1588 timestamping
- HW offloaded *ethdev Rx queue* to *eventdev event queue* packet injection
- Support Rx interrupt

40.2 Prerequisites

See `../platform/octeontx2` for setup information.

40.3 Compile time Config Options

The following options may be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_OCTEONTX2_PMD` (default `y`)
Toggle compilation of the `librte_pmd_octeontx2` driver.

40.4 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

To compile the OCTEON TX2 PMD for Linux arm64 gcc, use `arm64-octeontx2-linux-gcc` as target.

1. Running testpmd:

Follow instructions available in the document *compiling and testing a PMD for a NIC* to run `testpmd`.

Example output:

```
./build/app/testpmd -c 0x300 -w 0002:02:00.0 -- --portmask=0x1 --nb-cores=1 --port-topolog
EAL: Detected 24 lcore(s)
EAL: Detected 1 NUMA nodes
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket
EAL: No available hugepages reported in hugepages-2048kB
EAL: Probing VFIO support...
EAL: VFIO support initialized
EAL: PCI device 0002:02:00.0 on NUMA socket 0
EAL:   probe driver: 177d:a063 net_octeontx2
EAL:   using IOMMU type 1 (Type 1)
testpmd: create a new mbuf pool <mbuf_pool_socket_0>: n=267456, size=2176, socket=0
testpmd: preferred mempool ops selected: octeontx2_npa
Configuring Port 0 (socket 0)
PMD: Port 0: Link Up - speed 40000 Mbps - full-duplex

Port 0: link state change event
Port 0: 36:10:66:88:7A:57
Checking link statuses...
Done
No commandline core given, start packet forwarding
io packet forwarding - ports=1 - cores=1 - streams=1 - NUMA support enabled, MP allocation
Logical Core 9 (socket 0) forwards packets on 1 streams:
  RX P=0/Q=0 (socket 0) -> TX P=0/Q=0 (socket 0) peer=02:00:00:00:00:00

io packet forwarding packets/burst=32
nb forwarding cores=1 - nb forwarding ports=1
```

```

port 0: RX queue number: 1 Tx queue number: 1
Rx offloads=0x0 Tx offloads=0x10000
RX queue: 0
  RX desc=512 - RX free threshold=0
  RX threshold registers: pthresh=0 hthresh=0 wthresh=0
  RX Offloads=0x0
TX queue: 0
  TX desc=512 - TX free threshold=0
  TX threshold registers: pthresh=0 hthresh=0 wthresh=0
  TX offloads=0x10000 - TX RS bit threshold=0
Press enter to exit

```

40.5 Runtime Config Options

- Rx&Tx scalar mode enable (default 0)

Ethdev supports both scalar and vector mode, it may be selected at runtime using `scalar_enable` devargs parameter.

- RSS reta size (default 64)

RSS redirection table size may be configured during runtime using `reta_size` devargs parameter.

For example:

```
-w 0002:02:00.0,reta_size=256
```

With the above configuration, reta table of size 256 is populated.

- Flow priority levels (default 3)

RTE Flow priority levels can be configured during runtime using `flow_max_priority` devargs parameter.

For example:

```
-w 0002:02:00.0,flow_max_priority=10
```

With the above configuration, priority level was set to 10 (0-9). Max priority level supported is 32.

- Reserve Flow entries (default 8)

RTE flow entries can be pre allocated and the size of pre allocation can be selected runtime using `flow_prealloc_size` devargs parameter.

For example:

```
-w 0002:02:00.0,flow_prealloc_size=4
```

With the above configuration, pre alloc size was set to 4. Max pre alloc size supported is 32.

- Max SQB buffer count (default 512)

Send queue descriptor buffer count may be limited during runtime using `max_sqb_count` devargs parameter.

For example:

```
-w 0002:02:00.0,max_sqb_count=64
```

With the above configuration, each send queue's descriptor buffer count is limited to a maximum of 64 buffers.

- `switch_header enable` (default none)

A port can be configured to a specific switch header type by using `switch_header devargs` parameter.

For example:

```
-w 0002:02:00.0,switch_header="higig2"
```

With the above configuration, `higig2` will be enabled on that port and the traffic on this port should be `higig2` traffic only. Supported switch header types are “`higig2`” and “`dsa`”.

Note: Above devarg parameters are configurable per device, user needs to pass the parameters to all the PCIe devices if application requires to configure on all the ethdev ports.

40.6 Limitations

40.6.1 `mempool1_octeontx2` external mempool handler dependency

The OCTEON TX2 SoC family NIC has inbuilt HW assisted external mempool manager. `net_octeontx2` pmd only works with `mempool1_octeontx2` mempool handler as it is performance wise most effective way for packet allocation and Tx buffer recycling on OCTEON TX2 SoC platform.

40.6.2 CRC stripping

The OCTEON TX2 SoC family NICs strip the CRC for every packet being received by the host interface irrespective of the offload configuration.

40.6.3 Multicast MAC filtering

`net_octeontx2` pmd supports multicast mac filtering feature only on physical function devices.

40.6.4 SDP interface support

OCTEON TX2 SDP interface support is limited to PF device, No VF support.

40.7 Debugging Options

Table 40.1: OCTEON TX2 ethdev debug options

#	Component	EAL log command
1	NIX	<code>-log-level='pmd.net.octeonx2,8'</code>
2	NPC	<code>-log-level='pmd.net.octeonx2.flow,8'</code>

40.8 RTE Flow Support

The OCTEON TX2 SoC family NIC has support for the following patterns and actions.

Patterns:

Table 40.2: Item types

#	Pattern Type
1	<code>RTE_FLOW_ITEM_TYPE_ETH</code>
2	<code>RTE_FLOW_ITEM_TYPE_VLAN</code>
3	<code>RTE_FLOW_ITEM_TYPE_E_TAG</code>
4	<code>RTE_FLOW_ITEM_TYPE_IPV4</code>
5	<code>RTE_FLOW_ITEM_TYPE_IPV6</code>
6	<code>RTE_FLOW_ITEM_TYPE_ARP_ETH_IPV4</code>
7	<code>RTE_FLOW_ITEM_TYPE_MPLS</code>
8	<code>RTE_FLOW_ITEM_TYPE_ICMP</code>
9	<code>RTE_FLOW_ITEM_TYPE_UDP</code>
10	<code>RTE_FLOW_ITEM_TYPE_TCP</code>
11	<code>RTE_FLOW_ITEM_TYPE_SCTP</code>
12	<code>RTE_FLOW_ITEM_TYPE_ESP</code>
13	<code>RTE_FLOW_ITEM_TYPE_GRE</code>
14	<code>RTE_FLOW_ITEM_TYPE_NVGRE</code>
15	<code>RTE_FLOW_ITEM_TYPE_VXLAN</code>
16	<code>RTE_FLOW_ITEM_TYPE_GTPC</code>
17	<code>RTE_FLOW_ITEM_TYPE_GTPU</code>
18	<code>RTE_FLOW_ITEM_TYPE_GENEVE</code>
19	<code>RTE_FLOW_ITEM_TYPE_VXLAN_GPE</code>
20	<code>RTE_FLOW_ITEM_TYPE_IPV6_EXT</code>
21	<code>RTE_FLOW_ITEM_TYPE_VOID</code>
22	<code>RTE_FLOW_ITEM_TYPE_ANY</code>
23	<code>RTE_FLOW_ITEM_TYPE_GRE_KEY</code>
24	<code>RTE_FLOW_ITEM_TYPE_HIGIG2</code>

Note: `RTE_FLOW_ITEM_TYPE_GRE_KEY` works only when checksum and routing bits in the GRE header are equal to 0.

Actions:

Table 40.3: Ingress action types

#	Action Type
1	RTE_FLOW_ACTION_TYPE_VOID
2	RTE_FLOW_ACTION_TYPE_MARK
3	RTE_FLOW_ACTION_TYPE_FLAG
4	RTE_FLOW_ACTION_TYPE_COUNT
5	RTE_FLOW_ACTION_TYPE_DROP
6	RTE_FLOW_ACTION_TYPE_QUEUE
7	RTE_FLOW_ACTION_TYPE_RSS
8	RTE_FLOW_ACTION_TYPE_SECURITY
9	RTE_FLOW_ACTION_TYPE_PF
10	RTE_FLOW_ACTION_TYPE_VF

Table 40.4: Egress action types

#	Action Type
1	RTE_FLOW_ACTION_TYPE_COUNT
2	RTE_FLOW_ACTION_TYPE_DROP

PFE POLL MODE DRIVER

The PFE NIC PMD (`librte_pmd_pfe`) provides poll mode driver support for the inbuilt NIC found in the **NXP LS1012** SoC.

More information can be found at [NXP Official Website](#).

41.1 PFE

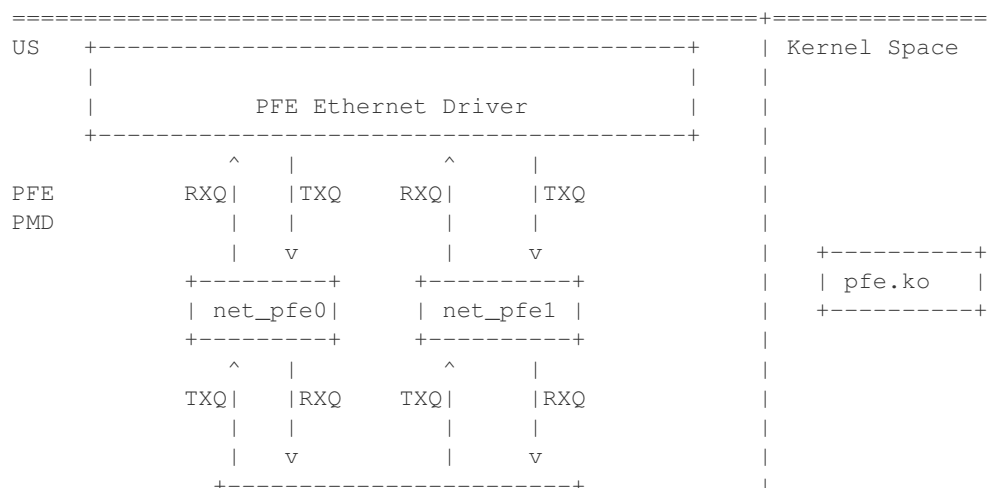
This section provides an overview of the NXP PFE and how it is integrated into the DPDK.

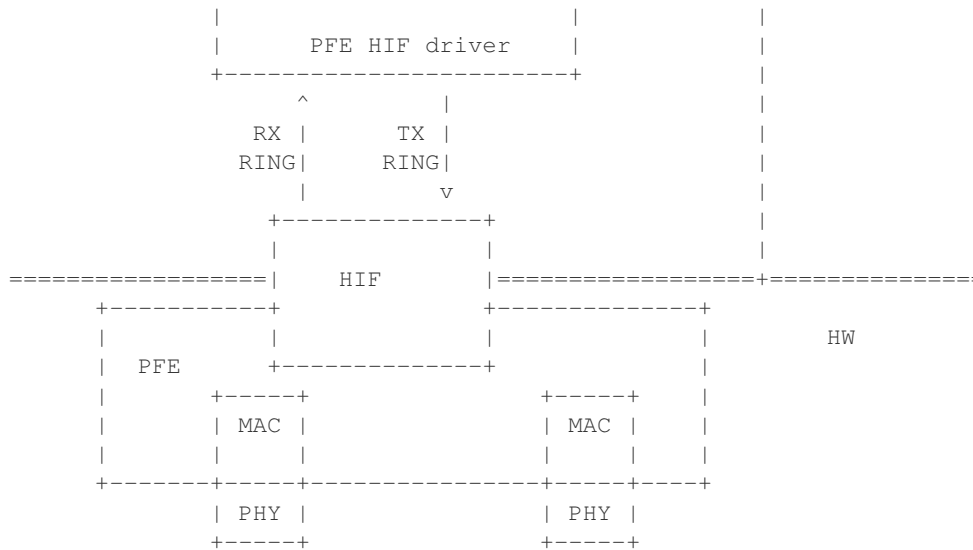
Contents summary

- PFE overview
- PFE features
- Supported PFE SoCs
- Prerequisites
- Driver compilation and testing
- Limitations

41.1.1 PFE Overview

PFE is a hardware programmable packet forwarding engine to provide high performance Ethernet interfaces. The diagram below shows a system level overview of PFE:





The HIF, PFE, MAC and PHY are the hardware blocks, the pfe.ko is a kernel module, the PFE HIF driver and the PFE ethernet driver combined represent as DPDK PFE poll mode driver are running in the userspace.

The PFE hardware supports one HIF (host interface) RX ring and one TX ring to send and receive packets through packet forwarding engine. Both network interface traffic is multiplexed and send over HIF queue.

net_pfe0 and net_pfe1 are logical ethernet interfaces, created by HIF client driver. HIF driver is responsible for send and receive packets between host interface and these logical interfaces. PFE ethernet driver is a hardware independent and register with the HIF client driver to transmit and receive packets from HIF via logical interfaces.

pfe.ko is required for PHY initialisation and also responsible for creating the character device “pfe_us_cdev” which will be used for interacting with the kernel layer for link status.

41.1.2 PFE Features

- L3/L4 checksum offload
- Packet type parsing
- Basic stats
- MTU update
- Promiscuous mode
- Allmulticast mode
- Link status
- ARMv8

41.1.3 Supported PFE SoCs

- LS1012

41.1.4 Prerequisites

Below are some pre-requisites for executing PFE PMD on a PFE compatible board:

1. **ARM 64 Tool Chain**

For example, the [*aarch64* Linaro Toolchain](#).

2. **Linux Kernel**

It can be obtained from [NXP's Github hosting](#).

3. **Rootfile system**

Any *aarch64* supporting filesystem can be used. For example, Ubuntu 16.04 LTS (Xenial) or 18.04 (Bionic) userland which can be obtained from [here](#).

4. The ethernet device will be registered as virtual device, so pfe has dependency on **rte_bus_vdev** library and it is mandatory to use `-vdev` with value `net_pfe` to run DPDK application.

The following dependencies are not part of DPDK and must be installed separately:

- **NXP Linux LSDK**

NXP Layerscape software development kit (LSDK) includes support for family of QorIQ® ARM-Architecture-based system on chip (SoC) processors and corresponding boards.

It includes the Linux board support packages (BSPs) for NXP SoCs, a fully operational tool chain, kernel and board specific modules.

LSDK and related information can be obtained from: [LSDK](#)

- **pfe kernel module**

pfe kernel module can be obtained from NXP Layerscape software development kit at location `/lib/modules/<kernel version>/kernel/drivers/staging/fsl_ppfe` in rootfs. Module should be loaded using below command:

```
insmod pfe.ko us=1
```

41.1.5 Driver compilation and testing

Follow instructions available in the document [compiling and testing a PMD for a NIC](#) to launch **testpmd**

Additionally, PFE driver needs `-vdev` as an input with value `net_pfe` to execute DPDK application. There is an optional parameter `intf` available to specify port ID. PFE driver supports only two interfaces, so valid values for `intf` are 0 and 1. see the command below:

```
<dpdk app> <EAL args> --vdev="net_pfe0,intf=0" --vdev="net_pfe1,intf=1" -- ...
```

41.1.6 Limitations

- Multi buffer pool cannot be supported.

QEDE POLL MODE DRIVER

The QEDE poll mode driver library (`librte_pmd_qede`) implements support for **QLogic FastLinQ QL4xxx 10G/25G/40G/50G/100G Intelligent Ethernet Adapters (IEA) and Converged Network Adapters (CNA)** family of adapters as well as SR-IOV virtual functions (VF). It is supported on several standard Linux distros like RHEL, SLES, Ubuntu etc. It is compile-tested under FreeBSD OS.

More information can be found at [QLogic Corporation's Website](#).

42.1 Supported Features

- Unicast/Multicast filtering
- Promiscuous mode
- Allmulti mode
- Port hardware statistics
- Jumbo frames
- Multiple MAC address
- MTU change
- Default pause flow control
- Multiprocess aware
- Scatter-Gather
- Multiple Rx/Tx queues
- RSS (with RETA/hash table/key)
- TSS
- Stateless checksum offloads (IPv4/IPv6/TCP/UDP)
- LRO/TSO
- VLAN offload - Filtering and stripping
- N-tuple filter and flow director (limited support)
- NPAR (NIC Partitioning)
- SR-IOV VF
- GRE Tunneling offload

- GENEVE Tunneling offload
- VXLAN Tunneling offload
- MPLSoUDP Tx Tunneling offload
- Generic flow API

42.2 Non-supported Features

- SR-IOV PF

42.3 Co-existence considerations

- QLogic FastLinQ QL4xxxx CNAs support Ethernet, RDMA, iSCSI and FCoE functionalities. These functionalities are supported using QLogic Linux kernel drivers qed, qede, qedr, qedi and qedf. DPDK is supported on these adapters using qede PMD.
- When SR-IOV is not enabled on the adapter, QLogic Linux kernel drivers (qed, qede, qedr, qedi and qedf) and qede PMD can't be attached to different PFs on a given QLogic FastLinQ QL4xxx adapter. A given adapter needs to be completely used by DPDK or Linux drivers. Before binding DPDK driver to one or more PFs on the adapter, please make sure to unbind Linux drivers from all PFs of the adapter. If there are multiple adapters on the system, one or more adapters can be used by DPDK driver completely and other adapters can be used by Linux drivers completely.
- When SR-IOV is enabled on the adapter, Linux kernel drivers (qed, qede, qedr, qedi and qedf) can be bound to the PFs of a given adapter and either qede PMD or Linux drivers (qed and qede) can be bound to the VFs of the adapter.
- For sharing an adapter between DPDK and Linux drivers, SRIOV needs to be enabled. Bind all the PFs to Linux Drivers(qed/qede). Create a VF on PFs where DPDK is desired and bind these VFs to qede_pmd. Binding of PFs simultaneously to DPDK and Linux drivers on a given adapter is not supported.

42.4 Supported QLogic Adapters

- QLogic FastLinQ QL4xxxx 10G/25G/40G/50G/100G Intelligent Ethernet Adapters (IEA) and Converged Network Adapters (CNA)

42.5 Prerequisites

- Requires storm firmware version **8.40.33.0**. Firmware may be available in-box in certain newer Linux distros under the standard directory E.g. `/lib/firmware/qed/qed_init_values-8.40.33.0.bin`. If the required firmware files are not available then download it from [linux-firmware git repository](#).
- Requires the NIC be updated minimally with **8.30.x.x** Management firmware(MFW) version supported for that NIC. It is highly recommended that the NIC be updated with the latest available management firmware version to get latest feature set. Management Firmware and Firmware

Upgrade Utility for Cavium FastLinQ(r) branded adapters can be downloaded from [Driver Download Center](#). For downloading Firmware Upgrade Utility, select NIC category, model and Linux distro. To update the management firmware, refer to the instructions in the Firmware Upgrade Utility Readme document. For OEM branded adapters please follow the instruction provided by the OEM to update the Management Firmware on the NIC.

- SR-IOV requires Linux PF driver version **8.20.x.x** or higher. If the required PF driver is not available then download it from [QLogic Driver Download Center](#). For downloading PF driver, select adapter category, model and Linux distro.

42.5.1 Performance note

- For better performance, it is recommended to use 4K or higher RX/TX rings.

42.5.2 Config File Options

The following options can be modified in the `.config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_QEDE_PMD` (default **y**)

Toggle compilation of QEDE PMD driver.

- `CONFIG_RTE_LIBRTE_QEDE_DEBUG_TX` (default **n**)

Toggle display of transmit fast path run-time messages.

- `CONFIG_RTE_LIBRTE_QEDE_DEBUG_RX` (default **n**)

Toggle display of receive fast path run-time messages.

- `CONFIG_RTE_LIBRTE_QEDE_FW` (default **""**)

Gives absolute path of firmware file. Eg: `"/lib/firmware/qed/qed_init_values-8.40.33.0.bin"`. Empty string indicates driver will pick up the firmware file from the default location `/lib/firmware/qed`. CAUTION this option is more for custom firmware, it is not recommended for use under normal condition.

42.5.3 Config notes

When there are multiple adapters and/or large number of Rx/Tx queues configured on the adapters, the default (2560) number of memzone descriptors may not be enough. Please increase the number of memzone descriptors to a higher number as needed. When sufficient number of memzone descriptors are not configured, user can potentially run into following error.

```
EAL: memzone_reserve_aligned_thread_unsafe(): No more room in config
```

42.6 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

42.7 RTE Flow Support

QLogic FastLinQ QL4xxxx NICs has support for the following patterns and actions.

Patterns:

Table 42.1: Item types

#	Pattern Type
1	RTE_FLOW_ITEM_TYPE_IPV4
2	RTE_FLOW_ITEM_TYPE_IPV6
3	RTE_FLOW_ITEM_TYPE_UDP
4	RTE_FLOW_ITEM_TYPE_TCP

Actions:

Table 42.2: Ingress action types

#	Action Type
1	RTE_FLOW_ACTION_TYPE_QUEUE
2	RTE_FLOW_ACTION_TYPE_DROP

42.8 SR-IOV: Prerequisites and Sample Application Notes

This section provides instructions to configure SR-IOV with Linux OS.

Note: `librte_pmd_qede` will be used to bind to SR-IOV VF device and Linux native kernel driver (`qede`) will function as SR-IOV PF driver. Requires PF driver to be 8.20.x.x or higher.

1. Verify SR-IOV and ARI capability is enabled on the adapter using `lspci`:

```
lspci -s <slot> -vvv
```

Example output:

```
[...]
Capabilities: [1b8 v1] Alternative Routing-ID Interpretation (ARI)
[...]
Capabilities: [1c0 v1] Single Root I/O Virtualization (SR-IOV)
[...]
Kernel driver in use: igb_uio
```

2. Load the kernel module:

```
modprobe qede
```

Example output:

```
systemd-udevd[4848]: renamed network interface eth0 to ens5f0
systemd-udevd[4848]: renamed network interface eth1 to ens5f1
```

3. Bring up the PF ports:

```
ifconfig ens5f0 up
ifconfig ens5f1 up
```

4. Create VF device(s):

Echo the number of VFs to be created into "`sriov_numvfs`" sysfs entry of the parent PF.

Example output:

```
echo 2 > /sys/devices/pci0000:00/0000:00:03.0/0000:81:00.0/sriov_numvfs
```

5. Assign VF MAC address:

Assign MAC address to the VF using `iproute2` utility. The syntax is:

```
ip link set <PF iface> vf <VF id> mac <macaddr>
```

Example output:

```
ip link set ens5f0 vf 0 mac 52:54:00:2f:9d:e8
```

6. PCI Passthrough:

The VF devices may be passed through to the guest VM using `virt-manager` or `virsh`. QEDE PMD should be used to bind the VF devices in the guest VM using the instructions from Driver compilation and testing section above.

7. Running testpmd (Supply `--log-level="pmd.net.qede.driver:info` to view informational messages):

Refer to the document [compiling and testing a PMD for a NIC](#) to run `testpmd` application.

Example output:

```
testpmd -l 0,4-11 -n 4 -- -i --nb-cores=8 --portmask=0xf --rxd=4096 \
--txd=4096 --txfreet=4068 --enable-rx-cksum --rxq=4 --txq=4 \
--rss-ip --rss-udp

[...]

EAL: PCI device 0000:84:00.0 on NUMA socket 1
EAL: probe driver: 1077:1634 rte_qede_pmd
EAL: Not managed by a supported kernel driver, skipped
EAL: PCI device 0000:84:00.1 on NUMA socket 1
EAL: probe driver: 1077:1634 rte_qede_pmd
EAL: Not managed by a supported kernel driver, skipped
EAL: PCI device 0000:88:00.0 on NUMA socket 1
EAL: probe driver: 1077:1656 rte_qede_pmd
EAL: PCI memory mapped at 0x7f738b200000
EAL: PCI memory mapped at 0x7f738b280000
EAL: PCI memory mapped at 0x7f738b300000
PMD: Chip details : BB1
PMD: Driver version : QEDE PMD 8.7.9.0_1.0.0
PMD: Firmware version : 8.7.7.0
PMD: Management firmware version : 8.7.8.0
PMD: Firmware file : /lib/firmware/qed/qed_init_values_zipped-8.7.7.0.bin
[QEDE PMD: (84:00.0:dpgk-port-0)]qede_common_dev_init:macaddr \
00:0e:1e:d2:09:9c
[...]
[QEDE PMD: (84:00.0:dpgk-port-0)]qede_tx_queue_setup:txq 0 num_desc 4096 \
tx_free_thresh 4068 socket 0
[QEDE PMD: (84:00.0:dpgk-port-0)]qede_tx_queue_setup:txq 1 num_desc 4096 \
tx_free_thresh 4068 socket 0
[QEDE PMD: (84:00.0:dpgk-port-0)]qede_tx_queue_setup:txq 2 num_desc 4096 \
tx_free_thresh 4068 socket 0
[QEDE PMD: (84:00.0:dpgk-port-0)]qede_tx_queue_setup:txq 3 num_desc 4096 \
tx_free_thresh 4068 socket 0
[QEDE PMD: (84:00.0:dpgk-port-0)]qede_rx_queue_setup:rxq 0 num_desc 4096 \
rx_buf_size=2148 socket 0
[QEDE PMD: (84:00.0:dpgk-port-0)]qede_rx_queue_setup:rxq 1 num_desc 4096 \
rx_buf_size=2148 socket 0
[QEDE PMD: (84:00.0:dpgk-port-0)]qede_rx_queue_setup:rxq 2 num_desc 4096 \
```

```
rx_buf_size=2148 socket 0
[QEDE PMD: (84:00.0:dpmk-port-0)]qede_rx_queue_setup:rxq 3 num_desc 4096 \
rx_buf_size=2148 socket 0
[QEDE PMD: (84:00.0:dpmk-port-0)]qede_dev_start:port 0
[QEDE PMD: (84:00.0:dpmk-port-0)]qede_dev_start:link status: down
[...]
Checking link statuses...
Port 0 Link Up - speed 25000 Mbps - full-duplex
Port 1 Link Up - speed 25000 Mbps - full-duplex
Port 2 Link Up - speed 25000 Mbps - full-duplex
Port 3 Link Up - speed 25000 Mbps - full-duplex
Done
testpmd>
```

SOLARFLARE LIBEFX-BASED POLL MODE DRIVER

The SFC EFX PMD (`librte_pmd_sfc_efx`) provides poll mode driver support for **Solarflare SFN7xxx and SFN8xxx** family of 10/40 Gbps adapters and **Solarflare XtremeScale X2xxx** family of 10/25/40/50/100 Gbps adapters. SFC EFX PMD has support for the latest Linux and FreeBSD operating systems.

More information can be found at [Solarflare Communications website](#).

43.1 Features

SFC EFX PMD has support for:

- Multiple transmit and receive queues
- Link state information including link status change interrupt
- IPv4/IPv6 TCP/UDP transmit checksum offload
- Inner IPv4/IPv6 TCP/UDP transmit checksum offload
- Port hardware statistics
- Extended statistics (see Solarflare Server Adapter User's Guide for the statistics description)
- Basic flow control
- MTU update
- Jumbo frames up to 9K
- Promiscuous mode
- Allmulticast mode
- TCP segmentation offload (TSO) including VXLAN and GENEVE encapsulated
- Multicast MAC filter
- IPv4/IPv6 TCP/UDP receive checksum offload
- Inner IPv4/IPv6 TCP/UDP receive checksum offload
- Received packet type information
- Receive side scaling (RSS)
- RSS hash
- Scattered Rx DMA for packet that are larger than a single Rx descriptor

- Receive queue interrupts
- Deferred receive and transmit queue start
- Transmit VLAN insertion (if running firmware variant supports it)
- Flow API
- Loopback

43.2 Non-supported Features

The features not yet supported include:

- Priority-based flow control
- Configurable RX CRC stripping (always stripped)
- Header split on receive
- VLAN filtering
- VLAN stripping
- LRO

43.3 Limitations

Due to requirements on receive buffer alignment and usage of the receive buffer for the auxiliary packet information provided by the NIC up to extra 269 (14 bytes prefix plus up to 255 bytes for end padding) bytes may be required in the receive buffer. It should be taken into account when mbuf pool for receive is created.

43.3.1 Equal stride super-buffer mode

When the receive queue uses equal stride super-buffer DMA mode, one HW Rx descriptor carries many Rx buffers which contiguously follow each other with some stride (equal to total size of `rte_mbuf` as mempool object). Each Rx buffer is an independent `rte_mbuf`. However dedicated mempool manager must be used when mempool for the Rx queue is created. The manager must support dequeue of the contiguous block of objects and provide mempool info API to get the block size.

Another limitation of a equal stride super-buffer mode, imposed by the firmware, is that it allows for a single RSS context.

43.4 Tunnels support

NVGRE, VXLAN and GENEVE tunnels are supported on SFN8xxx and X2xxx family adapters with full-feature firmware variant running. **sfboot** should be used to configure NIC to run full-feature firmware variant. See Solarflare Server Adapter User's Guide for details.

SFN8xxx and X2xxx family adapters provide either inner or outer packet classes. If adapter firmware advertises support for tunnels then the PMD configures the hardware to report inner classes, and outer

classes are not reported in received packets. However, for VXLAN and GENEVE tunnels the PMD does report UDP as the outer layer 4 packet type.

SFN8xxx and X2xxx family adapters report GENEVE packets as VXLAN. If UDP ports are configured for only one tunnel type then it is safe to treat VXLAN packet type indication as the corresponding UDP tunnel type.

43.5 Flow API support

Supported attributes:

- Ingress

Supported pattern items:

- VOID
- ETH (exact match of source/destination addresses, individual/group match of destination address, EtherType in the outer frame and exact match of destination addresses, individual/group match of destination address in the inner frame)
- VLAN (exact match of VID, double-tagging is supported)
- IPV4 (exact match of source/destination addresses, IP transport protocol)
- IPV6 (exact match of source/destination addresses, IP transport protocol)
- TCP (exact match of source/destination ports)
- UDP (exact match of source/destination ports)
- VXLAN (exact match of VXLAN network identifier)
- GENEVE (exact match of virtual network identifier, only Ethernet (0x6558) protocol type is supported)
- NVGRE (exact match of virtual subnet ID)

Supported actions:

- VOID
- QUEUE
- RSS
- DROP
- FLAG (supported only with ef10_essb Rx datapath)
- MARK (supported only with ef10_essb Rx datapath)

Validating flow rules depends on the firmware variant.

The `flow_isolated_mode` is supported.

43.5.1 Ethernet destination individual/group match

Ethernet item supports I/G matching, if only the corresponding bit is set in the mask of destination address. If destination address in the spec is multicast, it matches all multicast (and broadcast) packets, otherwise it matches unicast packets that are not filtered by other flow rules.

43.5.2 Exceptions to flow rules

There is a list of exceptional flow rule patterns which will not be accepted by the PMD. A pattern will be rejected if at least one of the conditions is met:

- Filtering by IPv4 or IPv6 EtherType without pattern items of internet layer and above.
- The last item is IPV4 or IPV6, and it's empty.
- Filtering by TCP or UDP IP transport protocol without pattern items of transport layer and above.
- The last item is TCP or UDP, and it's empty.

43.6 Supported NICs

- Solarflare XtremeScale Adapters:
 - Solarflare X2522 Dual Port SFP28 10/25GbE Adapter
 - Solarflare X2541 Single Port QSFP28 10/25G/100G Adapter
 - Solarflare X2542 Dual Port QSFP28 10/25G/100G Adapter
- Solarflare Flareon [Ultra] Server Adapters:
 - Solarflare SFN8522 Dual Port SFP+ Server Adapter
 - Solarflare SFN8522M Dual Port SFP+ Server Adapter
 - Solarflare SFN8042 Dual Port QSFP+ Server Adapter
 - Solarflare SFN8542 Dual Port QSFP+ Server Adapter
 - Solarflare SFN8722 Dual Port SFP+ OCP Server Adapter
 - Solarflare SFN7002F Dual Port SFP+ Server Adapter
 - Solarflare SFN7004F Quad Port SFP+ Server Adapter
 - Solarflare SFN7042Q Dual Port QSFP+ Server Adapter
 - Solarflare SFN7122F Dual Port SFP+ Server Adapter
 - Solarflare SFN7124F Quad Port SFP+ Server Adapter
 - Solarflare SFN7142Q Dual Port QSFP+ Server Adapter
 - Solarflare SFN7322F Precision Time Synchronization Server Adapter

43.7 Prerequisites

- Requires firmware version:
 - SFN7xxx: **4.7.1.1001** or higher
 - SFN8xxx: **6.0.2.1004** or higher

Visit [Solarflare Support Downloads](#) to get Solarflare Utilities (either Linux or FreeBSD) with the latest firmware. Follow instructions from Solarflare Server Adapter User's Guide to update firmware and configure the adapter.

43.8 Pre-Installation Configuration

43.8.1 Config File Options

The following options can be modified in the `.config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_SFC_EFX_PMD` (default **y**)
Enable compilation of Solarflare libefx-based poll-mode driver.
- `CONFIG_RTE_LIBRTE_SFC_EFX_DEBUG` (default **n**)
Enable compilation of the extra run-time consistency checks.

43.8.2 Per-Device Parameters

The following per-device parameters can be passed via EAL PCI device whitelist option like “-w 02:00:0,arg1=value1,...”.

Case-insensitive 1/y/yes/on or 0/n/no/off may be used to specify boolean parameters value.

- `rx_datapath` [`autolefx`|`ef10`|`ef10_esps`] (default **auto**)
Choose receive datapath implementation. **auto** allows the driver itself to make a choice based on firmware features available and required by the datapath implementation. **efx** chooses libefx-based datapath which supports Rx scatter. **ef10** chooses EF10 (SFN7xxx, SFN8xxx, X2xxx) native datapath which is more efficient than libefx-based and provides richer packet type classification. **ef10_esps** chooses SFNX2xxx equal stride packed stream datapath which may be used on DPDK firmware variant only (see notes about its limitations above).
- `tx_datapath` [`autolefx`|`ef10`|`ef10_simple`] (default **auto**)
Choose transmit datapath implementation. **auto** allows the driver itself to make a choice based on firmware features available and required by the datapath implementation. **efx** chooses libefx-based datapath which supports VLAN insertion (full-feature firmware variant only), TSO and multi-segment mbufs. Mbuf segments may come from different mempools, and mbuf reference counters are treated responsibly. **ef10** chooses EF10 (SFN7xxx, SFN8xxx, X2xxx) native datapath which is more efficient than libefx-based but has no VLAN insertion support yet. Mbuf segments may come from different mempools, and mbuf reference counters are treated responsibly. **ef10_simple** chooses EF10 (SFN7xxx, SFN8xxx, X2xxx) native datapath which is even more faster than **ef10** but does not support multi-segment mbufs, disallows multiple mempools and neglects mbuf reference counters.

- `perf_profile` [auto|throughput|low-latency] (default **throughput**)

Choose hardware tuning to be optimized for either throughput or low-latency. **auto** allows NIC firmware to make a choice based on installed licenses and firmware variant configured using **sfboot**.

- `stats_update_period_ms` [long] (default **1000**)

Adjust period in milliseconds to update port hardware statistics. The accepted range is 0 to 65535. The value of **0** may be used to disable periodic statistics update. One should note that it's only possible to set an arbitrary value on SFN8xxx and X2xxx provided that firmware version is 6.2.1.1033 or higher, otherwise any positive value will select a fixed update period of **1000** milliseconds

- `fw_variant` [dont-care|full-feature|ultra-low-latency| capture-packed-stream|dpdk] (default **dont-care**)

Choose the preferred firmware variant to use. In order for the selected option to have an effect, the **sfboot** utility must be configured with the **auto** firmware-variant option. The preferred firmware variant applies to all ports on the NIC. **dont-care** ensures that the driver can attach to an unprivileged function. The datapath firmware type to use is controlled by the **sfboot** utility. **full-feature** chooses full featured firmware. **ultra-low-latency** chooses firmware with fewer features but lower latency. **capture-packed-stream** chooses firmware for SolarCapture packed stream mode. **dpdk** chooses DPDK firmware with equal stride super-buffer Rx mode for higher Rx packet rate and packet marks support and firmware subvariant without checksumming on transmit for higher Tx packet rate if checksumming is not required.

- `rx_d_wait_timeout_ns` [long] (default **200 us**)

Adjust timeout in nanoseconds to head-of-line block to wait for Rx descriptors. The accepted range is 0 to 400 ms. Flow control should be enabled to make it work. The value of **0** disables it and packets are dropped immediately. When a packet is dropped because of no Rx descriptors, `rx_nodesc_drop_cnt` counter grows. The feature is supported only by the DPDK firmware variant when equal stride super-buffer Rx mode is used.

43.8.3 Dynamic Logging Parameters

One may leverage EAL option “-log-level” to change default levels for the log types supported by the driver. The option is used with an argument typically consisting of two parts separated by a colon.

Level value is the last part which takes a symbolic name (or integer). Log type is the former part which may shell match syntax. Depending on the choice of the expression, the given log level may be used either for some specific log type or for a subset of types.

SFC EFX PMD provides the following log types available for control:

- `pmd.net.sfc.driver` (default level is **notice**)

Affects driver-wide messages unrelated to any particular devices.

- `pmd.net.sfc.main` (default level is **notice**)

Matches a subset of per-port log types registered during runtime. A full name for a particular type may be obtained by appending a dot and a PCI device identifier (XXXX:XX:XX.X) to the prefix.

- `pmd.net.sfc.mcdi` (default level is **notice**)

Extra logging of the communication with the NIC's management CPU. The format of the log is consumed by the Solarflare netlogdecode cross-platform tool. May be managed per-port, as explained above.

SOFT NIC POLL MODE DRIVER

The Soft NIC allows building custom NIC pipelines in software. The Soft NIC pipeline is DIY and reconfigurable through `firmware` (DPDK Packet Framework script).

The Soft NIC leverages the DPDK Packet Framework libraries (`librte_port`, `librte_table` and `librte_pipeline`) to make it modular, flexible and extensible with new functionality. Please refer to DPDK Programmer's Guide, Chapter `Packet Framework` and DPDK Sample Application User Guide, Chapter `IP Pipeline Application` for more details.

The Soft NIC is configured through the standard DPDK `ethdev` API (`ethdev`, `flow`, `QoS`, `security`). The internal framework is not externally visible.

Key benefits:

- Can be used to augment missing features to HW NICs.
- Allows consumption of advanced DPDK features without application redesign.
- Allows out-of-the-box performance boost of DPDK consumers applications simply by instantiating this type of Ethernet device.

44.1 Flow

- `Device creation`: Each Soft NIC instance is a virtual device.
- `Device start`: The Soft NIC firmware script is executed every time the device is started. The firmware script typically creates several internal objects, such as: memory pools, SW queues, traffic manager, action profiles, pipelines, etc.
- `Device stop`: All the internal objects that were previously created by the firmware script during device start are now destroyed.
- `Device run`: Each Soft NIC device needs one or several CPU cores to run. The firmware script maps each internal pipeline to a CPU core. Multiple pipelines can be mapped to the same CPU core. In order for a given pipeline assigned to CPU core X to run, the application needs to periodically call on CPU core X the `rte_pmd_softnic_run()` function for the current Soft NIC device.
- `Application run`: The application reads packets from the Soft NIC device RX queues and writes packets to the Soft NIC device TX queues.

44.2 Supported Operating Systems

Any Linux distribution fulfilling the conditions described in `System Requirements` section of the DPDK documentation or refer to *DPDK Release Notes*.

44.3 Build options

The default PMD configuration available in the `common_linux` configuration file:

```
CONFIG_RTE_LIBRTE_PMD_SOFTNIC=y
```

Once the DPDK is built, all the DPDK applications include support for the Soft NIC PMD.

44.4 Soft NIC PMD arguments

The user can specify below arguments in EAL `--vdev` options to create the Soft NIC device instance:

```
-vdev "net_softnic0,firmware=firmware.cli,conn_port=8086"
```

1. `firmware`: path to the firmware script used for Soft NIC configuration. The example “firmware” script is provided at *drivers/net/softnic/*. (Optional: No, Default = NA)
2. `conn_port`: tcp connection port (non-zero value) used by remote client (for examples- telnet, netcat, etc.) to connect and configure Soft NIC device in run-time. (Optional: yes, Default value: 0, no connection with external client)
3. `cpu_id`: numa node id. (Optional: yes, Default value: 0)
4. `tm_n_queues`: number of traffic manager’s scheduler queues. The traffic manager is based on DPDK *librte_sched* library. (Optional: yes, Default value: 65,536 queues)
5. `tm_qsize0`: size of scheduler queue 0 per traffic class of the pipes/subscribers. (Optional: yes, Default: 64)
6. `tm_qsize1`: size of scheduler queue 1 per traffic class of the pipes/subscribers. (Optional: yes, Default: 64)
7. `tm_qsize2`: size of scheduler queue 2 per traffic class of the pipes/subscribers. (Optional: yes, Default: 64)
8. `tm_qsize3`: size of scheduler queue 3 per traffic class of the pipes/subscribers. (Optional: yes, Default: 64)

44.5 Soft NIC testing

- Run `testpmd` application in Soft NIC forwarding mode with loopback feature enabled on Soft NIC port:

```
./testpmd -c 0x3 --vdev 'net_softnic0,firmware=<script path>/firmware.cli,cpu_id=0,co
--forward-mode=softnic --portmask=0x2

...
Interactive-mode selected
Set softnic packet forwarding mode
```

```
...
Configuring Port 0 (socket 0)
Port 0: 90:E2:BA:37:9D:DC
Configuring Port 1 (socket 0)

; SPD-License-Identifier: BSD-3-Clause
; Copyright(c) 2018 Intel Corporation

link LINK dev 0000:02:00.0

pipeline RX period 10 offset_port_id 0
pipeline RX port in bsz 32 link LINK rxq 0
pipeline RX port out bsz 32 swq RXQ0
pipeline RX table match stub
pipeline RX port in 0 table 0

pipeline TX period 10 offset_port_id 0
pipeline TX port in bsz 32 swq TXQ0
pipeline TX port out bsz 32 link LINK txq 0
pipeline TX table match stub
pipeline TX port in 0 table 0

thread 1 pipeline RX enable
thread 1 pipeline TX enable
Port 1: 00:00:00:00:00:00
Checking link statuses...
Done
testpmd>
```

- Start forwarding

```
testpmd> start
softnic packet forwarding - ports=1 - cores=1 - streams=1 - NUMA support enabled, MP
Logical Core 1 (socket 0) forwards packets on 1 streams:
RX P=2/Q=0 (socket 0) -> TX P=2/Q=0 (socket 0) peer=02:00:00:00:00:02

softnic packet forwarding packets/burst=32
nb forwarding cores=1 - nb forwarding ports=1
port 0: RX queue number: 1 Tx queue number: 1
Rx offloads=0x1000 Tx offloads=0x0
RX queue: 0
RX desc=512 - RX free threshold=32
RX threshold registers: pthresh=8 hthresh=8 wthresh=0
RX Offloads=0x0
TX queue: 0
TX desc=512 - TX free threshold=32
TX threshold registers: pthresh=32 hthresh=0 wthresh=0
TX offloads=0x0 - TX RS bit threshold=32
port 1: RX queue number: 1 Tx queue number: 1
Rx offloads=0x0 Tx offloads=0x0
RX queue: 0
RX desc=0 - RX free threshold=0
RX threshold registers: pthresh=0 hthresh=0 wthresh=0
RX Offloads=0x0
TX queue: 0
TX desc=0 - TX free threshold=0
TX threshold registers: pthresh=0 hthresh=0 wthresh=0
TX offloads=0x0 - TX RS bit threshold=0
```

- Start remote client (e.g. telnet) to communicate with the softnic device:

```
$ telnet 127.0.0.1 8086
Trying 127.0.0.1...
Connected to 127.0.0.1.
```

```
Escape character is '^]'.
```

```
Welcome to Soft NIC!
```

```
softnic>
```

- Add/update Soft NIC pipeline table match-action entries from telnet client:

```
softnic> pipeline RX table 0 rule add match default action fwd port 0
softnic> pipeline TX table 0 rule add match default action fwd port 0
```

44.6 Soft NIC Firmware

The Soft NIC firmware, for example- *softnic/firmware.cli*, consists of following CLI commands for creating and managing software based NIC pipelines. For more details, please refer to CLI command description provided in *softnic/rte_eth_softnic_cli.c*.

- Physical port for packets send/receive:

```
link LINK dev 0000:02:00.0
```

- Pipeline create:

```
pipeline RX period 10 offset_port_id 0 (Soft NIC rx-path pipeline)
pipeline TX period 10 offset_port_id 0 (Soft NIC tx-path pipeline)
```

- Pipeline input/output port create

```
pipeline RX port in bsz 32 link LINK rxq 0 (Soft NIC rx pipeline input port)
pipeline RX port out bsz 32 swq RXQ0 (Soft NIC rx pipeline output port)
pipeline TX port in bsz 32 swq TXQ0 (Soft NIC tx pipeline input port)
pipeline TX port out bsz 32 link LINK txq 0 (Soft NIC tx pipeline output port)
```

- Pipeline table create

```
pipeline RX table match stub (Soft NIC rx pipeline match-action table)
pipeline TX table match stub (Soft NIC tx pipeline match-action table)
```

- Pipeline input port connection with table

```
pipeline RX port in 0 table 0 (Soft NIC rx pipeline input port 0 connection)
pipeline TX port in 0 table 0 (Soft NIC tx pipeline input port 0 connection)
```

- Pipeline table match-action rules add

```
pipeline RX table 0 rule add match default action fwd port 0 (Soft NIC rx pipe
pipeline TX table 0 rule add match default action fwd port 0 (Soft NIC tx pipe
```

- Enable pipeline on CPU thread

```
thread 1 pipeline RX enable (Soft NIC rx pipeline enable on cpu thread id 1)
thread 1 pipeline TX enable (Soft NIC tx pipeline enable on cpu thread id 1)
```

44.7 QoS API Support:

SoftNIC PMD implements ethdev traffic management APIs *rte_tm.h* that allow building and committing traffic manager hierarchy, configuring hierarchy nodes of the Quality of Service (QoS) scheduler supported by DPDK *librte_sched* library. Furthermore, APIs for run-time update to the traffic manager hierarchy are supported by PMD.

SoftNIC PMD also implements ethdev traffic metering and policing APIs `rte_mtr.h` that enables metering and marking of the packets with the appropriate color (green, yellow or red), according to the traffic metering algorithm. For the meter output color, policer actions like *keep the packet color same*, *change the packet color* or *drop the packet* can be configured.

Note: The SoftNIC does not support the meter object shared by several flows, thus only supports creating meter object private to the flow. Once meter object is successfully created, it can be linked to the specific flow by specifying the `meter` flow action in the flow rule.

44.8 Flow API support:

The SoftNIC PMD implements ethdev flow APIs `rte_flow.h` that allow validating flow rules, adding flow rules to the SoftNIC pipeline as table rules, deleting and querying the flow rules. The PMD provides new cli command for creating the flow group and their mapping to the SoftNIC pipeline and table. This cli should be configured as part of firmware file.

```
flowapi map group <group_id> ingress | egress pipeline <pipeline_name> \
table <table_id>
```

From the flow attributes of the flow, PMD uses the group id to get the mapped pipeline and table. PMD supports number of flow actions such as `JMP`, `QUEUE`, `RSS`, `DROP`, `COUNT`, `METER`, `VXLAN` etc.

Note: The flow must have one terminating actions i.e. `JMP` or `RSS` or `QUEUE` or `DROP`. For the count and drop actions the underlying PMD doesn't support the functionality yet. So it is not recommended for use.

The flow API can be tested with the help of `testpmd` application. The SoftNIC firmware specifies CLI commands for port configuration, pipeline creation, action profile creation and table creation. Once application gets initialized, the flow rules can be added through the `testpmd` CLI. The PMD will translate the flow rules to the SoftNIC pipeline tables rules.

44.8.1 Example:

Example demonstrates the flow queue action using the SoftNIC firmware and `testpmd` commands.

- Prepare SoftNIC firmware

```
link LINK0 dev 0000:83:00.0
link LINK1 dev 0000:81:00.0
pipeline RX period 10 offset_port_id 0
pipeline RX port in bsz 32 link LINK0 rxq 0
pipeline RX port in bsz 32 link LINK1 rxq 0
pipeline RX port out bsz 32 swq RXQ0
pipeline RX port out bsz 32 swq RXQ1
table action profile AP0 ipv4 offset 278 fwd
pipeline RX table match hash ext key 16 mask
    00FF0000FFFFFFFFFFFFFFFFFFFFFFFF \
    offset 278 buckets 16K size 65K action AP0
pipeline RX port in 0 table 0
pipeline RX port in 1 table 0
flowapi map group 0 ingress pipeline RX table 0
pipeline TX period 10 offset_port_id 0
```

```

pipeline TX port in bsz 32 swq TXQ0
pipeline TX port in bsz 32 swq TXQ1
pipeline TX port out bsz 32 link LINK0 txq 0
pipeline TX port out bsz 32 link LINK1 txq 0
pipeline TX table match hash ext key 16 mask
    00FF0000FFFFFFFFFFFFFFFFFFFFFFFF \
    offset 278 buckets 16K size 65K action AP0
pipeline TX port in 0 table 0
pipeline TX port in 1 table 0
pipeline TX table 0 rule add match hash ipv4_5tuple
    1.10.11.12 2.20.21.22 100 200 6 action fwd port 0
pipeline TX table 0 rule add match hash ipv4_5tuple
    1.10.11.13 2.20.21.23 100 200 6 action fwd port 1
thread 25 pipeline RX enable
thread 25 pipeline TX enable

```

- Run testpmd:

```

./x86_64-native-linux-gcc/app/testpmd -l 23-25 -n 4 \
    --vdev 'net_softnic0, \
    firmware=./drivers/net/softnic/ \
    firmware.cli, \
    cpu_id=1,conn_port=8086' -- \
    -i --forward-mode=softnic --rxq=2, \
    --txq=2, --disable-rss --portmask=0x4

```

- Configure flow rules on softnic:

```

flow create 2 group 0 ingress pattern eth / ipv4 proto mask 255 src \
    mask 255.255.255.255 dst mask 255.255.255.255 src spec
    1.10.11.12 dst spec 2.20.21.22 proto spec 6 / tcp src mask 65535 \
    dst mask 65535 src spec 100 dst spec 200 / end actions queue \
    index 0 / end
flow create 2 group 0 ingress pattern eth / ipv4 proto mask 255 src \
    mask 255.255.255.255 dst mask 255.255.255.255 src spec 1.10.11.13 \
    dst spec 2.20.21.23 proto spec 6 / tcp src mask 65535 dst mask \
    65535 src spec 100 dst spec 200 / end actions queue index 1 / end

```

SZEDATA2 POLL MODE DRIVER LIBRARY

The SZEDATA2 poll mode driver library implements support for the Netcope FPGA Boards (**NFB-40G2**, **NFB-100G2**, **NFB-200G2QL**) and Silicom **FB2CGG3** card, FPGA-based programmable NICs. The SZEDATA2 PMD uses interface provided by the libsize2 library to communicate with the NFB cards over the size2 layer.

More information about the [NFB cards](#) and used technology ([Netcope Development Kit](#)) can be found on the [Netcope Technologies website](#).

Note: This driver has external dependencies. Therefore it is disabled in default configuration files. It can be enabled by setting `CONFIG_RTE_LIBRTE_PMD_SZEDATA2=y` and recompiling.

Note: Currently the driver is supported only on x86_64 architectures. Only x86_64 versions of the external libraries are provided.

45.1 Prerequisites

This PMD requires kernel modules which are responsible for initialization and allocation of resources needed for size2 layer function. Communication between PMD and kernel modules is mediated by libsize2 library. These kernel modules and library are not part of DPDK and must be installed separately:

- **libsize2 library**

The library provides API for initialization of size2 transfers, receiving and transmitting data segments.

- **Kernel modules**

- combo6core
- combov3
- szedata2
- szedata2_cv3 or szedata2_cv3_fdt

Kernel modules manage initialization of hardware, allocation and sharing of resources for user space applications.

Information about getting the dependencies can be found [here](#).

45.1.1 Versions of the packages

The minimum version of the provided packages:

- for DPDK from 18.05: **4.4.1**
- for DPDK up to 18.02 (including): **3.0.5**

45.2 Configuration

These configuration options can be modified before compilation in the `.config` file:

- `CONFIG_RTE_LIBRTE_PMD_SZEDATA2` default value: **n**

Value **y** enables compilation of `szedata2` PMD.

45.3 Using the SZEDATA2 PMD

From DPDK version 16.04 the type of `SZEDATA2` PMD is changed to `PMD_PDEV`. `SZEDATA2` device is automatically recognized during EAL initialization. No special command line options are needed.

Kernel modules have to be loaded before running the DPDK application.

45.4 NFB card architecture

The NFB cards are multi-port multi-queue cards, where (generally) data from any Ethernet port may be sent to any queue. They were historically represented in DPDK as a single port.

However, the new NFB-200G2QL card employs an add-on cable which allows to connect it to two physical PCI-E slots at the same time (see the diagram below). This is done to allow 200 Gbps of traffic to be transferred through the PCI-E bus (note that a single PCI-E 3.0 x16 slot provides only 125 Gbps theoretical throughput).

Since each slot may be connected to a different CPU and therefore to a different NUMA node, the card is represented as two ports in DPDK (each with half of the queues), which allows DPDK to work with data from the individual queues on the right NUMA node.

45.5 Limitations

The `SZEDATA2` PMD does not support operations related to Ethernet ports (`link_up`, `link_down`, `set_mac_address`, etc.).

NFB cards employ multiple Ethernet ports. Until now, Ethernet port-related operations were performed on all of them (since the whole card was represented as a single port). With NFB-200G2QL card, this is no longer viable (see above).

Since there is no fixed mapping between the queues and Ethernet ports, and since a single card can be represented as two ports in DPDK, there is no way of telling which (if any) physical ports should be associated with individual ports in DPDK.

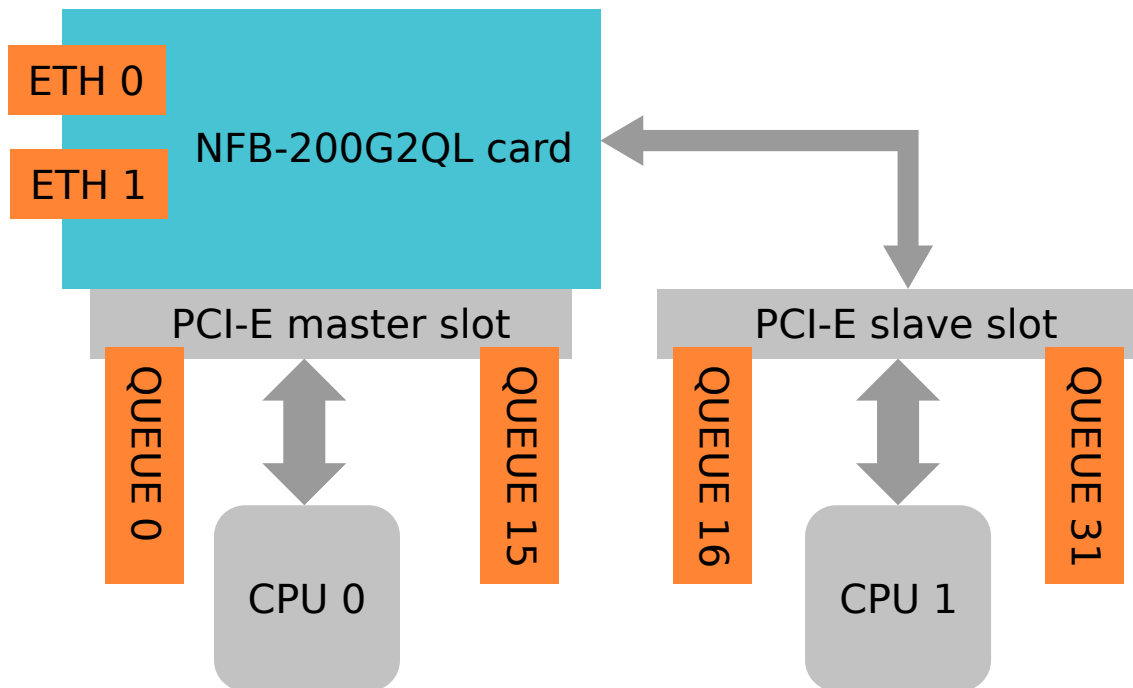


Fig. 45.1: NFB-200G2QL high-level diagram

45.6 Example of usage

Read packets from 0. and 1. receive channel and write them to 0. and 1. transmit channel:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 2 \
-- --port-topology=chained --rxq=2 --txq=2 --nb-cores=2 -i -a
```

Example output:

```
[...]
EAL: PCI device 0000:06:00.0 on NUMA socket -1
EAL: probe driver: 1b26:c1c1 rte_szedata2_pmd
PMD: Initializing szedata2 device (0000:06:00.0)
PMD: SZEDATA2 path: /dev/szedataII0
PMD: Available DMA channels RX: 8 TX: 8
PMD: resource0 phys_addr = 0xe8000000 len = 134217728 virt addr = 7f48f8000000
PMD: szedata2 device (0000:06:00.0) successfully initialized
Interactive-mode selected
Auto-start selected
Configuring Port 0 (socket 0)
Port 0: 00:11:17:00:00:00
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Done
Start automatic packet forwarding
io packet forwarding - CRC stripping disabled - packets/burst=32
nb forwarding cores=2 - nb forwarding ports=1
RX queues=2 - RX desc=128 - RX free threshold=0
RX threshold registers: pthresh=0 hthresh=0 wthresh=0
TX queues=2 - TX desc=512 - TX free threshold=0
TX threshold registers: pthresh=0 hthresh=0 wthresh=0
TX RS bit threshold=0 - TXQ flags=0x0
testpmd>
```

TUN|TAP POLL MODE DRIVER

The `rte_eth_tap.c` PMD creates a device using TAP interfaces on the local host. The PMD allows for DPDK and the host to communicate using a raw device interface on the host and in the DPDK application.

The device created is a TAP device, which sends/receives packet in a raw format with a L2 header. The usage for a TAP PMD is for connectivity to the local host using a TAP interface. When the TAP PMD is initialized it will create a number of tap devices in the host accessed via `ifconfig -a` or `ip` command. The commands can be used to assign and query the virtual like device.

These TAP interfaces can be used with Wireshark or tcpdump or Pktgen-DPDK along with being able to be used as a network connection to the DPDK application. The method enable one or more interfaces is to use the `--vdev=net_tap0` option on the DPDK application command line. Each `--vdev=net_tap1` option given will create an interface named `dtap0`, `dtap1`, and so on.

The interface name can be changed by adding the `iface=foo0`, for example:

```
--vdev=net_tap0,iface=foo0 --vdev=net_tap1,iface=fool, ...
```

Normally the PMD will generate a random MAC address, but when testing or with a static configuration the developer may need a fixed MAC address style. Using the option `mac=fixed` you can create a fixed known MAC address:

```
--vdev=net_tap0,mac=fixed
```

The MAC address will have a fixed value with the last octet incrementing by one for each interface string containing `mac=fixed`. The MAC address is formatted as `00:'d':'t':'a':'p':[00-FF]`. Convert the characters to hex and you get the actual MAC address: `00:64:74:61:70:[00-FF]`.

```
-vdev=net_tap0,mac="00:64:74:61:70:11"
```

The MAC address will have a user value passed as string. The MAC address is in format with delimiter `:.:`. The string is byte converted to hex and you get the actual MAC address: `00:64:74:61:70:11`.

It is possible to specify a remote netdevice to capture packets from by adding `remote=fool`, for example:

```
--vdev=net_tap,iface=tap0,remote=fool
```

If a `remote` is set, the tap MAC address will be set to match the remote one just after netdevice creation. Using TC rules, traffic from the remote netdevice will be redirected to the tap. If the tap is in promiscuous mode, then all packets will be redirected. In `allmulti` mode, all multicast packets will be redirected.

Using the remote feature is especially useful for capturing traffic from a netdevice that has no support in the DPDK. It is possible to add explicit `rte_flow` rules on the tap PMD to capture specific traffic (see next section for examples).

After the DPDK application is started you can send and receive packets on the interface using the standard `rx_burst/tx_burst` APIs in DPDK. From the host point of view you can use any host tool like `tcpdump`, `Wireshark`, `ping`, `Pktgen` and others to communicate with the DPDK application. The DPDK application may not understand network protocols like IPv4/6, UDP or TCP unless the application has been written to understand these protocols.

If you need the interface as a real network interface meaning running and has a valid IP address then you can do this with the following commands:

```
sudo ip link set dtap0 up; sudo ip addr add 192.168.0.250/24 dev dtap0
sudo ip link set dtap1 up; sudo ip addr add 192.168.1.250/24 dev dtap1
```

Please change the IP addresses as you see fit.

If routing is enabled on the host you can also communicate with the DPDK App over the internet via a standard socket layer application as long as you account for the protocol handling in the application.

If you have a Network Stack in your DPDK application or something like it you can utilize that stack to handle the network protocols. Plus you would be able to address the interface using an IP address assigned to the internal interface.

The TUN PMD allows user to create a TUN device on host. The PMD allows user to transmit and receive packets via DPDK API calls with L3 header and payload. The devices in host can be accessed via `ifconfig` or `ip` command. TUN interfaces are passed to DPDK `rte_eal_init` arguments as `--vdev=net_tunX`, where X stands for unique id, example:

```
--vdev=net_tun0 --vdev=net_tun1,iface=fool, ...
```

Unlike TAP PMD, TUN PMD does not support user arguments as `MAC` or `remote` user options. Default interface name is `dtunX`, where X stands for unique id.

46.1 Flow API support

The tap PMD supports major flow API pattern items and actions, when running on linux kernels above 4.2 ("Flower" classifier required). The kernel support can be checked with this command:

```
zcat /proc/config.gz | ( grep 'CLS_FLOWER=' || echo 'not supported' ) |
tee -a /dev/stderr | grep -q '=m' &&
lsmod | ( grep cls_flower || echo 'try modprobe cls_flower' )
```

Supported items:

- `eth`: `src` and `dst` (with variable masks), and `eth_type` (0xffff mask).
- `vlan`: `vid`, `pcp`, but not `eid`. (requires kernel 4.9)
- `ipv4/6`: `src` and `dst` (with variable masks), and `ip_proto` (0xffff mask).
- `udp/tcp`: `src` and `dst` port (0xffff) mask.

Supported actions:

- `DROP`
- `QUEUE`
- `PASSTHRU`
- `RSS` (requires kernel 4.9)

It is generally not possible to provide a “last” item. However, if the “last” item, once masked, is identical to the masked spec, then it is supported.

Only IPv4/6 and MAC addresses can use a variable mask. All other items need a full mask (exact match).

As rules are translated to TC, it is possible to show them with something like:

```
tc -s filter show dev tap1 parent 1:
```

46.1.1 Examples of testpmd flow rules

Drop packets for destination IP 192.0.2.1:

```
testpmd> flow create 0 priority 1 ingress pattern eth / ipv4 dst is 192.0.2.1 \
/ end actions drop / end
```

Ensure packets from a given MAC address are received on a queue 2:

```
testpmd> flow create 0 priority 2 ingress pattern eth src is 06:05:04:03:02:01 \
/ end actions queue index 2 / end
```

Drop UDP packets in vlan 3:

```
testpmd> flow create 0 priority 3 ingress pattern eth / vlan vid is 3 / \
ipv4 proto is 17 / end actions drop / end
```

Distribute IPv4 TCP packets using RSS to a given MAC address over queues 0-3:

```
testpmd> flow create 0 priority 4 ingress pattern eth dst is 0a:0b:0c:0d:0e:0f \
/ ipv4 / tcp / end actions rss queues 0 1 2 3 end / end
```

46.2 Multi-process sharing

It is possible to attach an existing TAP device in a secondary process, by declaring it as a vdev with the same name as in the primary process, and without any parameter.

The port attached in a secondary process will give access to the statistics and the queues. Therefore it can be used for monitoring or Rx/Tx processing.

The IPC synchronization of Rx/Tx queues is currently limited:

- Maximum 8 queues shared
- Synchronized on probing, but not on later port update

46.3 Example

The following is a simple example of using the TAP PMD with the Pktgen packet generator. It requires that the `socat` utility is installed on the test system.

Build DPDK, then pull down Pktgen and build `pktgen` using the DPDK SDK/Target used to build the `dpdk` you pulled down.

Run `pktgen` from the `pktgen` directory in a terminal with a commandline like the following:

```

sudo ./app/app/x86_64-native-linux-gcc/app/pktgen -l 1-5 -n 4          \
--proc-type auto --log-level debug --socket-mem 512,512 --file-prefix pg \
--vdev=net_tap0 --vdev=net_tap1 -b 05:00.0 -b 05:00.1              \
-b 04:00.0 -b 04:00.1 -b 04:00.2 -b 04:00.3                      \
-b 81:00.0 -b 81:00.1 -b 81:00.2 -b 81:00.3                      \
-b 82:00.0 -b 83:00.0 -- -T -P -m [2:3].0 -m [4:5].1             \
-f themes/black-yellow.theme

```

Verify with `ifconfig -a` command in a different xterm window, should have a `dtap0` and `dtap1` interfaces created.

Next set the links for the two interfaces to up via the commands below:

```

sudo ip link set dtap0 up; sudo ip addr add 192.168.0.250/24 dev dtap0
sudo ip link set dtap1 up; sudo ip addr add 192.168.1.250/24 dev dtap1

```

Then use `socat` to create a loopback for the two interfaces:

```

sudo socat interface:dtap0 interface:dtap1

```

Then on the Pktgen command line interface you can start sending packets using the commands `start 0` and `start 1` or you can start both at the same time with `start all`. The command `str` is an alias for `start all` and `stp` is an alias for `stop all`.

While running you should see the 64 byte counters increasing to verify the traffic is being looped back. You can use `set all size XXX` to change the size of the packets after you stop the traffic. Use `pktgen help` command to see a list of all commands. You can also use the `-f` option to load commands at startup in command line or Lua script in `pktgen`.

46.4 RSS specifics

Packet distribution in TAP is done by the kernel which has a default distribution. This feature is adding RSS distribution based on eBPF code. The default eBPF code calculates RSS hash based on Toeplitz algorithm for a fixed RSS key. It is calculated on fixed packet offsets. For IPv4 and IPv6 it is calculated over src/dst addresses (8 or 32 bytes for IPv4 or IPv6 respectively) and src/dst TCP/UDP ports (4 bytes).

The RSS algorithm is written in file `tap_bpf_program.c` which does not take part in TAP PMD compilation. Instead this file is compiled in advance to eBPF object file. The eBPF object file is then parsed and translated into eBPF byte code in the format of C arrays of eBPF instructions. The C array of eBPF instructions is part of TAP PMD tree and is taking part in TAP PMD compilation. At run time the C arrays are uploaded to the kernel via BPF system calls and the RSS hash is calculated by the kernel.

It is possible to support different RSS hash algorithms by updating file `tap_bpf_program.c`. In order to add a new RSS hash algorithm follow these steps:

1. Write the new RSS implementation in file `tap_bpf_program.c`

BPF programs which are uploaded to the kernel correspond to C functions under different ELF sections.

2. Install LLVM library and clang compiler versions 3.7 and above
3. Compile `tap_bpf_program.c` via LLVM into an object file:

```

clang -O2 -emit-llvm -c tap_bpf_program.c -o - | llc -march=bpf \
-filetype=obj -o <tap_bpf_program.o>

```

4. Use a tool that receives two parameters: an eBPF object file and a section name, and prints out the section as a C array of eBPF instructions. Embed the C array in your TAP PMD tree.

The C arrays are uploaded to the kernel using BPF system calls.

`tc` (traffic control) is a well known user space utility program used to configure the Linux kernel packet scheduler. It is usually packaged as part of the `iproute2` package. Since commit `11c39b5e9` (“`tc: add eBPF support to f_bpf`”) `tc` can be used to uploads eBPF code to the kernel and can be patched in order to print the C arrays of eBPF instructions just before calling the BPF system call. Please refer to `iproute2` package file `lib/bpf.c` function `bpf_prog_load()`.

An example utility for eBPF instruction generation in the format of C arrays will be added in next releases

TAP reports on supported RSS functions as part of `dev_infos_get` callback: `ETH_RSS_IP`, `ETH_RSS_UDP` and `ETH_RSS_TCP`. **Known limitation:** TAP supports all of the above hash functions together and not in partial combinations.

46.5 Systems supporting flow API

- “tc flower” classifier requires linux kernel above 4.2
- eBPF/RSS requires linux kernel above 4.9

RH7.3	No flow rule support
RH7.4	No RSS action support
RH7.5	No RSS action support
SLES 15, kernel 4.12	No limitation
Azure Ubuntu 16.04, kernel 4.13	No limitation

THUNDERX NICVF POLL MODE DRIVER

The ThunderX NICVF PMD (`librte_pmd_thunderx_nicvf`) provides poll mode driver support for the inbuilt NIC found in the **Cavium ThunderX** SoC family as well as their virtual functions (VF) in SR-IOV context.

More information can be found at [Cavium, Inc Official Website](#).

47.1 Features

Features of the ThunderX PMD are:

- Multiple queues for TX and RX
- Receive Side Scaling (RSS)
- Packet type information
- Checksum offload
- Promiscuous mode
- Multicast mode
- Port hardware statistics
- Jumbo frames
- Link state information
- Scattered and gather for TX and RX
- VLAN stripping
- SR-IOV VF
- NUMA support
- Multi queue set support (up to 96 queues (12 queue sets)) per port
- Skip data bytes

47.2 Supported ThunderX SoCs

- CN88xx
- CN81xx

- CN83xx

47.3 Prerequisites

- Follow the DPDK Getting Started Guide for Linux to setup the basic DPDK environment.

47.4 Pre-Installation Configuration

47.4.1 Config File Options

The following options can be modified in the `config` file. Please note that enabling debugging options may affect system performance.

- `CONFIG_RTE_LIBRTE_THUNDERX_NICVF_PMD` (default `y`)
Toggle compilation of the `librte_pmd_thunderx_nicvf` driver.
- `CONFIG_RTE_LIBRTE_THUNDERX_NICVF_DEBUG_RX` (default `n`)
Toggle asserts of receive fast path.
- `CONFIG_RTE_LIBRTE_THUNDERX_NICVF_DEBUG_TX` (default `n`)
Toggle asserts of transmit fast path.

47.5 Driver compilation and testing

Refer to the document *compiling and testing a PMD for a NIC* for details.

To compile the ThunderX NICVF PMD for Linux arm64 gcc, use `arm64-thunderx-linux-gcc` as target.

47.6 Linux

47.6.1 SR-IOV: Prerequisites and sample Application Notes

Current ThunderX NIC PF/VF kernel modules maps each physical Ethernet port automatically to virtual function (VF) and presented them as PCIe-like SR-IOV device. This section provides instructions to configure SR-IOV with Linux OS.

1. Verify PF devices capabilities using `lspci`:

```
lspci -vvv
```

Example output:

```
0002:01:00.0 Ethernet controller: Cavium Networks Device a01e (rev 01)
...
Capabilities: [100 v1] Alternative Routing-ID Interpretation (ARI)
...
Capabilities: [180 v1] Single Root I/O Virtualization (SR-IOV)
...
Kernel driver in use: thunder-nic
...
```

Note: Unless thunder-nic driver is in use make sure your kernel config includes CONFIG_THUNDER_NIC_PF setting.

2. Verify VF devices capabilities and drivers using lspci:

```
lspci -vvv
```

Example output:

```
0002:01:00.1 Ethernet controller: Cavium Networks Device 0011 (rev 01)
...
Capabilities: [100 v1] Alternative Routing-ID Interpretation (ARI)
...
Kernel driver in use: thunder-nicvf
...

0002:01:00.2 Ethernet controller: Cavium Networks Device 0011 (rev 01)
...
Capabilities: [100 v1] Alternative Routing-ID Interpretation (ARI)
...
Kernel driver in use: thunder-nicvf
...
```

Note: Unless thunder-nicvf driver is in use make sure your kernel config includes CONFIG_THUNDER_NIC_VF setting.

3. Pass VF device to VM context (PCIe Passthrough):

The VF devices may be passed through to the guest VM using qemu or virt-manager or virsh etc.

Example qemu guest launch command:

```
sudo qemu-system-aarch64 -name vm1 \
-machine virt,gic_version=3,accel=kvm,usb=off \
-cpu host -m 4096 \
-smp 4,sockets=1,cores=8,threads=1 \
-nographic -nodefaults \
-kernel <kernel image> \
-append "root=/dev/vda console=ttyAMA0 rw hugepagesz=512M hugepages=3" \
-device vfio-pci,host=0002:01:00.1 \
-drive file=<rootfs.ext3>,if=none,id=disk1,format=raw \
-device virtio-blk-device,scsi=off,drive=disk1,id=virtio-disk1,bootindex=1 \
-netdev tap,id=net0,ifname=tap0,script=/etc/qemu-ifup_thunder \
-device virtio-net-device,netdev=net0 \
-serial stdio \
-mem-path /dev/huge
```

4. Enable VFIO-NOIOMMU mode (optional):

```
echo 1 > /sys/module/vfio/parameters/enable_unsafe_noiommu_mode
```

Note: VFIO-NOIOMMU is required only when running in VM context and should not be enabled otherwise.

5. Running testpmd:

Follow instructions available in the document *compiling and testing a PMD for a NIC* to run testpmd.

Example output:

```
./arm64-thunderx-linux-gcc/app/testpmd -l 0-3 -n 4 -w 0002:01:00.2 \
-- -i --no-flush-rx \
--port-topology=loop

...

PMD: rte_nicvf_pmd_init(): librte_pmd_thunderx nicvf version 1.0

...
EAL: probe driver: 177d:11 rte_nicvf_pmd
EAL: using IOMMU type 1 (Type 1)
EAL: PCI memory mapped at 0x3ffade50000
EAL: Trying to map BAR 4 that contains the MSI-X table.
      Trying offsets: 0x40000000000:0x0000, 0x10000:0x1f0000
EAL: PCI memory mapped at 0x3ffadc60000
PMD: nicvf_eth_dev_init(): nicvf: device (177d:11) 2:1:0:2
PMD: nicvf_eth_dev_init(): node=0 vf=1 mode=tns-bypass sqs=false
      loopback_supported=true
PMD: nicvf_eth_dev_init(): Port 0 (177d:11) mac=a6:c6:d9:17:78:01
Interactive-mode selected
Configuring Port 0 (socket 0)
...

PMD: nicvf_dev_configure(): Configured ethdev port0 hwcap=0x0
Port 0: A6:C6:D9:17:78:01
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd>
```

47.6.2 Multiple Queue Set per DPDK port configuration

There are two types of VFs:

- Primary VF
- Secondary VF

Each port consists of a primary VF and *n* secondary VF(s). Each VF provides 8 Tx/Rx queues to a port. When a given port is configured to use more than 8 queues, it requires one (or more) secondary VF. Each secondary VF adds 8 additional queues to the queue set.

During PMD driver initialization, the primary VF's are enumerated by checking the specific flag (see sqs message in DPDK boot log - sqs indicates secondary queue set). They are at the beginning of VF list (the remain ones are secondary VF's).

The primary VFs are used as master queue sets. Secondary VFs provide additional queue sets for primary ones. If a port is configured for more than 8 queues than it will request for additional queues from secondary VFs.

Secondary VFs cannot be shared between primary VFs.

Primary VFs are present on the beginning of the 'Network devices using kernel driver' list, secondary VFs are on the remaining on the remaining part of the list.

Note: The VNIC driver in the multiqueue setup works differently than other drivers like *ixgbe*. We need to bind separately each specific queue set device with the

usertools/dpdk-devbind.py utility.

Note: Depending on the hardware used, the kernel driver sets a threshold `vf_id`. VFs that try to attach with an id below or equal to this boundary are considered primary VFs. VFs that try to attach with an id above this boundary are considered secondary VFs.

47.6.3 Example device binding

If a system has three interfaces, a total of 18 VF devices will be created on a non-NUMA machine.

Note: NUMA systems have 12 VFs per port and non-NUMA 6 VFs per port.

```
# usertools/dpdk-devbind.py --status

Network devices using DPDK-compatible driver
=====
<none>

Network devices using kernel driver
=====
0000:01:10.0 'Device a026' if= drv=thunder-BGX unused=vfio-pci,uio_pci_generic
0000:01:10.1 'Device a026' if= drv=thunder-BGX unused=vfio-pci,uio_pci_generic
0002:01:00.0 'Device a01e' if= drv=thunder-nic unused=vfio-pci,uio_pci_generic
0002:01:00.1 'Device 0011' if=eth0 drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:00.2 'Device 0011' if=eth1 drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:00.3 'Device 0011' if=eth2 drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:00.4 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:00.5 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:00.6 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:00.7 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:01.0 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:01.1 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:01.2 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:01.3 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:01.4 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:01.5 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:01.6 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:01.7 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:02.0 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:02.1 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_generic
0002:01:02.2 'Device 0011' if= drv=thunder-nicvf unused=vfio-pci,uio_pci_generic

Other network devices
=====
0002:00:03.0 'Device a01f' unused=vfio-pci,uio_pci_generic
```

We want to bind two physical interfaces with 24 queues each device, we attach two primary VFs and four secondary queues. In our example we choose two 10G interfaces eth1 (0002:01:00.2) and eth2 (0002:01:00.3). We will choose four secondary queue sets from the ending of the list (0002:01:01.7-0002:01:02.2).

1. Bind two primary VFs to the `vfio-pci` driver:

```
usertools/dpdk-devbind.py -b vfio-pci 0002:01:00.2
usertools/dpdk-devbind.py -b vfio-pci 0002:01:00.3
```

2. Bind four primary VFs to the `vfio-pci` driver:

```
usertools/dpdk-devbind.py -b vfio-pci 0002:01:01.7
usertools/dpdk-devbind.py -b vfio-pci 0002:01:02.0
usertools/dpdk-devbind.py -b vfio-pci 0002:01:02.1
usertools/dpdk-devbind.py -b vfio-pci 0002:01:02.2
```

The `nicvf thunderx` driver will make use of attached secondary VFs automatically during the interface configuration stage.

47.7 Module params

47.7.1 `skip_data_bytes`

This feature is used to create a hole between HEADROOM and actual data. Size of hole is specified in bytes as module param(“`skip_data_bytes`”) to pmd. This scheme is useful when application would like to insert vlan header without disturbing HEADROOM.

Example:

```
-w 0002:01:00.2,skip_data_bytes=8
```

47.8 Limitations

47.8.1 CRC stripping

The ThunderX SoC family NICs strip the CRC for every packets coming into the host interface irrespective of the offload configuration.

47.8.2 Maximum packet length

The ThunderX SoC family NICs support a maximum of a 9K jumbo frame. The value is fixed and cannot be changed. So, even when the `rxmode.max_rx_pkt_len` member of `struct rte_eth_conf` is set to a value lower than 9200, frames up to 9200 bytes can still reach the host interface.

47.8.3 Maximum packet segments

The ThunderX SoC family NICs support up to 12 segments per packet when working in scatter/gather mode. So, setting MTU will result with `EINVAL` when the frame size does not fit in the maximum number of segments.

47.8.4 `skip_data_bytes`

Maximum limit of `skip_data_bytes` is 128 bytes and number of bytes should be multiple of 8.

VDEV_NETVSC DRIVER

The VDEV_NETVSC driver (`librte_pmd_vdev_netvsc`) provides support for NetVSC interfaces and associated SR-IOV virtual function (VF) devices found in Linux virtual machines running on Microsoft Hyper-V (including Azure) platforms.

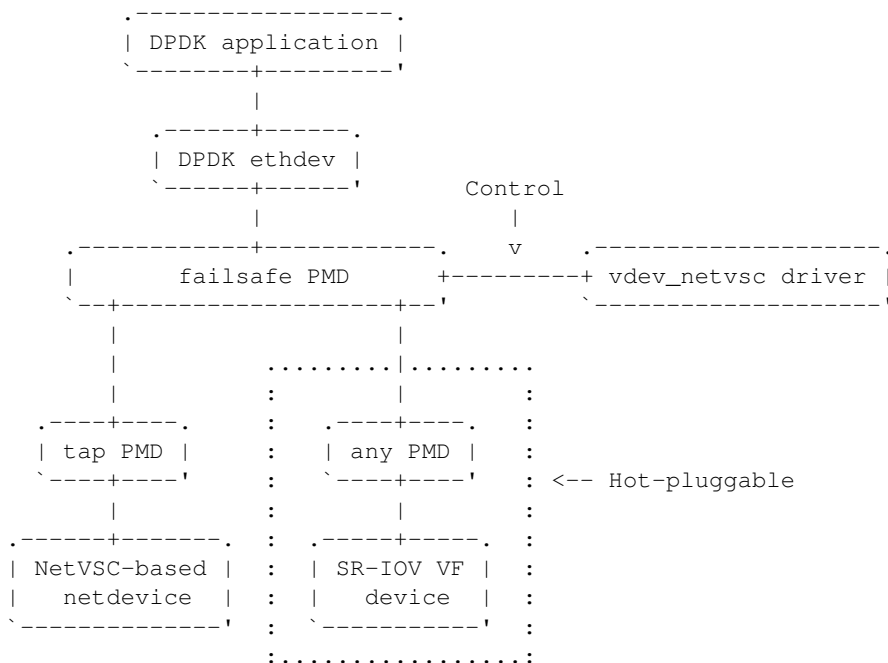
48.1 Implementation details

Each instance of this driver effectively needs to drive two devices: the NetVSC interface proper and its SR-IOV VF (referred to as “physical” from this point on) counterpart sharing the same MAC address.

Physical devices are part of the host system and cannot be maintained during VM migration. From a VM standpoint they appear as hot-plug devices that come and go without prior notice.

When the physical device is present, egress and most of the ingress traffic flows through it; only multi-casts and other hypervisor control still flow through NetVSC. Otherwise, NetVSC acts as a fallback for all traffic.

To avoid unnecessary code duplication and ensure maximum performance, handling of physical devices is left to their original PMDs; this virtual device driver (also known as *vdev*) manages other PMDs as summarized by the following block diagram:



This driver implementation may be temporary and should be improved or removed either when hot-plug will be fully supported in EAL and bus drivers or when a new NetVSC driver will be integrated.

48.2 Build options

- `CONFIG_RTE_LIBRTE_VDEV_NETVSC_PMD` (default `y`)

Toggle compilation of this driver.

48.3 Run-time parameters

This driver is invoked automatically in Hyper-V VM systems unless the user invoked it by command line using `--vdev=net_vdev_netvsc` EAL option.

The following device parameters are supported:

- `iface` [string]
Provide a specific NetVSC interface (netdevice) name to attach this driver to. Can be provided multiple times for additional instances.
- `mac` [string]
Same as `iface` except a suitable NetVSC interface is located using its MAC address.
- `force` [int]
If nonzero, forces the use of specified interfaces even if not detected as NetVSC.
- `ignore` [int]
If nonzero, ignores the driver running (actually used to disable the auto-detection in Hyper-V VM).

Note: Not specifying either `iface` or `mac` makes this driver attach itself to all unrouted NetVSC interfaces found on the system. Specifying the device makes this driver attach itself to the device regardless the device routes.

POLL MODE DRIVER FOR EMULATED VIRTIO NIC

Virtio is a para-virtualization framework initiated by IBM, and supported by KVM hypervisor. In the Data Plane Development Kit (DPDK), we provide a virtio Poll Mode Driver (PMD) as a software solution, comparing to SRIOV hardware solution, for fast guest VM to guest VM communication and guest VM to host communication.

Vhost is a kernel acceleration module for virtio qemu backend. The DPDK extends kni to support vhost raw socket interface, which enables vhost to directly read/ write packets from/to a physical port. With this enhancement, virtio could achieve quite promising performance.

For basic qemu-KVM installation and other Intel EM poll mode driver in guest VM, please refer to Chapter “Driver for VM Emulated Devices”.

In this chapter, we will demonstrate usage of virtio PMD driver with two backends, standard qemu vhost back end and vhost kni back end.

49.1 Virtio Implementation in DPDK

For details about the virtio spec, refer to the latest [VIRTIO \(Virtual I/O\) Device Specification](#).

As a PMD, virtio provides packet reception and transmission callbacks.

In Rx, packets described by the used descriptors in vring are available for virtio to burst out.

In Tx, packets described by the used descriptors in vring are available for virtio to clean. Virtio will enqueue to be transmitted packets into vring, make them available to the device, and then notify the host back end if necessary.

49.2 Features and Limitations of virtio PMD

In this release, the virtio PMD driver provides the basic functionality of packet reception and transmission.

- It supports merge-able buffers per packet when receiving packets and scattered buffer per packet when transmitting packets. The packet size supported is from 64 to 1518.
- It supports multicast packets and promiscuous mode.
- The descriptor number for the Rx/Tx queue is hard-coded to be 256 by qemu 2.7 and below. If given a different descriptor number by the upper application, the virtio PMD generates a warning and fall back to the hard-coded value. Rx queue size can be configurable and up to 1024 since qemu 2.8 and above. Rx queue size is 256 by default. Tx queue size is still hard-coded to be 256.

- Features of mac/vlan filter are supported, negotiation with vhost/backend are needed to support them. When backend can't support vlan filter, virtio app on guest should not enable vlan filter in order to make sure the virtio port is configured correctly. E.g. do not specify '-enable-hw-vlan' in testpmd command line. Note that, mac/vlan filter is best effort: unwanted packets could still arrive.
- "RTE_PKTMBUF_HEADROOM" should be defined no less than "sizeof(struct virtio_net_hdr_mrg_rxbuf)", which is 12 bytes when mergeable or "VIRTIO_F_VERSION_1" is set. no less than "sizeof(struct virtio_net_hdr)", which is 10 bytes, when using non-mergeable.
- Virtio does not support runtime configuration.
- Virtio supports Link State interrupt.
- Virtio supports Rx interrupt (so far, only support 1:1 mapping for queue/interrupt).
- Virtio supports software vlan stripping and inserting.
- Virtio supports using port IO to get PCI resource when uio/igb_uio module is not available.

49.3 Prerequisites

The following prerequisites apply:

- In the BIOS, turn VT-x and VT-d on
- Linux kernel with KVM module; vhost module loaded and ioeventfd supported. Qemu standard backend without vhost support isn't tested, and probably isn't supported.

49.4 Virtio with kni vhost Back End

This section demonstrates kni vhost back end example setup for Phy-VM Communication.

Host2VM communication example

1. Load the kni kernel module:

```
insmod rte_kni.ko
```

Other basic DPDK preparations like hugepage enabling, uio port binding are not listed here. Please refer to the *DPDK Getting Started Guide* for detailed instructions.

2. Launch the kni user application:

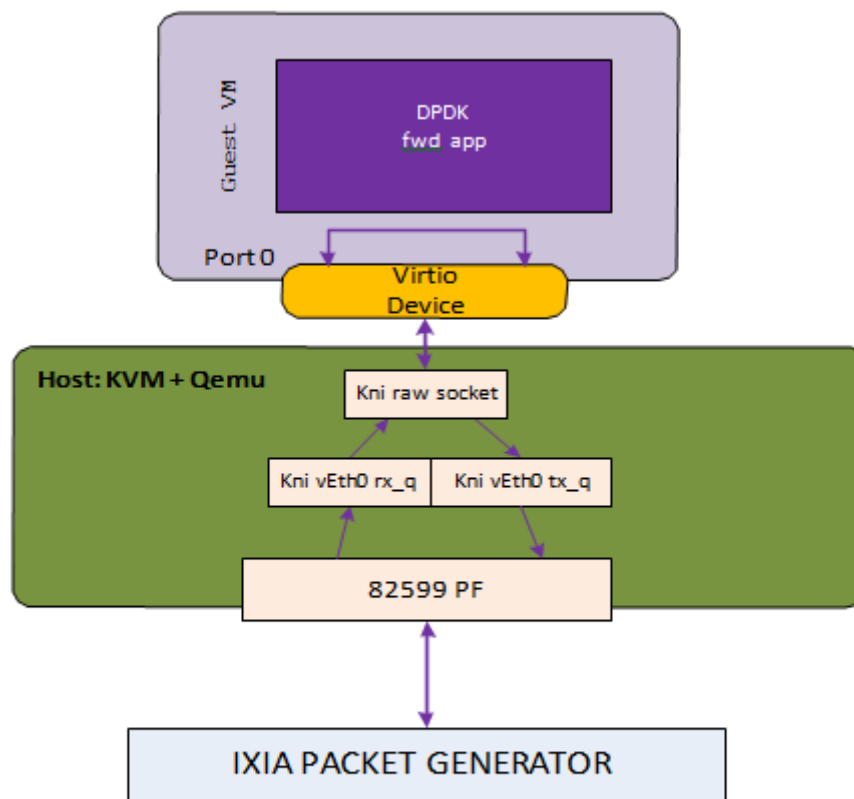
```
examples/kni/build/app/kni -l 0-3 -n 4 -- -p 0x1 -P --config="(0,1,3) "
```

This command generates one network device vEth0 for physical port. If specify more physical ports, the generated network device will be vEth1, vEth2, and so on.

For each physical port, kni creates two user threads. One thread loops to fetch packets from the physical NIC port into the kni receive queue. The other user thread loops to send packets in the kni transmit queue.

For each physical port, kni also creates a kernel thread that retrieves packets from the kni receive queue, place them onto kni's raw socket's queue and wake up the vhost kernel thread to exchange packets with the virtio virt queue.

For more details about kni, please refer to kni.



Host2VM communication example

Fig. 49.1: Host2VM Communication Example Using kni vhost Back End

3. Enable the kni raw socket functionality for the specified physical NIC port, get the generated file descriptor and set it in the qemu command line parameter. Always remember to set `ioeventfd_on` and `vhost_on`.

Example:

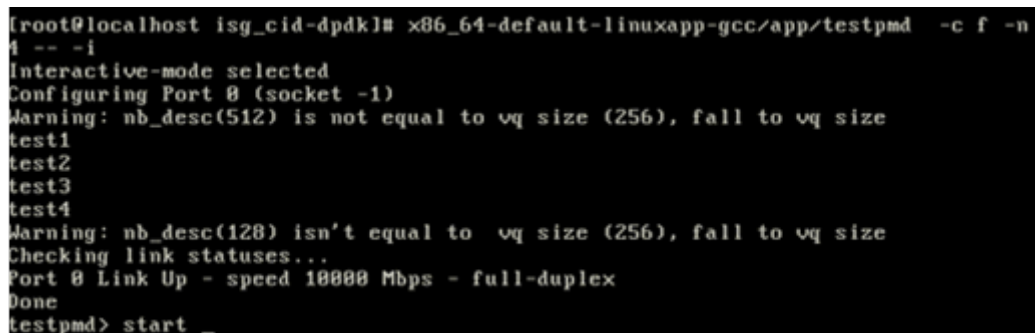
```
echo 1 > /sys/class/net/vEth0/sock_en
fd=`cat /sys/class/net/vEth0/sock_fd`
exec qemu-system-x86_64 -enable-kvm -cpu host \
-m 2048 -smp 4 -name dpdk-test1-vm1 \
-drive file=/data/DPDKVMS/dpdk-vm.img \
-netdev tap, fd=$fd,id=mynet_kni, script=no,vhost=on \
-device virtio-net-pci,netdev=mynet_kni,bus=pci.0,addr=0x3,ioeventfd=on \
-vnc:1 -daemonize
```

In the above example, virtio port 0 in the guest VM will be associated with vEth0, which in turns corresponds to a physical port, which means received packets come from vEth0, and transmitted packets is sent to vEth0.

4. In the guest, bind the virtio device to the `uio_pci_generic` kernel module and start the forwarding application. When the virtio port in guest bursts Rx, it is getting packets from the raw socket's receive queue. When the virtio port bursts Tx, it is sending packet to the `tx_q`.

```
modprobe uio
echo 512 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
modprobe uio_pci_generic
python usertools/dpdk-devbind.py -b uio_pci_generic 00:03.0
```

We use `testpmd` as the forwarding application in this example.



```
root@localhost isg_cid-dpdkl# x86_64-default-linuxapp-gcc/app/testpmd -c f -n
Interactive-mode selected
Configuring Port 0 (socket -1)
Warning: nb_desc(512) is not equal to vq size (256), fall to vq size
test1
test2
test3
test4
Warning: nb_desc(128) isn't equal to vq size (256), fall to vq size
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Done
testpmd> start _
```

Fig. 49.2: Running testpmd

5. Use IXIA packet generator to inject a packet stream into the KNI physical port.

The packet reception and transmission flow path is:

IXIA packet generator->82599 PF->KNI Rx queue->KNI raw socket queue->Guest VM virtio port 0 Rx burst->Guest VM virtio port 0 Tx burst-> KNI Tx queue ->82599 PF-> IXIA packet generator

49.5 Virtio with qemu virtio Back End

```
qemu-system-x86_64 -enable-kvm -cpu host -m 2048 -smp 2 -mem-path /dev/
hugepages -mem-prealloc
-drive file=/data/DPDKVMS/dpdk-vm1
-netdev tap,id=vm1_p1,ifname=tap0,script=no,vhost=on
```

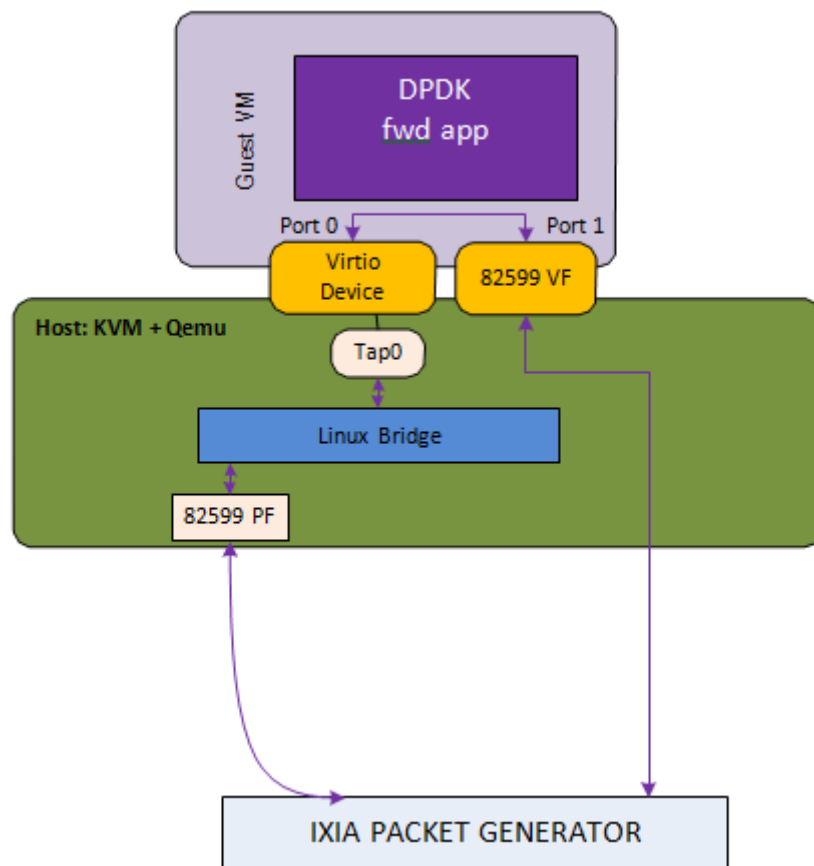


Fig. 49.3: Host2VM Communication Example Using qemu vhost Back End

```
-device virtio-net-pci,netdev=vm1_p1,bus=pci.0,addr=0x3,ioeventfd=on  
-device pci-assign,host=04:10.1 \
```

In this example, the packet reception flow path is:

IXIA packet generator->82599 PF->Linux Bridge->TAP0's socket queue-> Guest VM virtio port 0 Rx burst-> Guest VM 82599 VF port1 Tx burst-> IXIA packet generator

The packet transmission flow is:

IXIA packet generator-> Guest VM 82599 VF port1 Rx burst-> Guest VM virtio port 0 Tx burst-> tap -> Linux Bridge->82599 PF-> IXIA packet generator

49.6 Virtio PMD Rx/Tx Callbacks

Virtio driver has 6 Rx callbacks and 3 Tx callbacks.

Rx callbacks:

1. `virtio_recv_pkts`: Regular version without mergeable Rx buffer support for split virtqueue.
2. `virtio_recv_mergeable_pkts`: Regular version with mergeable Rx buffer support for split virtqueue.
3. `virtio_recv_pkts_vec`: Vector version without mergeable Rx buffer support, also fixes the available ring indexes and uses vector instructions to optimize performance for split virtqueue.
4. `virtio_recv_pkts_inorder`: In-order version with mergeable and non-mergeable Rx buffer support for split virtqueue.
5. `virtio_recv_pkts_packed`: Regular and in-order version without mergeable Rx buffer support for packed virtqueue.
6. `virtio_recv_mergeable_pkts_packed`: Regular and in-order version with mergeable Rx buffer support for packed virtqueue.

Tx callbacks:

1. `virtio_xmit_pkts`: Regular version for split virtqueue.
2. `virtio_xmit_pkts_inorder`: In-order version for split virtqueue.
3. `virtio_xmit_pkts_packed`: Regular and in-order version for packed virtqueue.

By default, the non-vector callbacks are used:

- For Rx: If mergeable Rx buffers is disabled then `virtio_recv_pkts` or `virtio_recv_pkts_packed` will be used, otherwise `virtio_recv_mergeable_pkts` or `virtio_recv_mergeable_pkts_packed` will be used.
- For Tx: `virtio_xmit_pkts` or `virtio_xmit_pkts_packed` will be used.

Vector callbacks will be used when:

- Mergeable Rx buffers is disabled.

The corresponding callbacks are:

- For Rx: `virtio_recv_pkts_vec`.

There is no vector callbacks for packed virtqueue for now.

Example of using the vector version of the virtio poll mode driver in `testpmd`:

```
testpmd -l 0-2 -n 4 -- -i --rxq=1 --txq=1 --nb-cores=1
```

In-order callbacks only work on simulated virtio user vdev.

For split virtqueue:

- For Rx: If in-order is enabled then `virtio_recv_pkts_inorder` is used.
- For Tx: If in-order is enabled then `virtio_xmit_pkts_inorder` is used.

For packed virtqueue, the default callbacks already support the in-order feature.

49.7 Interrupt mode

There are three kinds of interrupts from a virtio device over PCI bus: config interrupt, Rx interrupts, and Tx interrupts. Config interrupt is used for notification of device configuration changes, especially link status (lsc). Interrupt mode is translated into Rx interrupts in the context of DPDK.

Note: Virtio PMD already has support for receiving lsc from qemu when the link status changes, especially when vhost user disconnects. However, it fails to do that if the VM is created by qemu 2.6.2 or below, since the capability to detect vhost user disconnection is introduced in qemu 2.7.0.

49.7.1 Prerequisites for Rx interrupts

To support Rx interrupts, #. Check if guest kernel supports VFIO-NOIOMMU:

Linux started to support VFIO-NOIOMMU since 4.8.0. Make sure the guest kernel is compiled with:

```
CONFIG_VFIO_NOIOMMU=y
```

1. Properly set msix vectors when starting VM:

Enable multi-queue when starting VM, and specify msix vectors in qemu cmdline. (N+1) is the minimum, and (2N+2) is mostly recommended.

```
$(QEMU) ... -device virtio-net-pci,mq=on,vectors=2N+2 ...
```

2. In VM, insert vfio module in NOIOMMU mode:

```
modprobe vfio enable_unsafe_noiommu_mode=1
modprobe vfio-pci
```

3. In VM, bind the virtio device with vfio-pci:

```
python usertools/dpdk-devbind.py -b vfio-pci 00:03.0
```

49.7.2 Example

Here we use l3fwd-power as an example to show how to get started.

Example:

```
$ l3fwd-power -l 0-1 -- -p 1 -P --config="(0,0,1)" \
--no-numa --parse-ptype
```

49.8 Virtio PMD arguments

Below devargs are supported by the PCI virtio driver:

1. `vdpa`:

A virtio device could also be driven by vDPA (vhost data path acceleration) driver, and works as a HW vhost backend. This argument is used to specify a virtio device needs to work in vDPA mode. (Default: 0 (disabled))

Below devargs are supported by the virtio-user vdev:

1. `path`:

It is used to specify a path to connect to vhost backend.

2. `mac`:

It is used to specify the MAC address.

3. `cq`:

It is used to enable the control queue. (Default: 0 (disabled))

4. `queue_size`:

It is used to specify the queue size. (Default: 256)

5. `queues`:

It is used to specify the queue number. (Default: 1)

6. `iface`:

It is used to specify the host interface name for vhost-kernel backend.

7. `server`:

It is used to enable the server mode when using vhost-user backend. (Default: 0 (disabled))

8. `mrg_rxbuf`:

It is used to enable virtio device mergeable Rx buffer feature. (Default: 1 (enabled))

9. `in_order`:

It is used to enable virtio device in-order feature. (Default: 1 (enabled))

10. `packed_vq`:

It is used to enable virtio device packed virtqueue feature. (Default: 0 (disabled))

49.9 Virtio paths Selection and Usage

Logically virtio-PMD has 9 paths based on the combination of virtio features (Rx mergeable, In-order, Packed virtqueue), below is an introduction of these features:

- **Rx mergeable:** With this feature negotiated, device can receive large packets by combining individual descriptors.
- **In-order:** Some devices always use descriptors in the same order in which they have been made available, these devices can offer the VIRTIO_F_IN_ORDER feature. With this feature negotiated, driver will use descriptors in order.
- **Packed virtqueue:** The structure of packed virtqueue is different from split virtqueue, split virtqueue is composed of available ring, used ring and descriptor table, while packed virtqueue is composed of descriptor ring, driver event suppression and device event suppression. The idea behind this is to improve performance by avoiding cache misses and make it easier for hardware to implement.

49.9.1 Virtio paths Selection

If packed virtqueue is not negotiated, below split virtqueue paths will be selected according to below configuration:

1. Split virtqueue mergeable path: If Rx mergeable is negotiated, in-order feature is not negotiated, this path will be selected.
2. Split virtqueue non-mergeable path: If Rx mergeable and in-order feature are not negotiated, also Rx offload(s) are requested, this path will be selected.
3. Split virtqueue in-order mergeable path: If Rx mergeable and in-order feature are both negotiated, this path will be selected.
4. Split virtqueue in-order non-mergeable path: If in-order feature is negotiated and Rx mergeable is not negotiated, this path will be selected.
5. Split virtqueue vectorized Rx path: If Rx mergeable is disabled and no Rx offload requested, this path will be selected.

If packed virtqueue is negotiated, below packed virtqueue paths will be selected according to below configuration:

1. Packed virtqueue mergeable path: If Rx mergeable is negotiated, in-order feature is not negotiated, this path will be selected.
2. Packed virtqueue non-mergeable path: If Rx mergeable and in-order feature are not negotiated, this path will be selected.
3. Packed virtqueue in-order mergeable path: If in-order and Rx mergeable feature are both negotiated, this path will be selected.
4. Packed virtqueue in-order non-mergeable path: If in-order feature is negotiated and Rx mergeable is not negotiated, this path will be selected.

49.9.2 Rx/Tx callbacks of each Virtio path

Refer to above description, virtio path and corresponding Rx/Tx callbacks will be selected automatically. Rx callbacks and Tx callbacks for each virtio path are shown in below table:

Table 49.1: Virtio Paths and Callbacks

Virtio paths	Rx callbacks	Tx callbacks
Split virtqueue mergeable path	virtio_recv_mergeable_pkts	virtio_xmit_pkts
Split virtqueue non-mergeable path	virtio_recv_pkts	virtio_xmit_pkts
Split virtqueue in-order mergeable path	virtio_recv_pkts_inorder	virtio_xmit_pkts_inorder
Split virtqueue in-order non-mergeable path	virtio_recv_pkts_inorder	virtio_xmit_pkts_inorder
Split virtqueue vectorized Rx path	virtio_recv_pkts_vec	virtio_xmit_pkts
Packed virtqueue mergeable path	virtio_recv_mergeable_pkts_packed	virtio_xmit_pkts_packed
Packed virtqueue non-mergeable path	virtio_recv_pkts_packed	virtio_xmit_pkts_packed
Packed virtqueue in-order mergeable path	virtio_recv_mergeable_pkts_packed	virtio_xmit_pkts_packed
Packed virtqueue in-order non-mergeable path	virtio_recv_pkts_packed	virtio_xmit_pkts_packed

49.9.3 Virtio paths Support Status from Release to Release

Virtio feature implementation:

- In-order feature is supported since DPDK 18.08 by adding new Rx/Tx callbacks `virtio_recv_pkts_inorder` and `virtio_xmit_pkts_inorder`.
- Packed virtqueue is supported since DPDK 19.02 by adding new Rx/Tx callbacks `virtio_recv_pkts_packed`, `virtio_recv_mergeable_pkts_packed` and `virtio_xmit_pkts_packed`.

All virtio paths support status are shown in below table:

Table 49.2: Virtio Paths and Releases

Virtio paths	16.11 ~ 18.05	18.08 ~ 18.11	19.02 ~ 19.11
Split virtqueue mergeable path	Y	Y	Y
Split virtqueue non-mergeable path	Y	Y	Y
Split virtqueue vectorized Rx path	Y	Y	Y
Split virtqueue simple Tx path	Y	N	N
Split virtqueue in-order mergeable path		Y	Y
Split virtqueue in-order non-mergeable path		Y	Y
Packed virtqueue mergeable path			Y
Packed virtqueue non-mergeable path			Y
Packed virtqueue in-order mergeable path			Y
Packed virtqueue in-order non-mergeable path			Y

49.9.4 QEMU Support Status

- Qemu now supports three paths of split virtqueue: Split virtqueue mergeable path, Split virtqueue non-mergeable path, Split virtqueue vectorized Rx path.

- Since qemu 4.2.0, Packed virtqueue mergeable path and Packed virtqueue non-mergeable path can be supported.

49.9.5 How to Debug

If you meet performance drop or some other issues after upgrading the driver or configuration, below steps can help you identify which path you selected and root cause faster.

1. Run vhost/virtio test case;
2. Run “perf top” and check virtio Rx/Tx callback names;
3. Identify which virtio path is selected refer to above table.

POLL MODE DRIVER THAT WRAPS VHOST LIBRARY

This PMD is a thin wrapper of the DPDK vhost library. The user can handle virtqueues as one of normal DPDK port.

50.1 Vhost Implementation in DPDK

Please refer to Chapter “Vhost Library” of *DPDK Programmer's Guide* to know detail of vhost.

50.2 Features and Limitations of vhost PMD

Currently, the vhost PMD provides the basic functionality of packet reception, transmission and event handling.

- It has multiple queues support.
- It supports `RTE_ETH_EVENT_INTR_LSC` and `RTE_ETH_EVENT_QUEUE_STATE` events.
- It supports Port Hotplug functionality.
- Don't need to stop RX/TX, when the user wants to stop a guest or a virtio-net driver on guest.

50.3 Vhost PMD arguments

The user can specify below arguments in `-vdev` option.

1. `iface`:

It is used to specify a path to connect to a QEMU virtio-net device.

2. `queues`:

It is used to specify the number of queues virtio-net device has. (Default: 1)

3. `iommu-support`:

It is used to enable iommu support in vhost library. (Default: 0 (disabled))

4. `postcopy-support`:

It is used to enable postcopy live-migration support in vhost library. (Default: 0 (disabled))

5. tso:

It is used to enable tso support in vhost library. (Default: 0 (disabled))

50.4 Vhost PMD event handling

This section describes how to handle vhost PMD events.

The user can register an event callback handler with `rte_eth_dev_callback_register()`. The registered callback handler will be invoked with one of below event types.

1. RTE_ETH_EVENT_INTR_LSC:

It means link status of the port was changed.

2. RTE_ETH_EVENT_QUEUE_STATE:

It means some of queue statuses were changed. Call `rte_eth_vhost_get_queue_event()` in the callback handler. Because changing multiple statuses may occur only one event, call the function repeatedly as long as it doesn't return negative value.

50.5 Vhost PMD with testpmd application

This section demonstrates vhost PMD with testpmd DPDK sample application.

1. Launch the testpmd with vhost PMD:

```
./testpmd -l 0-3 -n 4 --vdev 'net_vhost0,iface=/tmp/sock0,queues=1' -- -i
```

Other basic DPDK preparations like hugepage enabling here. Please refer to the *DPDK Getting Started Guide* for detailed instructions.

2. Launch the QEMU:

```
qemu-system-x86_64 <snip>
    -chardev socket,id=chr0,path=/tmp/sock0 \
    -netdev vhost-user,id=net0,chardev=chr0,vhostforce,queues=1 \
    -device virtio-net-pci,netdev=net0
```

This command attaches one virtio-net device to QEMU guest. After initialization processes between QEMU and DPDK vhost library are done, status of the port will be linked up.

POLL MODE DRIVER FOR PARAVIRTUAL VMXNET3 NIC

The VMXNET3 adapter is the next generation of a paravirtualized NIC, introduced by VMware* ESXi. It is designed for performance, offers all the features available in VMXNET2, and adds several new features such as, multi-queue support (also known as Receive Side Scaling, RSS), IPv6 offloads, and MSI/MSI-X interrupt delivery. One can use the same device in a DPDK application with VMXNET3 PMD introduced in DPDK API.

In this chapter, two setups with the use of the VMXNET3 PMD are demonstrated:

1. Vmxnet3 with a native NIC connected to a vSwitch
2. Vmxnet3 chaining VMs connected to a vSwitch

51.1 VMXNET3 Implementation in the DPDK

For details on the VMXNET3 device, refer to the VMXNET3 driver's `vmxnet3` directory and support manual from VMware*.

For performance details, refer to the following link from VMware:

http://www.vmware.com/pdf/vsp_4_vmxnet3_perf.pdf

As a PMD, the VMXNET3 driver provides the packet reception and transmission callbacks, `vmxnet3_recv_pkts` and `vmxnet3_xmit_pkts`.

The VMXNET3 PMD handles all the packet buffer memory allocation and resides in guest address space and it is solely responsible to free that memory when not needed. The packet buffers and features to be supported are made available to hypervisor via VMXNET3 PCI configuration space BARs. During RX/TX, the packet buffers are exchanged by their GPAs, and the hypervisor loads the buffers with packets in the RX case and sends packets to vSwitch in the TX case.

The VMXNET3 PMD is compiled with `vmxnet3` device headers. The interface is similar to that of the other PMDs available in the DPDK API. The driver pre-allocates the packet buffers and loads the command ring descriptors in advance. The hypervisor fills those packet buffers on packet arrival and write completion ring descriptors, which are eventually pulled by the PMD. After reception, the DPDK application frees the descriptors and loads new packet buffers for the coming packets. The interrupts are disabled and there is no notification required. This keeps performance up on the RX side, even though the device provides a notification feature.

In the transmit routine, the DPDK application fills packet buffer pointers in the descriptors of the command ring and notifies the hypervisor. In response the hypervisor takes packets and passes them to the vSwitch, It writes into the completion descriptors ring. The rings are read by the PMD in the next transmit routine call and the buffers and descriptors are freed from memory.

51.2 Features and Limitations of VMXNET3 PMD

In release 1.6.0, the VMXNET3 PMD provides the basic functionality of packet reception and transmission. There are several options available for filtering packets at VMXNET3 device level including:

1. MAC Address based filtering:
 - Unicast, Broadcast, All Multicast modes - SUPPORTED BY DEFAULT
 - Multicast with Multicast Filter table - NOT SUPPORTED
 - Promiscuous mode - SUPPORTED
 - RSS based load balancing between queues - SUPPORTED
2. VLAN filtering:
 - VLAN tag based filtering without load balancing - SUPPORTED

Note:

- Release 1.6.0 does not support separate headers and body receive cmd_ring and hence, multiple segment buffers are not supported. Only cmd_ring_0 is used for packet buffers, one for each descriptor.
- Receive and transmit of scattered packets is not supported.
- Multicast with Multicast Filter table is not supported.

51.3 Prerequisites

The following prerequisites apply:

- Before starting a VM, a VMXNET3 interface to a VM through VMware vSphere Client must be assigned. This is shown in the figure below.

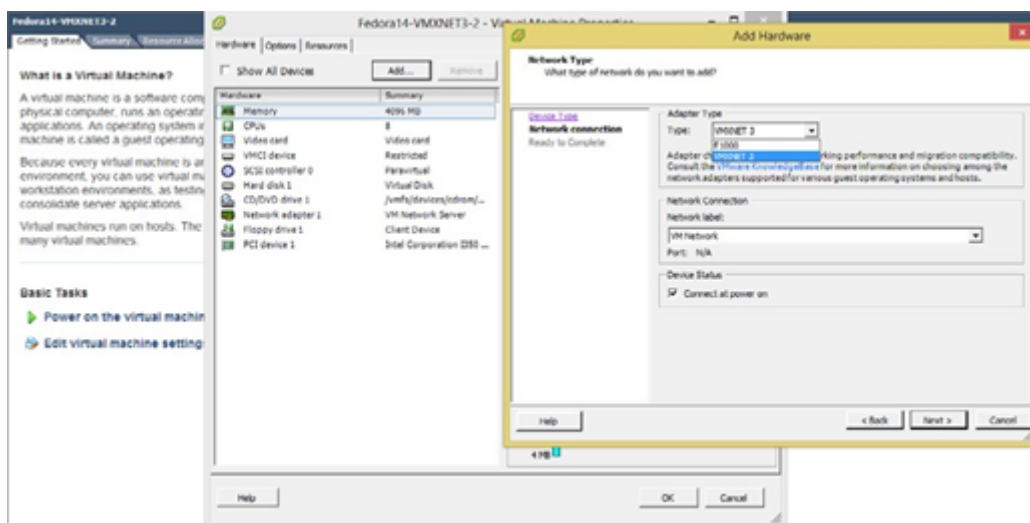


Fig. 51.1: Assigning a VMXNET3 interface to a VM using VMware vSphere Client

Note: Depending on the Virtual Machine type, the VMware vSphere Client shows Ethernet adaptors while adding an Ethernet device. Ensure that the VM type used offers a VMXNET3 device. Refer to the VMware documentation for a listed of VMs.

Note: Follow the *DPDK Getting Started Guide* to setup the basic DPDK environment.

Note: Follow the *DPDK Sample Application's User Guide*, L2 Forwarding/L3 Forwarding and TestPMD for instructions on how to run a DPDK application using an assigned VMXNET3 device.

51.4 VMXNET3 with a Native NIC Connected to a vSwitch

This section describes an example setup for Phy-vSwitch-VM-Phy communication.

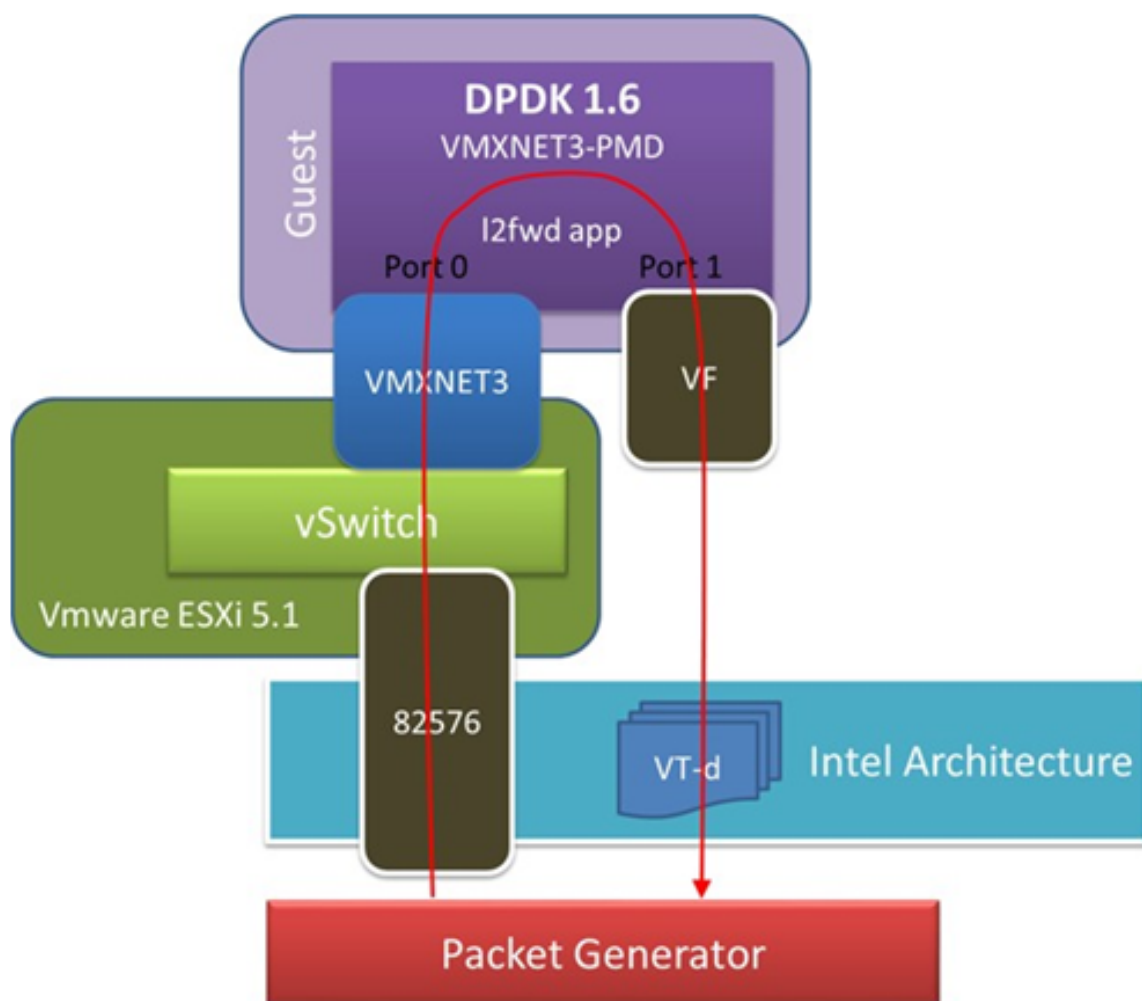


Fig. 51.2: VMXNET3 with a Native NIC Connected to a vSwitch

Note: Other instructions on preparing to use DPDK such as, hugepage enabling, uio port binding are not listed here. Please refer to *DPDK Getting Started Guide* and *DPDK Sample Application's User Guide* for detailed instructions.

The packet reception and transmission flow path is:

```

Packet generator -> 82576
                  -> VMware ESXi vSwitch
                  -> VMXNET3 device
                  -> Guest VM VMXNET3 port 0 rx burst
                  -> Guest VM 82599 VF port 0 tx burst
                  -> 82599 VF
                  -> Packet generator
  
```

51.5 VMXNET3 Chaining VMs Connected to a vSwitch

The following figure shows an example VM-to-VM communication over a Phy-VM-vSwitch-VM-Phy communication channel.

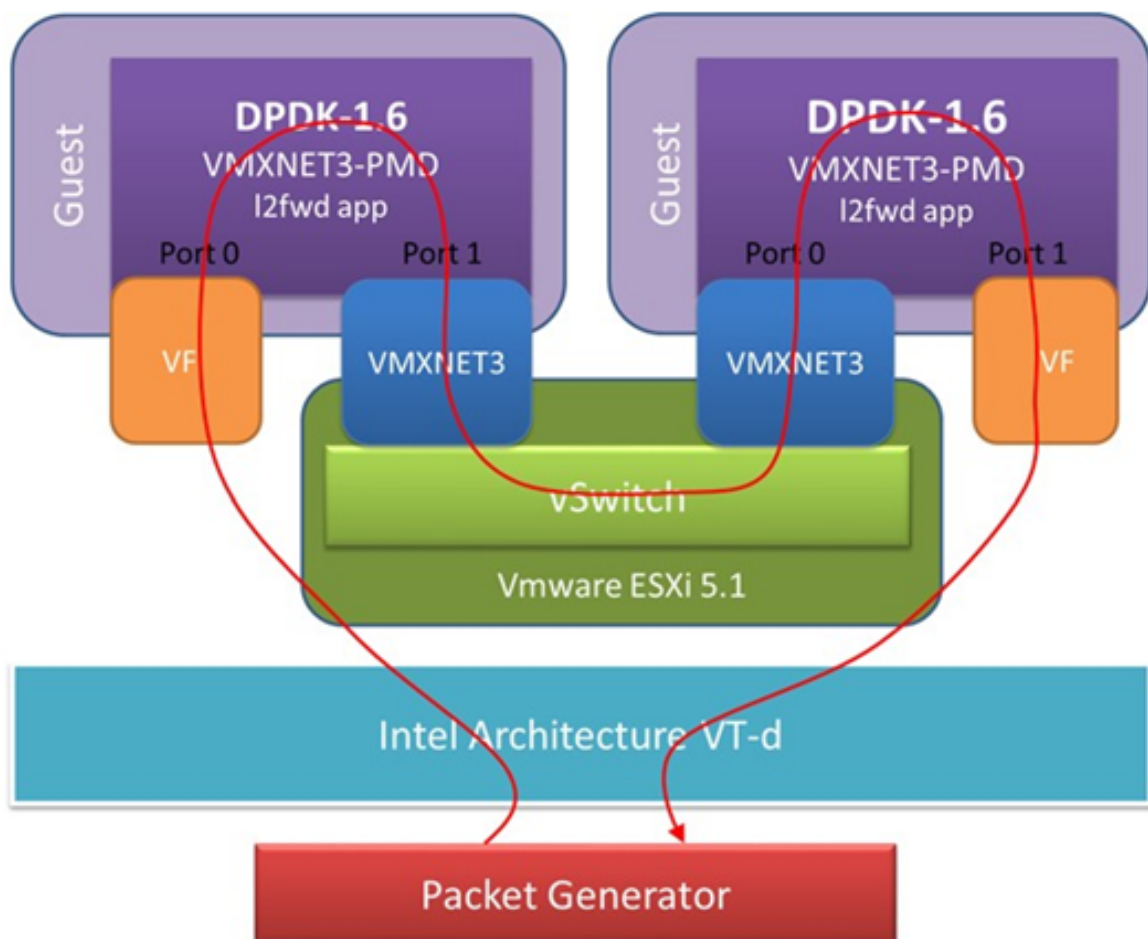


Fig. 51.3: VMXNET3 Chaining VMs Connected to a vSwitch

Note: When using the L2 Forwarding or L3 Forwarding applications, a destination MAC address needs to be written in packets to hit the other VM's VMXNET3 interface.

In this example, the packet flow path is:

```
Packet generator -> 82599 VF
                  -> Guest VM 82599 port 0 rx burst
                  -> Guest VM VMXNET3 port 1 tx burst
                  -> VMXNET3 device
                  -> VMware ESXi vSwitch
                  -> VMXNET3 device
                  -> Guest VM VMXNET3 port 0 rx burst
                  -> Guest VM 82599 VF port 1 tx burst
                  -> 82599 VF
                  -> Packet generator
```

LIBPCAP AND RING BASED POLL MODE DRIVERS

In addition to Poll Mode Drivers (PMDs) for physical and virtual hardware, the DPDK also includes pure-software PMDs, two of these drivers are:

- A libpcap -based PMD (`librte_pmd_pcap`) that reads and writes packets using libpcap, - both from files on disk, as well as from physical NIC devices using standard Linux kernel drivers.
- A ring-based PMD (`librte_pmd_ring`) that allows a set of software FIFOs (that is, `rte_ring`) to be accessed using the PMD APIs, as though they were physical NICs.

Note: The libpcap -based PMD is disabled by default in the build configuration files, owing to an external dependency on the libpcap development files which must be installed on the board. Once the libpcap development files are installed, the library can be enabled by setting `CONFIG_RTE_LIBRTE_PMD_PCAP=y` and recompiling the DPDK.

52.1 Using the Drivers from the EAL Command Line

For ease of use, the DPDK EAL also has been extended to allow pseudo-Ethernet devices, using one or more of these drivers, to be created at application startup time during EAL initialization.

To do so, the `-vdev=` parameter must be passed to the EAL. This takes take options to allow ring and pcap-based Ethernet to be allocated and used transparently by the application. This can be used, for example, for testing on a virtual machine where there are no Ethernet ports.

52.1.1 Libpcap-based PMD

Pcap-based devices can be created using the virtual device `-vdev` option. The device name must start with the `net_pcap` prefix followed by numbers or letters. The name is unique for each device. Each device can have multiple stream options and multiple devices can be used. Multiple device definitions can be arranged using multiple `-vdev`. Device name and stream options must be separated by commas as shown below:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \  
--vdev 'net_pcap0,stream_opt0=..,stream_opt1=..' \  
--vdev='net_pcap1,stream_opt0=..'
```

Device Streams

Multiple ways of stream definitions can be assessed and combined as long as the following two rules are respected:

- A device is provided with two different streams - reception and transmission.
- A device is provided with one network interface name used for reading and writing packets.

The different stream types are:

- `rx_pcap`: Defines a reception stream based on a pcap file. The driver reads each packet within the given pcap file as if it was receiving it from the wire. The value is a path to a valid pcap file.

`rx_pcap=/path/to/file.pcap`

- `tx_pcap`: Defines a transmission stream based on a pcap file. The driver writes each received packet to the given pcap file. The value is a path to a pcap file. The file is overwritten if it already exists and it is created if it does not.

`tx_pcap=/path/to/file.pcap`

- `rx_iface`: Defines a reception stream based on a network interface name. The driver reads packets from the given interface using the Linux kernel driver for that interface. The driver captures both the incoming and outgoing packets on that interface. The value is an interface name.

`rx_iface=eth0`

- `rx_iface_in`: Defines a reception stream based on a network interface name. The driver reads packets from the given interface using the Linux kernel driver for that interface. The driver captures only the incoming packets on that interface. The value is an interface name.

`rx_iface_in=eth0`

- `tx_iface`: Defines a transmission stream based on a network interface name. The driver sends packets to the given interface using the Linux kernel driver for that interface. The value is an interface name.

`tx_iface=eth0`

- `iface`: Defines a device mapping a network interface. The driver both reads and writes packets from and to the given interface. The value is an interface name.

`iface=eth0`

Runtime Config Options

- Use PCAP interface physical MAC

In case `iface=` configuration is set, user may want to use the selected interface's physical MAC address. This can be done with a `devarg phy_mac`, for example:

```
--vdev 'net_pcap0,iface=eth0,phy_mac=1'
```

- Use the RX PCAP file to infinitely receive packets

In case `rx_pcap=` configuration is set, user may want to use the selected PCAP file for rudimentary performance testing. This can be done with a `devarg infinite_rx`, for example:

```
--vdev 'net_pcap0,rx_pcap=file_rx.pcap,infinite_rx=1'
```

When this mode is used, it is recommended to drop all packets on transmit by not providing a tx_pcap or tx_iface.

This option is device wide, so all queues on a device will either have this enabled or disabled. This option should only be provided once per device.

- Drop all packets on transmit

The user may want to drop all packets on tx for a device. This can be done by not providing a tx_pcap or tx_iface, for example:

```
--vdev 'net_pcap0,rx_pcap=file_rx.pcap'
```

In this case, one tx drop queue is created for each rxq on that device.

- Receive no packets on Rx

The user may want to run without receiving any packets on Rx. This can be done by not providing a rx_pcap or rx_iface, for example:

```
--vdev 'net_pcap0,tx_pcap=file_tx.pcap'
```

In this case, one dummy rx queue is created for each tx queue argument passed

Examples of Usage

Read packets from one pcap file and write them to another:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
--vdev 'net_pcap0,rx_pcap=file_rx.pcap,tx_pcap=file_tx.pcap' \
-- --port-topology=chained
```

Read packets from a network interface and write them to a pcap file:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
--vdev 'net_pcap0,rx_iface=eth0,tx_pcap=file_tx.pcap' \
-- --port-topology=chained
```

Read packets from a pcap file and write them to a network interface:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
--vdev 'net_pcap0,rx_pcap=file_rx.pcap,tx_iface=eth1' \
-- --port-topology=chained
```

Forward packets through two network interfaces:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
--vdev 'net_pcap0,iface=eth0' --vdev='net_pcap1;iface=eth1'
```

Enable 2 tx queues on a network interface:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
--vdev 'net_pcap0,rx_iface=eth1,tx_iface=eth1,tx_iface=eth1' \
-- --txq 2
```

Read only incoming packets from a network interface and write them back to the same network interface:

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
--vdev 'net_pcap0,rx_iface_in=eth1,tx_iface=eth1'
```

Using libpcap-based PMD with the testpmd Application

One of the first things that testpmd does before starting to forward packets is to flush the RX streams by reading the first 512 packets on every RX stream and discarding them. When using a libpcap-based PMD this behavior can be turned off using the following command line option:

```
--no-flush-rx
```

It is also available in the runtime command line:

```
set flush_rx on/off
```

It is useful for the case where the rx_pcap is being used and no packets are meant to be discarded. Otherwise, the first 512 packets from the input pcap file will be discarded by the RX flushing operation.

```
$RTE_TARGET/app/testpmd -l 0-3 -n 4 \
--vdev 'net_pcap0,rx_pcap=file_rx.pcap,tx_pcap=file_tx.pcap' \
-- --port-topology=chained --no-flush-rx
```

Note: The network interface provided to the PMD should be up. The PMD will return an error if interface is down, and the PMD itself won't change the status of the external network interface.

52.1.2 Rings-based PMD

To run a DPDK application on a machine without any Ethernet devices, a pair of ring-based rte_ethdevs can be used as below. The device names passed to the `--vdev` option must start with `net_ring` and take no additional parameters. Multiple devices may be specified, separated by commas.

```
./testpmd -l 1-3 -n 4 --vdev=net_ring0 --vdev=net_ring1 -- -i
EAL: Detected lcore 1 as core 1 on socket 0
...

Interactive-mode selected
Configuring Port 0 (socket 0)
Configuring Port 1 (socket 0)
Checking link statuses...
Port 0 Link Up - speed 10000 Mbps - full-duplex
Port 1 Link Up - speed 10000 Mbps - full-duplex
Done

testpmd> start tx_first
io packet forwarding - CRC stripping disabled - packets/burst=16
nb forwarding cores=1 - nb forwarding ports=2
RX queues=1 - RX desc=128 - RX free threshold=0
RX threshold registers: pthresh=8 hthresh=8 wthresh=4
TX queues=1 - TX desc=512 - TX free threshold=0
TX threshold registers: pthresh=36 hthresh=0 wthresh=0
TX RS bit threshold=0 - TXQ flags=0x0

testpmd> stop
Telling cores to stop...
Waiting for lcores to finish...
```

```

----- Forward statistics for port 0 -----
RX-packets: 231192368      RX-dropped: 0      RX-total: 231192368
TX-packets: 231192384      TX-dropped: 0      TX-total: 231192384
-----

----- Forward statistics for port 1 -----
RX-packets: 231192368      RX-dropped: 0      RX-total: 231192368
TX-packets: 231192384      TX-dropped: 0      TX-total: 231192384
-----

+++++ Accumulated forward statistics for allports+++++
RX-packets: 462384736  RX-dropped: 0  RX-total: 462384736
TX-packets: 462384768  TX-dropped: 0  TX-total: 462384768
+++++

Done.

```

52.1.3 Using the Poll Mode Driver from an Application

Both drivers can provide similar APIs to allow the user to create a PMD, that is, `rte_ethdev` structure, instances at run-time in the end-application, for example, using `rte_eth_from_rings()` or `rte_eth_from_pcaps()` APIs. For the rings-based PMD, this functionality could be used, for example, to allow data exchange between cores using rings to be done in exactly the same way as sending or receiving packets from an Ethernet device. For the libpcap-based PMD, it allows an application to open one or more pcap files and use these as a source of packet input to the application.

Usage Examples

To create two pseudo-Ethernet ports where all traffic sent to a port is looped back for reception on the same port (error handling omitted for clarity):

```

#define RING_SIZE 256
#define NUM_RINGS 2
#define SOCKET0 0

struct rte_ring *ring[NUM_RINGS];
int port0, port1;

ring[0] = rte_ring_create("R0", RING_SIZE, SOCKET0, RING_F_SP_ENQ|RING_F_SC_DEQ);
ring[1] = rte_ring_create("R1", RING_SIZE, SOCKET0, RING_F_SP_ENQ|RING_F_SC_DEQ);

/* create two ethdev's */

port0 = rte_eth_from_rings("net_ring0", ring, NUM_RINGS, ring, NUM_RINGS, SOCKET0);
port1 = rte_eth_from_rings("net_ring1", ring, NUM_RINGS, ring, NUM_RINGS, SOCKET0);

```

To create two pseudo-Ethernet ports where the traffic is switched between them, that is, traffic sent to port 0 is read back from port 1 and vice-versa, the final two lines could be changed as below:

```

port0 = rte_eth_from_rings("net_ring0", &ring[0], 1, &ring[1], 1, SOCKET0);
port1 = rte_eth_from_rings("net_ring1", &ring[1], 1, &ring[0], 1, SOCKET0);

```

This type of configuration could be useful in a pipeline model, for example, where one may want to have inter-core communication using pseudo Ethernet devices rather than raw rings, for reasons of API consistency.

Enqueuing and dequeuing items from an `rte_ring` using the rings-based PMD may be slower than using the native rings API. This is because DPDK Ethernet drivers make use of function pointers to call the

appropriate enqueue or dequeue functions, while the `rte_ring` specific functions are direct function calls in the code and are often inlined by the compiler.

Once an `ethdev` has been created, for either a ring or a pcap-based PMD, it should be configured and started in the same way as a regular Ethernet device, that is, by calling `rte_eth_dev_configure()` to set the number of receive and transmit queues, then calling `rte_eth_rx_queue_setup()` / `tx_queue_setup()` for each of those queues and finally calling `rte_eth_dev_start()` to allow transmission and reception of packets to begin.

FAIL-SAFE POLL MODE DRIVER LIBRARY

The Fail-safe poll mode driver library (**librte_pmd_failsafe**) is a virtual device that allows using any device supporting hotplug (sudden device removal and plugging on its bus), without modifying other components relying on such device (application, other PMDs).

Additionally to the Seamless Hotplug feature, the Fail-safe PMD offers the ability to redirect operations to secondary devices when the primary has been removed from the system.

Note: The library is enabled by default. You can enable it or disable it manually by setting the `CONFIG_RTE_LIBRTE_PMD_FAILSAFE` configuration option.

53.1 Features

The Fail-safe PMD only supports a limited set of features. If you plan to use a device underneath the Fail-safe PMD with a specific feature, this feature must be supported by the Fail-safe PMD to avoid throwing any error.

A notable exception is the device removal feature. The fail-safe PMD being a virtual device, it cannot currently be removed in the sense of a specific bus hotplug, like for PCI for example. It will however enable this feature for its sub-device automatically, detecting those that are capable and register the relevant callback for such event.

Check the feature matrix for the complete set of supported features.

53.2 Compilation option

This option can be modified in the `$RTE_TARGET/build/.config` file.

- `CONFIG_RTE_LIBRTE_PMD_FAILSAFE` (default **y**)

Toggle compiling `librte_pmd_failsafe`.

53.3 Using the Fail-safe PMD from the EAL command line

The Fail-safe PMD can be used like most other DPDK virtual devices, by passing a `--vdev` parameter to the EAL when starting the application. The device name must start with the *net_failsafe* prefix,

followed by numbers or letters. This name must be unique for each device. Each fail-safe instance must have at least one sub-device, and at most two.

A sub-device can be any legal DPDK device, including possibly another fail-safe instance.

53.3.1 Fail-safe command line parameters

- **dev(<iface>)** parameter

This parameter allows the user to define a sub-device. The `<iface>` part of this parameter must be a valid device definition. It could be the argument provided to any `-w` device specification or the argument that would be given to a `--vdev` parameter (including a fail-safe). Enclosing the device definition within parenthesis here allows using additional sub-device parameters if need be. They will be passed on to the sub-device.

Note: In case of whitelist sub-device probed by EAL, fail-safe PMD will take the device as is, which means that EAL device options are taken in this case. When trying to use a PCI device automatically probed in blacklist mode, the syntax for the fail-safe must be with the full PCI id: `Domain:Bus:Device.Function`. See the usage example section.

- **exec(<shell command>)** parameter

This parameter allows the user to provide a command to the fail-safe PMD to execute and define a sub-device. It is done within a regular shell context. The first line of its output is read by the fail-safe PMD and otherwise interpreted as if passed by the regular **dev** parameter. Any other line is discarded. If the command fail or output an incorrect string, the sub-device is not initialized. All commas within the `shell command` are replaced by spaces before executing the command. This helps using scripts to specify devices.

- **fd(<file descriptor number>)** parameter

This parameter reads a device definition from an arbitrary file descriptor number in `<iface>` format as described above.

The file descriptor is read in non-blocking mode and is never closed in order to take only the last line into account (unlike `exec()`) at every probe attempt.

- **mac** parameter [MAC address]

This parameter allows the user to set a default MAC address to the fail-safe and all of its sub-devices. If no default mac address is provided, the fail-safe PMD will read the MAC address of the first of its sub-device to be successfully probed and use it as its default MAC address, trying to set it to all of its other sub-devices. If no sub-device was successfully probed at initialization, then a random MAC address is generated, that will be subsequently applied to all sub-device once they are probed.

- **hotplug_poll** parameter [UINT64] (default **2000**)

This parameter allows the user to configure the amount of time in milliseconds between two slave upkeep round.

53.3.2 Usage example

This section shows some example of using **testpmd** with a fail-safe PMD.

1. To build a PMD and configure DPDK, refer to the document *compiling and testing a PMD for a NIC*.
2. Start testpmd. The slave device should be blacklisted from normal EAL operations to avoid probing it twice when in PCI blacklist mode.

```
$RTE_TARGET/build/app/testpmd -c 0xff -n 4 \
--vdev 'net_failsafe0,mac=de:ad:be:ef:01:02,dev(84:00.0),dev(net_ring0)' \
-b 84:00.0 -b 00:04.0 -- -i
```

If the slave device being used is not blacklisted, it will be probed by the EAL first. When the fail-safe then tries to initialize it the probe operation fails.

Note that PCI blacklist mode is the default PCI operating mode.

3. Alternatively, it can be used alongside any other device in whitelist mode.

```
$RTE_TARGET/build/app/testpmd -c 0xff -n 4 \
--vdev 'net_failsafe0,mac=de:ad:be:ef:01:02,dev(84:00.0),dev(net_ring0)' \
-w 81:00.0 -- -i
```

4. Start testpmd using a flexible device definition

```
$RTE_TARGET/build/app/testpmd -c 0xff -n 4 --no-pci \
--vdev='net_failsafe0,exec(echo 84:00.0)' -- -i
```

5. Start testpmd, automatically probing the device 84:00.0 and using it with the fail-safe.

```
$RTE_TARGET/build/app/testpmd -c 0xff -n 4 \
--vdev 'net_failsafe0,dev(0000:84:00.0),dev(net_ring0)' -- -i
```

53.4 Using the Fail-safe PMD from an application

This driver strives to be as seamless as possible to existing applications, in order to propose the hotplug functionality in the easiest way possible.

Care must be taken, however, to respect the **ether** API concerning device access, and in particular, using the `RTE_ETH_FOREACH_DEV` macro to iterate over ethernet devices, instead of directly accessing them or by writing one's own device iterator.

53.5 Plug-in feature

A sub-device can be defined without existing on the system when the fail-safe PMD is initialized. Upon probing this device, the fail-safe PMD will detect its absence and postpone its use. It will then register for a periodic check on any missing sub-device.

During this time, the fail-safe PMD can be used normally, configured and told to emit and receive packets. It will store any applied configuration, and try to apply it upon the probing of its missing sub-device. After this configuration pass, the new sub-device will be synchronized with other sub-devices, i.e. be started if the fail-safe PMD has been started by the user before.

53.6 Plug-out feature

A sub-device supporting the device removal event can be removed from its bus at any time. The fail-safe PMD will register a callback for such event and react accordingly. It will try to safely stop, close and

uninit the sub-device having emitted this event, allowing it to free its eventual resources.

53.7 Fail-safe glossary

Fallback device [Secondary device] The fail-safe will fail-over onto this device when the preferred device is absent.

Preferred device [Primary device] The first declared sub-device in the fail-safe parameters. When this device is plugged, it is always used as emitting device. It is the main sub-device and is used as target for configuration operations if there is any ambiguity.

Upkeep round Periodical process when slaves are serviced. Each devices having a state different to that of the fail-safe device itself, is synchronized with it. Additionally, each slave having the remove flag set are cleaned-up.

Slave In the context of the fail-safe PMD, synonymous to sub-device.

Sub-device A device being utilized by the fail-safe PMD. This is another PMD running underneath the fail-safe PMD. Any sub-device can disappear at any time. The fail-safe will ensure that the device removal happens gracefully.