LSDKYOCTOUG

Layerscape Software Development Kit User Guide for Yocto

Supports LSDK 20.04 Yocto 3.1 release

Rev. 3.1 — 04 October 2021 User Guide

Contents

1 Introduction....1
2 Release notes....1
3 Download Yocto layers....6
4 Build Yocto images....7
5 Boot boards with Yocto image....7
6 Program TF-A binaries....14
7 QorIQ memory layout....18
8 Prebuilt toolchains....20
9 User space applications....21
10 Frequently asked questions....121
11 Related resources....121

1 Introduction

This document describes the LSDK Yocto release, a Yocto layer with LSDK components. This release provides recipes that integrate the latest LSDK components into the most recent Yocto drop. The recipes will eventually make their way into the next community Yocto version at www.yoctoproject.org.

2 Release notes

2.1 What's new

The table below shows Yocto branches and corresponding Yocto versions.

Table 1. Available Yocto versions

Yocto branch	Yocto version
dunfell	YP 3.1–LSDK 2004
zeus	YP 3.0–LSDK 1909
warrior	YP 2.7–LSDK 1906
thud	YP 2.6–LSDK 1809
sumo	YP 2.5–LSDK 1806

2.2 Feature support matrix

The tables below provide features of the current LSDK Yocto release and explain which feature is supported for which processor in the current software release. For these tables, the legends are defined as follows:

- Y - Feature is supported by software
- / - Feature is not supported by software
- na - Hardware feature is not available

Table 2. Key software features

Feature	LS1012A	LS1021A	LS1028A	LS1043A	LS1046A	LS1088A	LS2088A	LX2160A	MPC8548	P1010	P1020	P2020	P2041	P3041	P4080	P5040	T1024	T1042	T2080	T4240
32-bit user space, BE	/	/	/	/	/	/	/	/	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	/	/
64-bit user space, BE	/	na	/	/	/	/	/	/	na	na	na	na	na	na	na	Y	Y	Y	Y	Y

Table continues on the next page...

Table 2. Key software features (continued)

Feature	LS1012A	LS1021A	LS1028A	LS1043A	LS1046A	LS1088A	LS2088A	LX2160A	MPC8548	P1010	P1020	P2020	P2041	P3041	P4080	P5040	T1024	T1042	T2080	T4240
32-bit user space, LE	/	Y	/	/	/	/	/	/	na	na	na	na	na	na	na	na	na	na	na	na
64-bit user space, LE	Y	na	Y	Y	Y	Y	Y	Y	na	na	na	na	na	na	na	na	na	na	na	na
36-bit physical memory	Y	Y	na	na	na	na	na	na	Y	Y	Y	Y	Y	Y	Y	Y	Y	na	na	na
40-bit physical memory	Y	na	Y	Y	Y	Y	Y	Y	na	na	na	na	na	na	na	na	na	na	na	na
AIOP service layer	na	na	na	na	na	Y	Y	na	na	na	na	na	na	na	na	na	na	na	na	na
ASF	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/
Data Plane Development Kit (DPDK)	Y	/	Y	Y	Y	Y	Y	Y	/	/	/	/	/	/	/	/	/	/	/	/
EdgeScale - Edge computing	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/
Hugetlbfs	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Management Complex	na	na	na	na	na	Y	Y	Y	/	/	/	/	/	/	/	/	/	/	/	/
Open Portable Trust Execution Environment (OP-TEE)	Y	/	Y	Y	Y	Y	Y	Y	/	/	/	/	/	/	/	/	/	/	/	/
Secure boot	Y	Y	Y	Y	Y	Y	Y	Y	/	/	/	/	/	/	/	/	/	/	/	/
Time sensitive network (TSN)	na	na	Y	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
Unified Extensible Firmware Interface (UEFI)	/	/	/	Y	Y	/	Y	Y	/	/	/	/	/	/	/	/	/	/	/	/
USDPAA applications	na	na	na	/	/	na	na	na	na	na	na	na	/	/	/	/	/	/	/	/
Trusted Firmware-A (TF-A)	Y	/	Y	Y	Y	Y	Y	Y	na	na	na	na	na	na	na	na	na	na	na	na

Table 3. Virtualization features

Feature	LS1012A	LS1021A	LS1028A	LS1043A	LS1046A	LS1088A	LS2088A	LX2160A	MPC8548	P1010	P1020	P2020	P2041	P3041	P4080	P5040	T1024	T1042	T2080	T4240
KVM/QEMU	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
LXC	Y	Y	Y	Y	Y	Y	Y	Y	/	/	/	/	/	/	/	/	/	/	/	/
Libvirt	Y	Y	Y	Y	Y	Y	Y	Y	/	/	/	/	/	/	/	/	/	/	/	/
Network interface direct assignment	na	na	na	na	na	Y	Y	Y	na	na	na	na	na	na	na	na	na	na	na	na

Table continues on the next page...

Table 3. Virtualization features (continued)

Feature	LS1012A	LS1021A	LS1028A	LS1043A	LS1046A	LS1088A	LS2088A	LX2160A	MPC8548	P1010	P1020	P2020	P2041	P3041	P4080	P5040	T1024	T1042	T2080	T4240
VFIO	na	na	na	na	na	Y	Y	Y	na	na	na	na	na	na	na	na	na	na	na	na
Docker	Y	/	Y	Y	Y	Y	Y	Y	/	/	/	/	/	/	/	/	/	/	/	/

Table 4. Linux kernel driver features

Feature	LS1012A	LS1021A	LS1028A	LS1043A	LS1046A	LS1088A	LS2088A	LX2160A	MPC8548	P1010	P1020	P2020	P2041	P3041	P4080	P5040	T1024	T1042	T2080	T4240
Audio - I2S, SAI	Y	Y	Y	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
CAAM DMA	Y	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/
DCE	na	na	na	na	na	na	Y	Y	na	na	na	na	na	na	na	na	na	na	na	na
DCU	na	Y	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
Display - eDPNDP, LCD	na	na	Y	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
eLBC	na	na	na	na	na	na	na	na	na	na	Y	Y	Y	Y	Y	Y	na	na	na	na
DMA	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
DPAA1 - Ethernet, FMan, QMan, BMan	na	na	na	Y	Y	na	na	na	na	na	na	na	Y	Y	Y	Y	Y	Y	Y	Y
DPAA2 - Ethernet, L2Switching, QBMan	na	na	na	na	na	Y	Y	Y	na	na	na	na	na	na	na	na	na	na	na	na
DSPI	/	Y	/	Y	Y	na	Y	na	na	na	na	na	na	na	na	na	na	na	na	na
eSDHC	Y	Y	Y	Y	Y	Y	Y	Y	na	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
eSPI	na	na	na	na	na	na	na	na	na	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
ENETC	na	na	Y	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
FlexCAN	na	Y	Y	na	na	na	na	Y	na	na	na	na	na	na	na	na	na	na	na	na
FlexSPI	na	na	Y	na	na	na	na	Y	na	na	na	na	na	na	na	na	na	na	na	na
GPU	na	na	Y	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
I2C	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
IEEE1588, linuxptp	/	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
IFC	na	Y	na	Y	Y	Y	Y	na	na	na	na	na	na	na	na	na	Y	Y	Y	Y
TSN Ethernet switch	na	na	Y	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
LPUART	na	Y	/	/	/	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
QSPI	Y	Y	na	Y	Y	Y	Y	na	na	na	na	na	na	na	na	na	na	na	na	na

Table continues on the next page...

Table 4. Linux kernel driver features (continued)

Feature	LS1012A	LS1021A	LS1028A	LS1043A	LS1046A	LS1088A	LS2088A	LX2160A	MPC8548	P1010	P1020	P2020	P2041	P3041	P4080	P5040	T1024	T1042	T2080	T4240
PCIe root complex	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
PCIe endpoint	/	/	/	/	Y	/	/	Y	/	/	/	/	/	/	/	/	/	/	/	/
PFE	Y	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
Power management	Y	Y	Y	Y	Y	Y	Y	Y	/	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Preempt Real-Time	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/
SATA	Y	Y	Y	Y	Y	Y	Y	Y	na	na	na	na	Y	Y	Y	Y	Y	Y	Y	Y
SEC	Y	Y	Y	Y	Y	Y	Y	Y	na	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
TDM (QE)	na	na	na	Y	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
TSN	na	na	Y	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
USB	Y	Y	Y	Y	Y	Y	Y	Y	na	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
VeTSEC	na	Y	na	na	na	na	na	na	na	Y	Y	Y	na	na	na	na	na	na	na	na
VFIO for network resources	na	na	na	na	na	Y	Y	Y	na	na	na	na	na	na	na	na	na	na	na	na
Watchdog	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

2.3 Supported targets

The table below explains which processors and development boards are supported in different LSDK Yocto releases. In this table:

- 'Y' indicates that a development board is supported in an LSDK Yocto release
- 'N' indicates that a development board is not supported in an LSDK Yocto release

Processor	Board	YP 2.5–LSDK 1806	YP 2.6–LSDK 1809	YP 2.7–LSDK 1906	YP 3.0–LSDK 1909	YP 3.1–LSDK 2004
LS1012A (rev1.0 and rev2.0)	LS1012ARDB	Y	Y	Y	Y	Y
	FRWY-LS1012A	Y	Y	Y	Y	Y
LS1021A/LS1020A (rev2.0)	TWR-LS1021A	Y	Y	Y	Y	Y
LS1028A/LS1027A (rev1.0)	LS1028ARDB-PA	N	N	N	Y	Y
LS1043A/LS1023A (rev 1.1)	LS1043ARDB-PC	Y	Y	Y	Y	Y
	LS1043ARDB-PD	Y	Y	Y	Y	Y
LS1046A/LS1026A (rev1.0)	LS1046ARDB-PB	Y	Y	Y	Y	Y
	FRWY-LS1046A	N	N	Y	Y	Y
LS1088A (rev1.0)	LS1088ARDB	Y	N	N	N	N

Table continues on the next page...

Table continued from the previous page...

Processor	Board	YP 2.5–LSDK 1806	YP 2.6–LSDK 1809	YP 2.7–LSDK 1906	YP 3.0–LSDK 1909	YP 3.1–LSDK 2004
	LS1088ARDB-PB	Y	Y	Y	Y	Y
LS2088A/LS2084A/ LS2081A (rev1.0 and rev1.1)	LS2088ARDB	Y	Y	Y	Y	Y
LX2160A (rev1.0 and rev2.0)	LX2160ARDB	N	N	Y (only rev1.0)	Y (only rev1.0)	Y (both)
MPC8548 (rev3.1)	MPC8548CDS	Y	Y	Y	Y	Y
P1010 (rev2.01)	P1010RDB-PB	N	N	N	N	Y
P1020 (rev1.1)	P1020RDB-PD	Y	Y	Y	N	Y
P2020 (rev2.1)	P2020RDB-PCA	Y	Y	Y	Y	Y
P2041 (rev2.0)	P2041RDB-PC	Y	Y	Y	Y	Y
P3041 (rev2.0)	P3041DS-PC	Y	Y	Y	Y	Y
P4080 (rev3.0)	P4080DS	Y	Y	Y	Y	Y
P5040 (rev2.1)	P5040DS	Y	Y	Y	Y	Y
T1024 (rev1.0)	T1024RDB-PC	Y	Y	Y	Y	Y
T1042 (rev1.1)	T1042D4RDB-PA	Y	Y	Y	Y	Y
T2080 (rev1.1)	T2080RDB-PC	Y	Y	Y	Y	Y
T4240 (rev2.0)	T4240RDB-PB	Y	Y	Y	Y	Y

2.4 Fixed, open, and closed issues

The table below lists the issues fixed in current LSDK Yocto release.

NOTE

For the issues fixed in Linux, U-Boot, and other components, see "Fixed, open, and closed issues" section in "Release Notes" chapter of *Layerscape Software Development Kit User Guide* available at the following link:

https://www.nxp.com/design/software/embedded-software/linux-software-and-development-tools/layerscape-software-development-kit:LAYERSCAPE-SDK?tab=Documentation_Tab

Table 5. Fixed issues

ID	Description	Disposition	Opened in	Fixed in
QYOCTO-583	crconf update command cannot finish by itself	Fixed	Yocto 3.0	Yocto 3.1
QYOCTO-629	There is no cmake command in Yocto 3.0 full rootfs, XDP build on board failed	Fixed	Yocto 3.0	Yocto 3.1

The table below lists the open issues in current LSDK Yocto release.

NOTE

For the open issues in Linux, U-Boot, and other components, see "Fixed, open, and closed issues" section in "Release Notes" chapter of *Layerscape Software Development Kit User Guide* available at the following link:

https://www.nxp.com/design/software/embedded-software/linux-software-and-development-tools/layerscape-software-development-kit:LAYERSCAPE-SDK?tab=Documentation_Tab

Table 6. Open issues

ID	Description	Disposition	Opened in	Workaround
QUBOOT-5518	Optical XFI2 port does not work on T2080RDB	Open	Yocto 3.0	
QYOCTO-586	Guest rootfs boot failed on T2080RDB and T4240RDB with ext2.gz on Yocto 2.6, Yocto 2.7, Yocto 3.0, and Yocto 3.1	Open	Yocto 3.0	

3 Download Yocto layers

Before starting to download Yocto, prepare the build host for running and building Yocto. To prepare the build host environment, follow the instructions provided at the link below:

https://www.yoctoproject.org/docs/3.1/brief-yoctoprojectqs/brief-yoctoprojectqs.html

The subsections that follow provide two methods for downloading Yocto layers. You can use one of these methods (preferably Download Yocto layers according to repo manifest) to download Yocto layers.

3.1 Download Yocto layers according to repo manifest

To download all Yocto layers as per repo manifest using repo utility, perform these steps:

1. Install the repo utility:

```
$: mkdir ~/bin
$: curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
$: chmod a+x ~/bin/repo
```

2. Download the Yocto layers:

```
$: export PATH=${PATH}:~/bin
$: mkdir yocto-sdk
$: cd yocto-sdk
$: repo init -u https://source.codeaurora.org/external/qoriq/qoriq-components/yocto-sdk -b
dunfell
$: repo sync --no-clone-bundle
```

3.2 Download Yocto layers from community repository

As an alternative method, use the following steps to download Yocto layers from community repository:

1. To download all Yocto layers from community repository, run the following Git commands:

```
$ mkdir yocto-sdk
$ cd yocto-sdk
$ mkdir sources
$ git clone git://git.yoctoproject.org/poky sources/poky
$ git clone git://git.yoctoproject.org/meta-cloud-services sources/meta-cloud-services
$ git clone git://git.yoctoproject.org/meta-freescale sources/meta-freescale
$ git clone git://git.yoctoproject.org/meta-security sources/meta-security
```

```
$ git clone git://git.yoctoproject.org/meta-selinux sources/meta-selinux
$ git clone git://git.yoctoproject.org/meta-virtualization sources/meta-virtualization
$ git clone https://github.com/OSSystems/meta-browser sources/meta-browser
$ git clone https://github.com/kraj/meta-clang sources/meta-clang
$ git clone https://github.com/Freescale/meta-freescale-distro sources/meta-freescale-distro
$ git clone https://github.com/openembedded/meta-openembedded sources/meta-openembedded
$ git clone https://github.com/TimesysGit/meta-timesys sources/meta-timesys
$ git clone https://source.codeaurora.org/external/qoriq/qoriq-components/meta-qoriq sources/
meta-qoriq
```

2. Reset the revision of each Yocto layer as mentioned in the following manifest file:

   https://source.codeaurora.org/external/qoriq/qoriq-components/yocto-sdk/tree/default.xml?h=dunfell

3. Finally, copy the setup script:

```
$ cp sources/meta-qoriq/tools/setup-env .
```

4 Build Yocto images

The steps for building Yocto images are same for all boards with board name being the only change. For example, the commands for building Yocto images for LS2088ARDB are as follows:

```
$ cd yocto-sdk
$: . ./setup-env -m ls2088ardb
$: bitbake fsl-image-networking
$: bitbake fsl-image-networking-full
```

NOTE

After running the above commands, you will get the LS2088ARDB Yocto images in the yocto-sdk/build_ls2088ardb/tmp/deploy/images/ls2088ardb/ folder.

Build fsl-image-multimedia-full on LS1028ARDB

To build fsl-image-multimedia-full on the LS1028ARDB, follow these steps:

1. In the conf/local.conf file, uncomment the line that removes distro feature x11 (marked as bold in the below code listing) as the LS1028ARDB uses wayland:

```
#DISTRO_FEATURES_remove = " x11"
#PREFERRED_VERSION_weston = "8.0.0.imx"
#PREFERRED_VERSION_wayland-protocols = "1.18.imx"
#PREFERRED_VERSION_libdrm = "2.4.99.imx"
#PREFERRED_PROVIDER_virtual/libgl = "imx-gpu-viv"
#PREFERRED_PROVIDER_virtual/libgles1 = "imx-gpu-viv"
#PREFERRED_PROVIDER_virtual/libgles2 = "imx-gpu-viv"
#PREFERRED_PROVIDER_virtual/egl = "imx-gpu-viv"
```

2. Run the following command:

```
bitbake fsl-image-multimedia-full
```

5 Boot boards with Yocto image

5.1 Prerequisites

Before initiating to boot a board with a Yocto image, ensure that:

- The TFTP server is set up for image download
- A serial cable is connected from your PC to UART1 port of the board
- An Ethernet cable is connected to the first Ethernet port on the board

5.2 Booting board with RAM disk rootfs image

To boot a board with RAM disk rootfs image, perform these steps:

NOTE

For more information on a board, see "LSDK Quick Start Guide" section for the board in "Layerscape SDK user guide" chapter of *Layerscape Software Development Kit User Guide* available at the following link:

https://www.nxp.com/design/software/embedded-software/linux-software-and-development-tools/layerscape-software-development-kit:LAYERSCAPE-SDK?tab=Documentation_Tab

1. Power up or reset the board.
2. Press a key on the keyboard when prompted on the terminal to enter into the U-Boot command line.
3. Set up the environment in U-Boot:

```
=> setenv ipaddr <board_ipaddr>
=> setenv serverip <tftp_serverip>
```

The next command in setting up the environment in U-Boot is described for different boards in the table below.

Board	Command
FRWY-LS1012A LS1012ARDB	`=> setenv bootargs root=/dev/ram0 rw console=ttyS0,115200 earlycon=uart8250,mmio,0x21c0500 ramdisk_size=0x10000000`
TWR-LS1021A	`=> setenv bootargs root=/dev/ram0 rw console=ttyS0,115200 ramdisk_size=0x1000000`
LS1028ARDB	`=>setenv bootargs root=/dev/ram0 rw console=ttyS0,115200 earlycon=uart8250,mmio,0x21c0500 default_hugepagesz=2m hugepagesz=2m hugepages=256 ramdisk_size=100000000 video=1920x1080-32@60 cma=256M`
LS1043ARDB	`=> setenv bootargs root=/dev/ram0 rw console=ttyS0,115200 earlycon=uart8250,mmio,0x21c0500 ramdisk_size=0x10000000`
LS1046ARDB FRWY-LS1046A	`=> setenv bootargs root=/dev/ram0 rw console=ttyS0,115200 earlycon=uart8250,mmio,0x21c0500 ramdisk_size=0x10000000`
LS1088ARDB-PB	`=> setenv bootargs root=/dev/ram0 rw console=ttyS0,115200 earlycon=uart8250,mmio,0x21c0500 ramdisk_size=0x2000000 default_hugepagesz=2m hugepagesz=2m hugepages=512`

Table continues on the next page...

Table continued from the previous page...

Board	Command
LS2088ARDB	`=> setenv bootargs root=/dev/ram0 rw console=ttyS1,115200 earlycon=uart8250,mmio,0x21c0600 ramdisk_size=0x2000000 default_hugepagesz=1024m hugepagesz=1024m hugepages=2`
LX2160ARDB (rev1.0 and rev2.0)	`=> setenv bootargs console=ttyAMA0,115200 root=/dev/ram0 rw earlycon=pl011,mmio32,0x21c0000 ramdisk_size=0x2000000 default_hugepagesz=1024m hugepagesz=1024m hugepages=2 pci=pcie_bus_perf`
MPC8548CDS	`=> setenv bootargs root=/dev/ram rw console=ttyS1,115200 ramdisk_size=1000000 log_buf_len=128K`
Other PPC targets	`=> setenv bootargs root=/dev/ram rw console=ttyS0,115200 ramdisk_size=1000000 log_buf_len=128K`

4. For LS1088ARDB-PB, LS2088ARDB, or LX2160ARDB, enable DPAA2 Ethernet in Linux using commands described in the table below.

Board	Commands

LS1088ARDB-PB	`=> sf probe 0:0`
	`=> sf read 0x80000000 0xA00000 0x200000`
	`=> sf read 0x80200000 0xE00000 0x100000`
	`=> fsl_mc start mc 0x80000000 0x80200000`
	`=> sf read 0x80200000 0xd00000 0x100000`
	`=> fsl_mc lazyapply dpl 0x80200000`

LS2088ARDB	`=> fsl_mc start mc 0x580a00000 0x580e00000` `=> fsl_mc lazyapply dpl 0x580d00000`

LX2160ARDB	`=> sf probe 0:0`
	`=> sf read 0x80a00000 0xa00000 0x300000`
	`=> sf read 0x80e00000 0xe00000 0x100000`
	`=> fsl_mc start mc 0x80a00000 0x80e00000`
	`=> sf read 0x80d00000 0xd00000 0x100000`
	`=> fsl_mc lazyapply dpl 0x80d00000`

5. Download images and boot up the board using commands described in the table below.

Board	Commands

FRWY-LS1012A
LS1012ARDB
	`=> pci enum`
	`=> tftp 0x84080000 Image-<board>.bin`
	`=> tftp 0x88000000 fsl-image-networking-<board>.ext2.gz.u-boot`

Table continues on the next page...

Table continued from the previous page...

Board	Commands
	FRWY-LS1012A: => tftp 0x8f000000 fsl-ls1012a-frwy.dtb LS1012ARDB: => tftp 0x8f000000 fsl-ls1012a-rdb.dtb => pfe stop => booti 0x84080000 0x88000000 0x8f000000
TWR-LS1021A	=> tftp 0x82000000 uImage-ls1021atwr.bin => tftp 0x88000000 fsl-image-networkingls1021atwr.ext2.gz.u-boot => tftp 0x8f000000 uImage-ls1021a-twr.dtb => bootm 0x82000000 0x88000000 0x8f000000
LS1028ARDB LS1043ARDB LS1046ARDB FRWY-LS1046A LS1088ARDB-PB LS2088ARDB LX2160ARDB	=> tftp 0x82000000 Image-<board>.bin => tftp 0xa0000000 fsl-image-networking-<board>.ext2.gz.u-boot LS1028ARDB: => tftp 0x8f000000 fsl-ls1028a-rdb.dtb LS1043ARDB: => tftp 0x8f000000 fsl-ls1043a-rdb-sdk.dtb LS1046ARDB: => tftp 0x8f000000 fsl-ls1046a-rdb-sdk.dtb FRWY-LS1046A: => tftp 0x8f000000 fsl-ls1046a-frwy-sdk.dtb LS1088ARDB-PB: => tftp 0x8f000000 fsl-ls1088a-rdb.dtb LS2088ARDB: => tftp 0x8f000000 fsl-ls2088a-rdb.dtb LX2160ARDB: => tftp 0x8f000000 fsl-lx2160a-rdb.dtb => booti 0x82000000 0xa0000000 0x8f000000
MPC8548CDS	=> tftpboot 0x01000000 uImage-mpc8548cds.bin => tftpboot 0x03000000 fsl-image-networkingmpc8548cds.ext2.gz.u-boot => tftpboot 0x02000000 uImagempc8548cds_32b.dtb => bootm 0x01000000 0x03000000 0x02000000
P1020RDB P2020RDB P2041RDB	=> tftpboot 0x01000000 uImage-<board>.bin => tftpboot 0x04000000 fsl-image-networking-<board>.ext2.gz.u-boot => tftpboot 0x02000000 uImage-<board>.dtb => bootm 0x01000000 0x04000000 0x02000000

Table continues on the next page...

Table continued from the previous page...

Board	Commands
P3041DS P4080DS P5040DS	

T1024RDB
T1042D4RDB
T2080RDB (64-bit)
T4240RDB (64-bit)
	`=> tftpboot 0x01000000 uImage-<board>.bin`
	`=> tftpboot 0x05000000 fsl-image-networking-<board>.ext2.gz.u-boot`
	`=> tftpboot 0x02000000 uImage-<board>.dtb`
	`=> bootm 0x01000000 0x05000000 0x02000000`

NOTE

For flashing separate firmware images on alternate bank, find memory details for different firmware in QorIQ memory layout.

5.3 Booting board with full rootfs from large storage device

To boot a board with full rootfs from large storage device, such as SD card, USB mass storage device, or SATA device, perform these steps:

NOTE

For more information on a board, see "LSDK Quick Start Guide" section for the board in "Layerscape SDK user guide" chapter of *Layerscape Software Development Kit User Guide* available at the following link:

https://www.nxp.com/design/software/embedded-software/linux-software-and-development-tools/layerscape-software-development-kit:LAYERSCAPE-SDK?tab=Documentation_Tab

1. Prepare the media (SD/USB/SATA) and format it to ext2 format.
2. Mount the ext2 partition, extract a full rootfs into this partition, and unmount the partition.
3. Power up or reset the board.
4. Press a key on the keyboard when prompted on the terminal to enter into the U-Boot command line.
5. Set up the environment in U-Boot using command described in the table below.

Board	Command
FRWY-LS1012A LS1012ARDB LS1043ARDB LS1046ARDB FRWY-LS1046A	`=> setenv bootargs root=/dev/sda* rootdelay=5 rw console=ttyS0,115200 earlycon=uart8250,mmio,0x21c0500`
TWR-LS1021A	`=> setenv bootargs root=/dev/sda* rootdelay=5 rw console=ttyS0,115200`

Table continues on the next page...

Table continued from the previous page...

Board	Command
LS1028ARDB	`=> setenv bootargs root=/dev/mmcblk0p4 rw rootdelay=5 console=ttyS0,115200 earlycon=uart8250,mmio,0x21c0500 video=3840x2160-32@60 cma=256M default_hugepagesz=2m hugepagesz=2m hugepages=256`
LS1088ARDB-PB	`=> setenv bootargs root=/dev/sda* rootdelay=5 rw console=ttyS0,115200 earlycon=uart8250,mmio,0x21c0500 default_hugepagesz=2m hugepagesz=2m hugepages=512`
LS2088ARDB	`=> setenv bootargs root=/dev/sda* rootdelay=5 rw console=ttyS1,115200 earlycon=uart8250,mmio,0x21c0600 default_hugepagesz=1024m hugepagesz=1024m hugepages=8`
LX2160ARDB	`=> setenv bootargs console=ttyAMA0,115200 root=/dev/sda* rw rootdelay=10 earlycon=pl011,mmio32,0x21c0000 ramdisk_size=0x2000000 default_hugepagesz=1024m hugepagesz=1024m hugepages=2 pci=pcie_bus_perf`

6. For LS1088ARDB-PB, LS2088ARDB, or LX2160ARDB, enable DPAA2 Ethernet in Linux using commands described in the table below.

Board	Commands

LS1088ARDB-PB	`=> sf probe 0:0`
	`=> sf read 0x80000000 0xA00000 0x200000`
	`=> sf read 0x80200000 0xE00000 0x100000`
	`=> fsl_mc start mc 0x80000000 0x80200000`
	`=> sf read 0x80200000 0xd00000 0x100000`
	`=> fsl_mc lazyapply dpl 0x80200000`

LS2088ARDB	`=> fsl_mc start mc 0x580a00000 0x580e00000` `=> fsl_mc lazyapply dpl 0x580d00000`

LX2160ARDB	`=> sf probe 0:0`
	`=> sf read 0x80a00000 0xa00000 0x300000`
	`=> sf read 0x80e00000 0xe00000 0x100000`
	`=> fsl_mc start mc 0x80a00000 0x80e00000`
	`=> sf read 0x80d00000 0xd00000 0x100000`
	`=> fsl_mc lazyapply dpl 0x80d00000`

7. Download images and boot up the board using commands described in the table below.

Board	Commands
FRWY-LS1012A LS1012ARDB	=> pci enum => tftp 0x84080000 Image-<board>.bin FRWY-LS1012A: => tftp 0x8f000000 fsl-ls1012a-frwy.dtb LS1012ARDB: => tftp 0x8f000000 fsl-ls1012a-rdb.dtb => pfe stop => booti 0x84080000 - 0x8f000000
TWR-LS1021A	=> tftp 0x82000000 uImage-ls1021atwr.bin => tftp 0x8f000000 uImage-ls1021a-twr.dtb => bootm 0x82000000 - 0x8f000000
LS1028ARDB LS1043ARDB LS1046ARDB FRWY-LS1046A LS1088ARDB-PB LS2088ARDB LX2160ARDB	=> tftp 0x82000000 Image-<board>.bin LS1028ARDB: => tftp 0x8f000000 fsl-ls1028a-rdb.dtb LS1043ARDB: => tftp 0x8f000000 fsl-ls1043a-rdb-sdk.dtb LS1046ARDB: => tftp 0x8f000000 fsl-ls1046a-rdb-sdk.dtb FRWY-LS1046A: => tftp 0x8f000000 fsl-ls1046a-frwy-sdk.dtb LS1088ARDB-PB: => tftp 0x8f000000 fsl-ls1088a-rdb.dtb LS2088ARDB: => tftp 0x8f000000 fsl-ls2088a-rdb.dtb LX2160ARDB: => tftp 0x8f000000 fsl-lx2160a-rdb.dtb => booti 0x82000000 - 0x8f000000

5.4 Secure boot

To build a secure boot image, follow these steps:

1. Install the repo utility (follow instructions provided in Download Yocto layers according to repo manifest).
2. Download all Yocto layers according to repo manifest using the repo utility:

   ```
   $: mkdir yocto-sdk
   $: cd yocto-sdk
   $: repo init -u ssh://git@bitbucket.sw.nxp.com/lfac/lf-sdk.git -b <branch>
   $: repo sync
   ```

   where <branch> indicates the Yocto branch (find Yocto branch and version details in Table 1).

3. Build <machine> directory:

```
source ./setup-env -m <machine>
```

where <machine> represents one of the following values:

- ls1012ardb
- ls1012afrwy
- ls1021atwr
- ls1043ardb
- ls1046ardb
- ls1046afrwy
- ls1088ardb-pb
- ls1028ardb
- ls2088ardb
- lx2160ardb-rev2 (only LX2160ARDB with LX2160A silicon revision 2 is supported)
- lx2162aqds

4. Set the following variable in the conf/local.conf file:

```
DISTRO_FEATURES_append = ' secure'
```

5. Build the secure boot image:

```
$: bitbake secure-boot-qoriq
```

Secure images with RSA key pairs (srk.pri, srk.pub) get generated along with srk_hash.txt file under the tmp/deploy/images/<board> directory.

To enable secure boot on QorIQ (Layerscape) platforms, see section 6.1.1 ("Secure boot") of *Layerscape Software Development Kit User Guide* available at the following link:

https://www.nxp.com/design/software/embedded-software/linux-software-and-development-tools/layerscape-software-development-kit:LAYERSCAPE-SDK?tab=Documentation_Tab

6 Program TF-A binaries

The table below describes boot types supported for programming TF-A binaries in different Layerscape boards.

Table 7. Supported boot types

Board	QSPI NOR flash	FlexSPI NOR flash	IFC NOR flash	NAND flash	SD card	eMMC
FRWY-LS1012A	Y					
LS1012ARDB	Y					
LS1028ARDB		Y			Y	Y
LS1043ARDB			Y	Y	Y	
FRWY-LS1046A	Y				Y	
LS1046ARDB	Y				Y	

Table continues on the next page...

Table 7. Supported boot types (continued)

Board	QSPI NOR flash	FlexSPI NOR flash	IFC NOR flash	NAND flash	SD card	eMMC
LS1088ARDB-PB	Y				Y	
LS2088ARDB			Y			
LX2160ARDB		Y			Y	Y

6.1 Program TF-A binaries on QSPI NOR flash

To program TF-A binaries on QSPI NOR flash, follow these steps:

1. Boot the board from QSPI NOR flash 0.
2. Program QSPI NOR flash 1:

```
=> sf probe 0:1
```

3. Flash bl2_qspi.pbl:

```
=> tftp 0xa0000000 bl2_qspi.pbl
=> sf erase 0x0 +$filesize && sf write 0xa0000000 0x0 $filesize
```

4. Flash fip_uboot.bin:

```
=> tftp 0xa0000000 fip_uboot.bin
=> sf erase 0x100000 +$filesize && sf write 0xa0000000 0x100000 $filesize
```

5. Boot the board from QSPI NOR flash 1. The board boots with TF-A.

6.2 Program TF-A binaries on FlexSPI NOR flash

FlexSPI NOR flash is supported as a booting option in LX2160A and LS1028A.

For LX2160A, the steps to program TF-A binaries on FlexSPI NOR flash are as follows:

1. Boot the board from FlexSPI NOR flash 0.
2. Program FlexSPI NOR flash 1:

```
=> sf probe 0:1
```

3. Flash bl2_flexspi_nor.pbl:

```
=> tftp 0xa0000000 bl2_flexspi_nor.pbl
=> sf erase 0x0 +$filesize && sf write 0xa0000000 0x0 $filesize
```

4. Flash fip_uboot.bin:

```
=> tftp 0xa0000000 fip_uboot.bin
=> sf erase 0x100000 +$filesize && sf write 0xa0000000 0x100000 $filesize
```

5. Boot the board from FlexSPI NOR flash 1. The board boots with TF-A.

For LS1028A, the steps to program TF-A binaries on FlexSPI NOR flash are as follows:

1. Boot the board from FlexSPI NOR flash 0.

2. Program FlexSPI NOR flash 0 (LS1028A has only one bank):

```
=> sf probe 0:0
```

3. Flash bl2_flexspi_nor.pbl:

```
=> tftp 0x80000000 bl2_flexspi_nor.pbl
=> sf erase 0x0 +$filesize && sf write 0x80000000 0x0 $filesize
```

4. Flash fip_uboot.bin:

```
=> tftp 0x80000000 fip_uboot.bin
=> sf erase 0x100000 +$filesize && sf write 0x80000000 0x100000 $filesize
```

5. Flash DP firmware:

```
=> tftp 0x80000000 ls1028ardb/dp/ls1028a-dp-fw.bin
=> sf erase 0x900000 +$filesize && sf write 0x80000000 0x900000 $filesize
```

6. Boot the board from FlexSPI NOR flash 0. The board boots with TF-A.

6.3 Program TF-A binaries on IFC NOR flash

IFC NOR flash is supported as a booting option in LS1043A and LS2088A.

For LS1043A, the steps to program TF-A binaries on IFC NOR flash are as follows:

1. Boot the board from default bank.
2. Flash bl2_nor.pbl to alternate bank:

```
=> tftp 0x82000000 $path/bl2_nor.pbl
=> pro off all; erase 0x64000000 +$filesize; cp.b 0x82000000 0x64000000 $filesize
```

3. Flash fip_uboot.bin to alternate bank:

```
=> tftp 0x82000000 $path/fip_uboot.bin
=> pro off all; erase 0x64100000 +$filesize; cp.b 0x82000000 0x64100000 $filesize
```

4. Boot the board from alternate bank. The board boots with TF-A.

For LS2088A, the steps to program TF-A binaries on IFC NOR flash are as follows:

1. Boot the board from default bank.
2. Flash bl2_nor.pbl to alternate bank:

```
=> tftp 0x82000000 $path/bl2_nor.pbl
=> pro off all; erase 0x584000000 +$filesize; cp.b 0x82000000 0x584000000 $filesize
```

3. Flash fip_uboot.bin to alternate bank:

```
=> tftp 0x82000000 $path/fip_uboot.bin
=> pro off all; erase 0x584100000 +$filesize; cp.b 0x82000000 0x584100000 $filesize
```

4. Boot the board from alternate bank. The board boots with TF-A.

6.4 Program TF-A binaries on NAND flash

To program TF-A binaries on NAND flash, follow these steps:

1. Boot the board from default bank.
2. Flash bl2_nand.pbl to NAND flash:

```
=> tftp 82000000 $path/bl2_nand.pbl
=> nand erase 0x0 $filesize;nand write 0x82000000 0x0 $filesize;
```

3. Flash fip_uboot.bin to NAND flash:

```
=> tftp 82000000 $path/fip_uboot.bin
=> nand erase 0x100000 $filesize;nand write 0x82000000 0x100000 $filesize;
```

4. Boot the board from NAND flash. The board boots with TF-A.

6.5 Program TF-A binaries on SD card

To program TF-A binaries on SD card, follow these steps:

1. Boot the board from default bank.
2. Flash bl2_sd.pbl to SD card:

```
=> tftp 82000000 bl2_sd.pbl
=> mmc write 82000000 8 <blk_cnt>
```

   where <blk_cnt> is the number of blocks in SD card that must be written. It is calculated based on file size. For example, if bl2_sd.pbl is loaded from the TFTP server and the number of bytes transferred is 82809 (14379 hex), then <blk_cnt> is calculated as:

   82809/512 = 161 (A1 hex)

   For this example, `mmc write` command becomes:

```
=> mmc write 82000000 8 A1
```

3. Flash fip_uboot.bin to SD card:

```
=> tftp 82000000 fip_uboot.bin
=> mmc write 82000000 800 <blk_cnt>
```

4. Boot the board from SD card. The board boots with TF-A.

6.6 Program TF-A binaries on eMMC device

To program TF-A binaries on eMMC device, follow these steps:

1. Boot the board from default bank.
2. Select eMMC device:

```
=> mmc dev 1
```

3. Flash bl2_emmc.pbl to eMMC device:

```
=> tftp 82000000 bl2_emmc.pbl
=> mmc write 82000000 8 <blk_cnt>
```

   where <blk_cnt> is the number of blocks in eMMC that must be written. It is calculated based on file size. For example, if bl2_emmc.pbl is loaded from the TFTP server and the number of bytes transferred is 82809 (14379 hex), then <blk_cnt> is calculated as:

82809/512 = 161 (A1 hex)

For this example, `mmc write` command becomes:

```
=> mmc write 82000000 8 A1
```

4. Flash fip_uboot.bin to eMMC device:

```
=> tftp 82000000 fip_uboot.bin
=> mmc write 82000000 800 <blk_cnt>
```

5. Boot the board from eMMC device. The board boots with TF-A.

7 QorIQ memory layout

The following table shows the memory layout of various firmware stored in NOR/NAND/QSPI/XSPI flash device or SD card on all QorIQ Reference Design Boards.

NOTE

When the board boots from NOR flash, the NOR bank it loads boot image from is called "current bank" and the other bank is called "alternate bank". For example, if LS1043ARDB boots from NOR bank 4, then to update an image on NOR bank 0, you have to use the "alternate bank" address range, `0x64000000 - 0x64F00000`.

Table 8. Unified 64 MiB memory layout of NOR/QSPI/XSPI/NAND/SD media

Firmware definition	Max size	Flash Offset (QSPI/XSPI/ NAND flash)	Absolute address (NOR current bank on LS1043ARDB, LS1021ATWR)	Absolute address (NOR alternate bank on LS1043ARDB, LS1021ATWR)	Absolute address (NOR current bank on LS2088ARDB)	Absolute address (NOR alternate bank on LS2088ARDB)	SD start block no.
RCW + PBI + BL2 (bl2_<boot_mode>.pbl)[1]	1 MiB	0x00000000	0x60000000	0x64000000	0x580000000	0x584000000	0x00008
TF-A FIP image (BL31 + TEE (BL32) + U-Boot/ UEFI (Bl33)) (fip.bin)[2]	4 MiB	0x00100000	0x60100000	0x64100000	0x580100000	0x584100000	0x00800
Boot firmware environment	1 MiB	0x00500000	0x60500000	0x64500000	0x580500000	0x584500000	0x02800
Secure boot headers	2 MiB	0x00600000	0x60600000	0x64600000	0x580600000	0x584600000	0x03000
Secure header or DDR PHY FW	512 KiB	0x00800000	0x60800000	0x64800000	0x580800000	0x584800000	0x04000
Fuse provisioning header	512 KiB	0x00880000	0x60880000	0x64880000	0x580880000	0x584880000	0x04400
DPAA1 FMan microcode	256 KiB	0x00900000	0x60900000	0x64900000	0x580900000	0x584900000	0x04800

Table continues on the next page...

Table 8. Unified 64 MiB memory layout of NOR/QSPI/XSPI/NAND/SD media (continued)

Firmware definition		Max size	Flash Offset (QSPI/XSPI/NAND flash)	Absolute address (NOR current bank on LS1043ARDB, LS1021ATWR)	Absolute address (NOR alternate bank on LS1043ARDB, LS1021ATWR)	Absolute address (NOR current bank on LS2088ARDB)	Absolute address (NOR alternate bank on LS2088ARDB)	SD start block no.
QE firmware or DP firmware		256 KiB	0x00940000	0x60940000	0x64940000	0x580940000	0x584940000	0x04A00
Ethernet PHY firmware		256 KiB	0x00980000	0x60980000	0x64980000	0x580980000	0x584980000	0x04C00
Script for flashing image		256 KiB	0x009C0000	0x609C0000	0x649C0000	0x5809C0000	0x5849C0000	0x04E00
DPAA2-MC or PFE firmware		3 MiB	0x00A00000	0x60A00000	0x64A00000	0x580A00000	0x584A00000	0x05000
DPAA2 DPL		1 MiB	0x00D00000	0x60D00000	0x64D00000	0x580D00000	0x584D00000	0x06800
DPAA2 DPC		1 MiB	0x00E00000	0x60E00000	0x64E00000	0x580E00000	0x584E00000	0x07000
Device tree (needed by UEFI)		1 MiB	0x00F00000	0x60F00000	0x64F00000	0x580F00000	0x584F00000	0x07800
Kernel	lsdk_linux_<arch>_LS_tiny.itb	16 MiB	0x01000000	0x61000000	0x65000000	0x581000000	0x585000000	0x08000
RAM disk rfs		32 MiB	0x02000000	0x62000000	0x66000000	0x582000000	0x586000000	0x10000

1. For any update in the BL2 source code or RCW binary, the `bl2_<boot_mode>.pbl` binary must be recompiled (see Layerscape Software Development Kit User Guide for more information).
2. For any update in the BL31, BL32, or BL33 binaries, the `fip.bin` binary needs to be recompiled (see Layerscape Software Development Kit User Guide for more information).

Table 9. 2 MB memory layout of QSPI/SD media on LS1012AFRWY

Firmware definition	Max. size	Location	SD start block no.
RCW+PBI+BL2 (bl2_<boot_mode>.pbl)	64 KB	0x0000_0000 - 0x0000_FFFF	0x00008
Reserved	64 KB	0x0001_0000 - 0x0001_FFFF	0x00080
PFE firmware	256 KB	0x0002_0000 - 0x0005_FFFF	0x00100
FIP (BL31+BL32+BL33)	1 MB	0x0006_0000 - 0x000D_FFFF	0x00300
Environment variables	64 KB	0x001D_0000 - 0x001D_FFFF	0x00E80
Reserved	64 KB	0x001E_0000 - 0x001E_FFFF	0x00F00
Secure boot headers	64 KB	0x001F_0000 - 0x001F_FFFF	0x00F80

8 Prebuilt toolchains

Download prebuilt toolchain binaries

Prebuilt toolchains for supported targets are available on NXP official image mirror. The table below provides links from NXP official image mirror to download prebuilt toolchain binaries.

Table 10. Prebuilt toolchain download links

Target type	Toolchain download link
ARM32	https://www.nxp.com/lgfiles/sdk/lsdk2004-yocto31/fsl-qoriq-glibc-x86_64-fsl-toolchain-cortexa7hf-neon-toolchain-3.1.sh
ARM64	https://www.nxp.com/lgfiles/sdk/lsdk2004-yocto31/fsl-qoriq-glibc-x86_64-fsl-toolchain-aarch64-toolchain-3.1.sh
PPCE500V2	https://www.nxp.com/lgfiles/sdk/lsdk2004-yocto31/fsl-qoriq-glibc-x86_64-fsl-toolchain-ppce500v2-toolchain-3.1.sh
PPCE500MC	https://www.nxp.com/lgfiles/sdk/lsdk2004-yocto31/fsl-qoriq-glibc-x86_64-fsl-toolchain-ppce500mc-toolchain-3.1.sh
PPCE5500	https://www.nxp.com/lgfiles/sdk/lsdk2004-yocto31/fsl-qoriq-glibc-x86_64-fsl-toolchain-ppce5500-toolchain-3.1.sh
PPCE5500-64B	https://www.nxp.com/lgfiles/sdk/lsdk2004-yocto31/fsl-qoriq-glibc-x86_64-fsl-toolchain-ppc64e5500-toolchain-3.1.sh
PPCE6500	https://www.nxp.com/lgfiles/sdk/lsdk2004-yocto31/fsl-qoriq-glibc-x86_64-fsl-toolchain-ppce6500-toolchain-3.1.sh
PPCE6500-64B	https://www.nxp.com/lgfiles/sdk/lsdk2004-yocto31/fsl-qoriq-glibc-x86_64-fsl-toolchain-ppc64e6500-toolchain-3.1.sh

Install a prebuilt toolchain

You can install a prebuilt toolchain as follows:

```
$ ./fsl-qoriq-glibc-<host_arch>-fsl-toolchain-<core>-toolchain-<release>.sh
```

NOTE

The default installation path for a prebuilt toolchain is /opt/fsl-qoriq/<release>/. If desired, you can specify a different installation path during installation.

Use a prebuilt toolchain

To use a prebuilt toolchain after installation:

1. Go to the location where the toolchain is installed and source the environment-setup-<core> file. This sets up the correct path to the build tools and also exports some environment variables that are relevant for development (for example, `$CC`, `$ARCH`, `$CROSS_COMPILE`, `$LDFLAGS`, and so on).
2. Invoke the compiler using the `$CC` variable (for example, `$CC <source files>`).

NOTE

This is a sysrooted toolchain. GCC needs option `--sysroot=<path-to-target-sysroot>` to find target fragments and libraries (for example, `crt*`, `libgcc.a`) as the default sysroot is poisoned (made non-existent). However, when invoking the compiler through the `$CC` variable, you do not need to pass the `--sysroot` parameter as it is already included in the variable (you can verify it by running the `echo $CC` command).

9 User space applications

This section provides build and compile instructions related to these user space applications: Data Plane Development Kit (DPDK) and QEMU.

9.1 DPDK

9.1.1 Introduction

DPDK is a user space packet processing framework.

This guide contains instructions for installing and configuring the user space Data Plane Development Kit (DPDK) v19.11 software. Besides highlighting the applicable platforms, this guide describes steps for compiling and executing sample DPDK applications in a Linux application (*linuxapp*) environment over Layerscape boards.

OVS-DPDK is a software switching package that uses DPDK as the underlying platform. The guide also details methods to execute *ovs-dpdk* along with DPDK over Layerscape boards.

9.1.1.1 Supported platforms and platform-specific details

DPDK supports LS1012A, LS1028A, LS1043A, LS1046A, LS1088A, LS2088A, and LX2160 family of SoCs. This section details the architectural and port layout of their Reference Design Boards. Port layout information is especially relevant while executing DPDK applications - to map DPDK port number to physical ports.

Refer to the following for board-specific information:

9.1.1.1.1 LS1012A Reference Design Board (RDB)

LS1012A is a PPFE-based platform. For more information on LS1012ARDB, see www.nxp.com/LS1012ARDB

Hardware Specification of LS1012ARDB

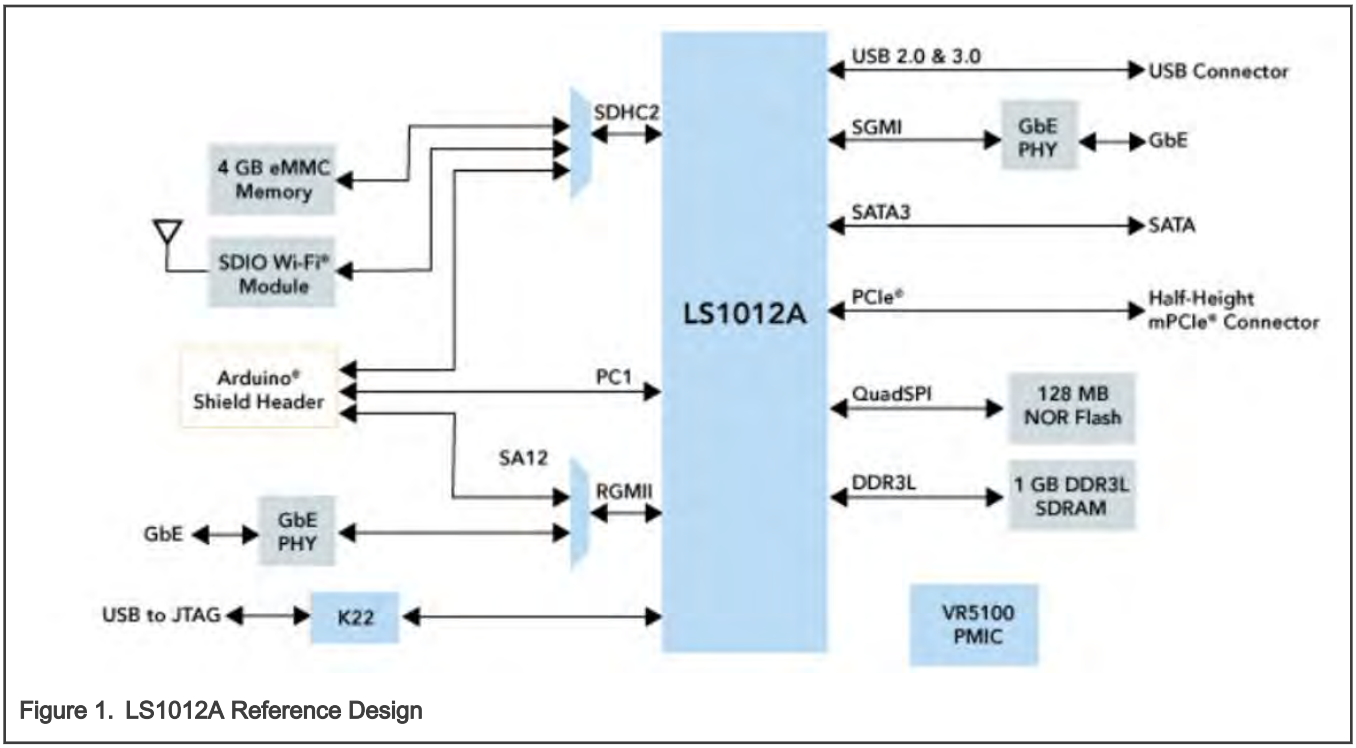


Figure 1. LS1012A Reference Design

LS1012ARDB Port Layout

Figure 2. LS1012A Port Layout

Label on Case	DPDK vdev Port Names
ETH1	eth_pfe0
ETH2	eth_pfe1

9.1.1.1.2 LS1028A Reference Design Board (RDB)

The LS1028A industrial applications processor includes a TSN-enabled Ethernet switch and Ethernet controllers to support converged IT and OT networks. For more information on LS1028ARDB, http://www.nxp.com/LS1028ARDB.

Hardware specification

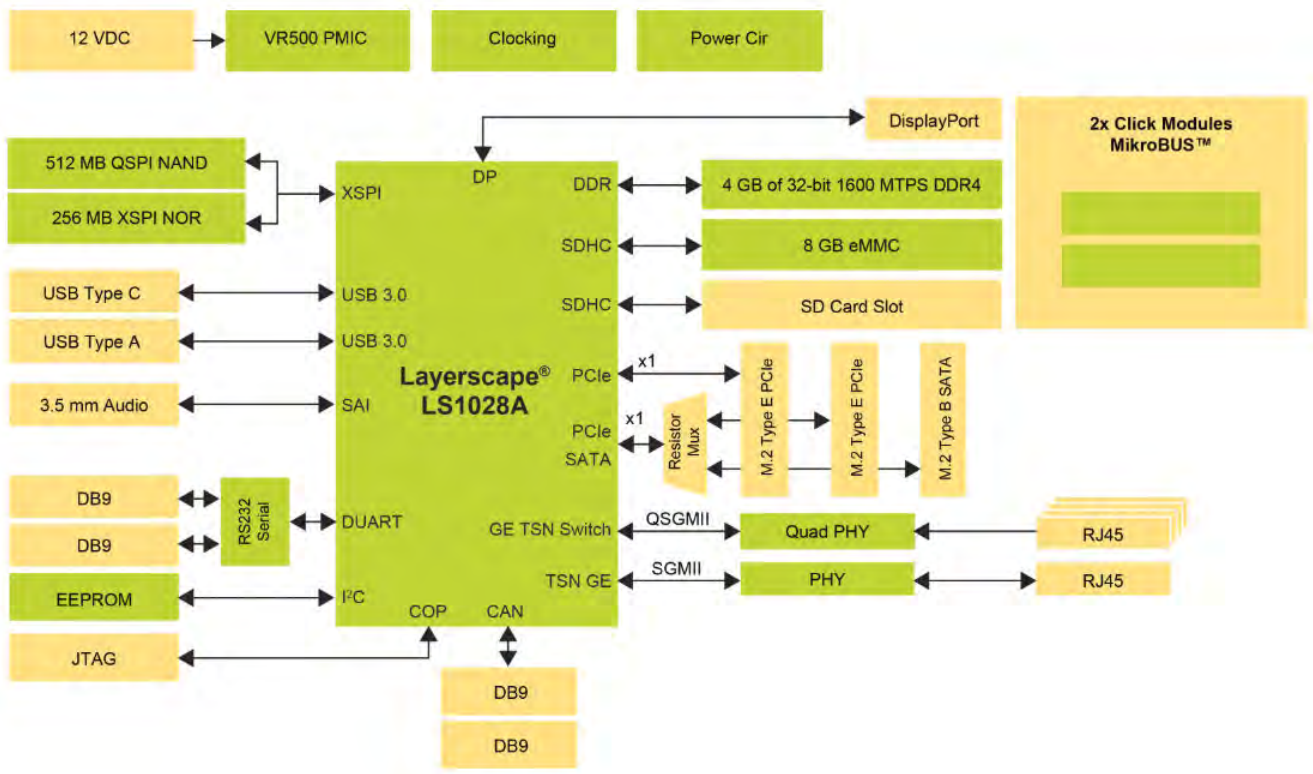


Figure 3. LS1028A architecture

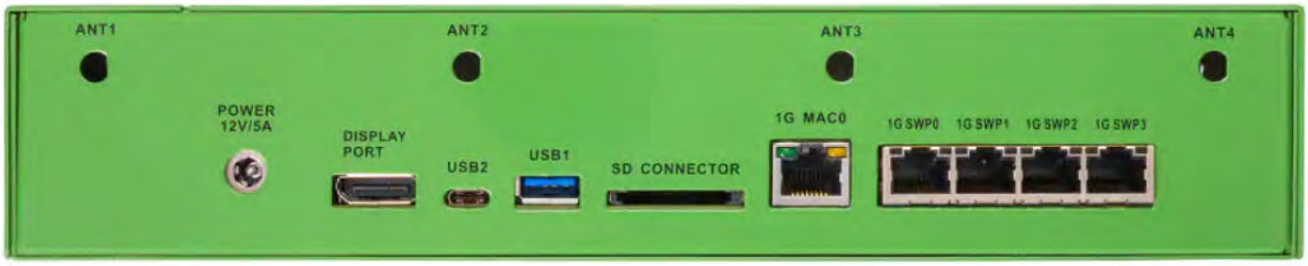


Figure 4. LS1028ARDB port layout

Label on Case	PCI address of interface
1G MAC0	0000:00:00.0
1G SWP0	NA
1G SWP1	NA
1G SWP2	NA
1G SWP3	NA

9.1.1.1.3 LS1043A Reference Design Board (RDB)

LS1043A is a DPAA-based platform. For more information on LS1043ARDB, see www.nxp.com/LS1043ARDB

Hardware Specification of LS1043ARDB

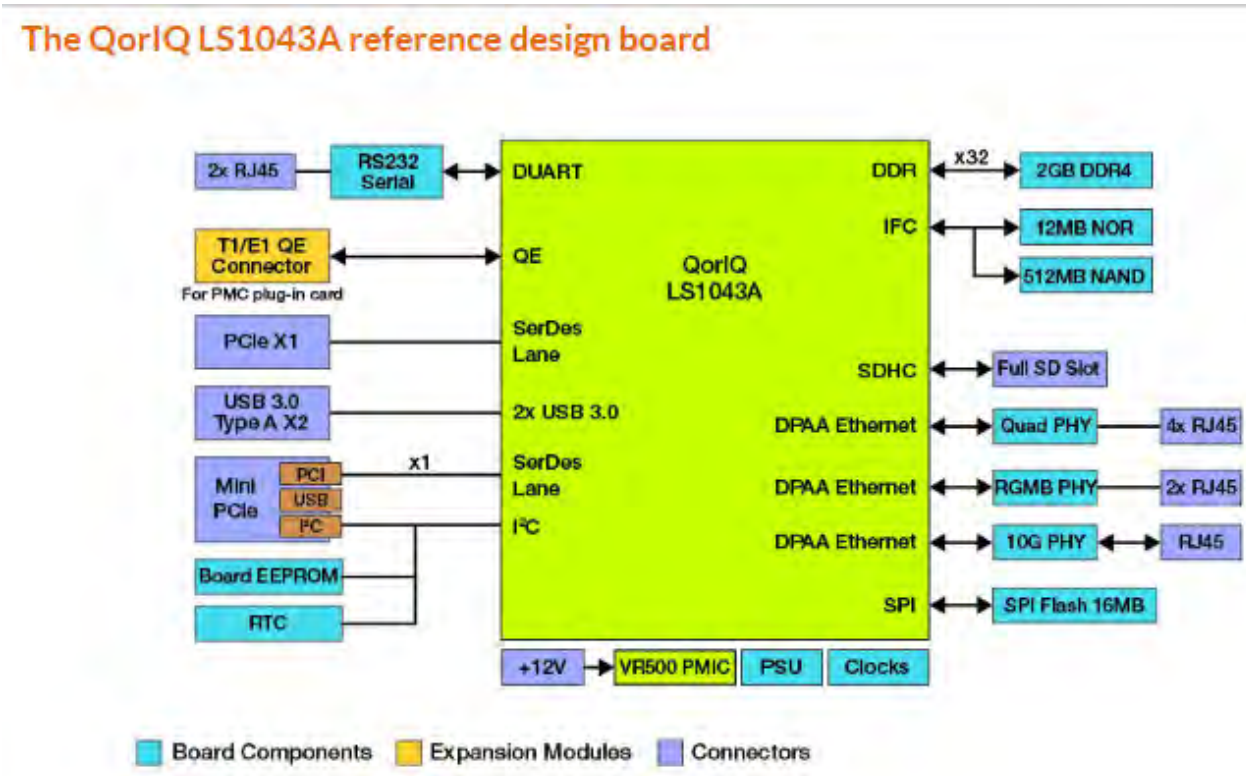


Figure 5. LS1043A Reference Design

LS1043ARDB Port Layout

Figure 6. LS1043A Port Layout

Label on Case	FMan Port Names	User space Ports	Comment
QSGMII.P0	FM0-MAC1	0	1G Port
QSGMII.P1	FM0-MAC2	1	1G Port

Table continues on the next page...

Table continued from the previous page...

RGMII1	FM0-MAC3	2	1G Port
RGMII2	FM0-MAC4	3	1G Port
QSGMII.P2	FM0-MAC5	4	1G Port
QSGMII.P3	FM0-MAC6	5	1G Port
10G	FM0-MAC9	6	10G - Copper Port

NOTE

Information provided in the "User space Ports" column above is conditional to default Device tree (DTB) provided as part of Board Support Package. The ordering can change for a custom DTB.

9.1.1.1.4 LS1046A Reference Design Board (RDB) / LS1046A Freeway Board (FRWY)

LS1046A is a DPAA-based platform. For more information on LS1046ARDB, see www.nxp.com/LS1046ARDB and for LS1046A Freeway, see www.nxp.com/FRWY-LS1046A.

Hardware specification of LS1046ARDB

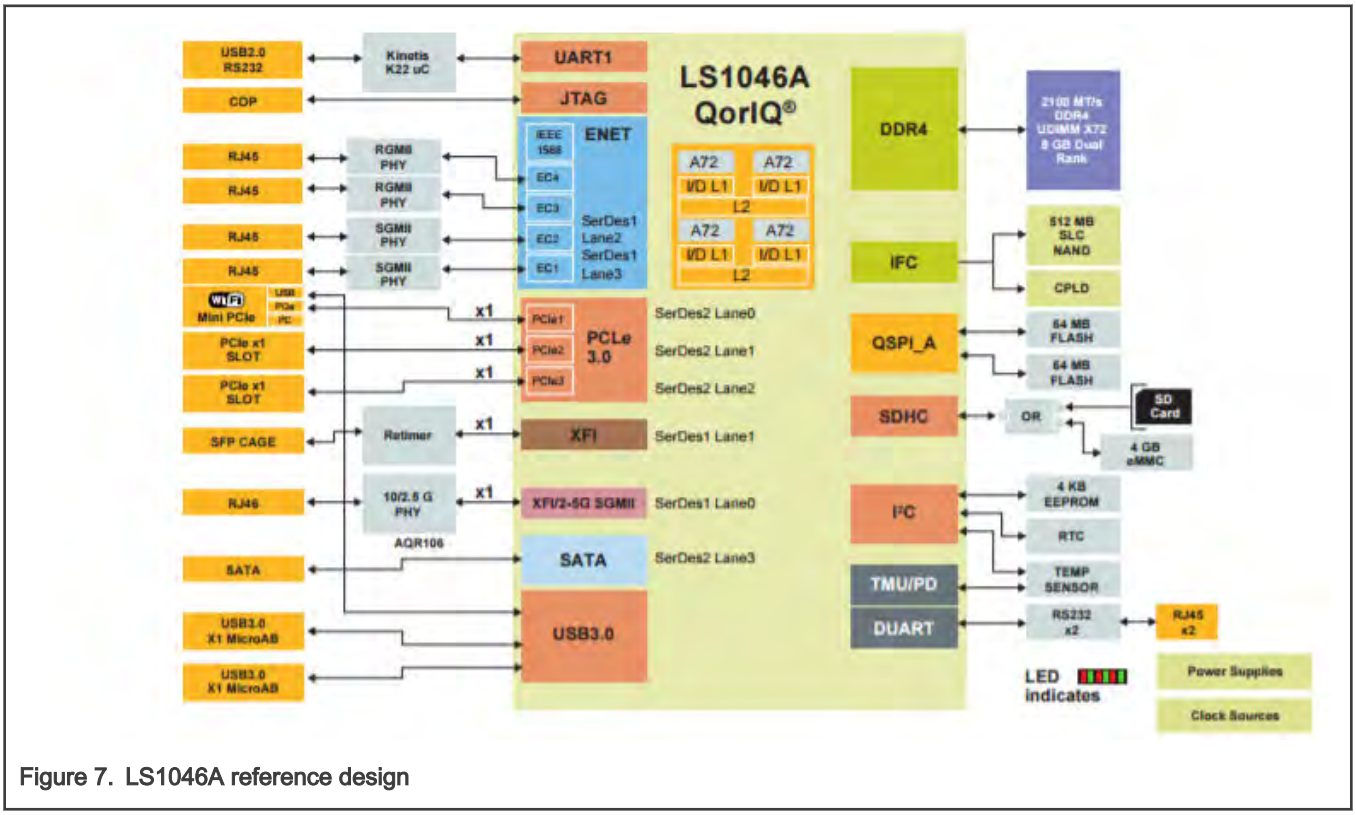


Figure 7. LS1046A reference design

LS1046ARDB port layout

Figure 8. LS1046ARDB port layout

Label on case	FMan port names	User space ports	Comment
RGMII1	FM0-MAC3	0	1G Port
RGMII2	FM0-MAC4	1	1G Port
SGMII1	FM0-MAC5	2	1G Port
SGMII2	FM0-MAC6	3	1G Port
10G-Copper	FM0-MAC9	4	10G – Copper Port
10G-SFP+	FM0-MAC10	5	10G – SFP+ Optical Port

NOTE

Information provided in the "User space Ports" column above is conditional to default Device tree (DTB) provided as part of LSDK. The ordering can change for a custom DTB.

FRWY-LS1046A port layout

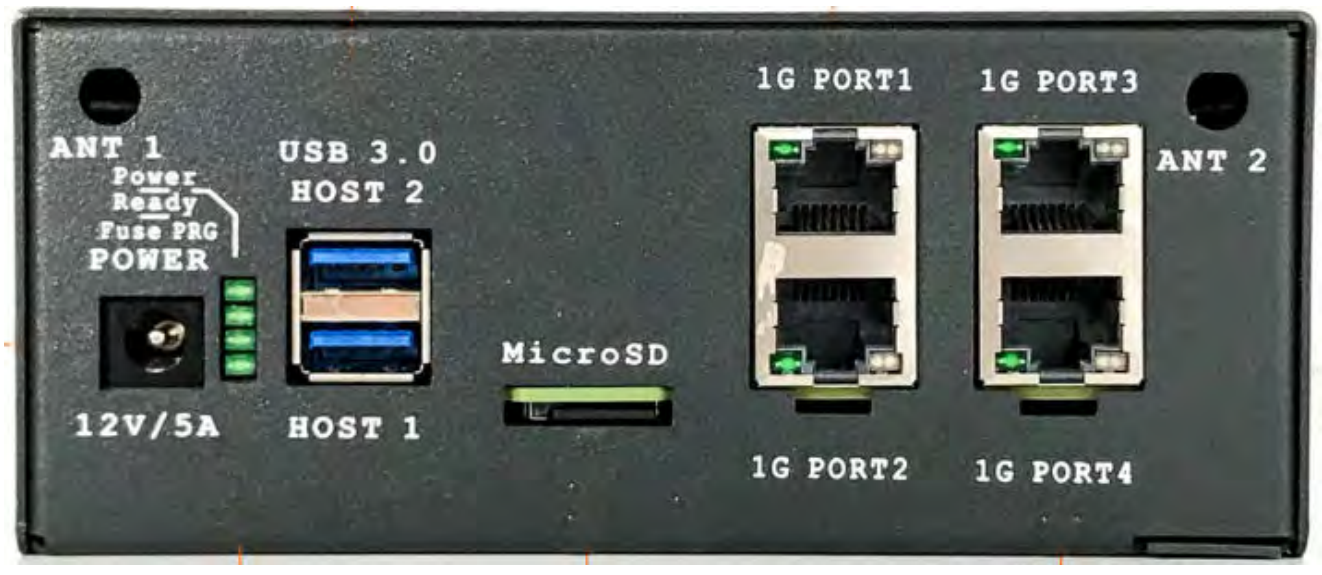


Figure 9. FRWY-LS1046A port layout

Label on case	FMan port names	User space ports	Comment
1G PORT1	FM0-MAC1	0	1G Port
1G PORT2	FM0-MAC5	1	1G Port
1G PORT3	FM0-MAC6	2	1G Port
1G PORT4	FM0-MAC10	3	1G Port

9.1.1.1.5 LS1088A Reference Design Board (RDB)

LS1088A is a DPAA2 based platform. For more information on LS1088A, see www.nxp.com/LS1088ARDB.

Hardware Specifications of LS1088ARDB

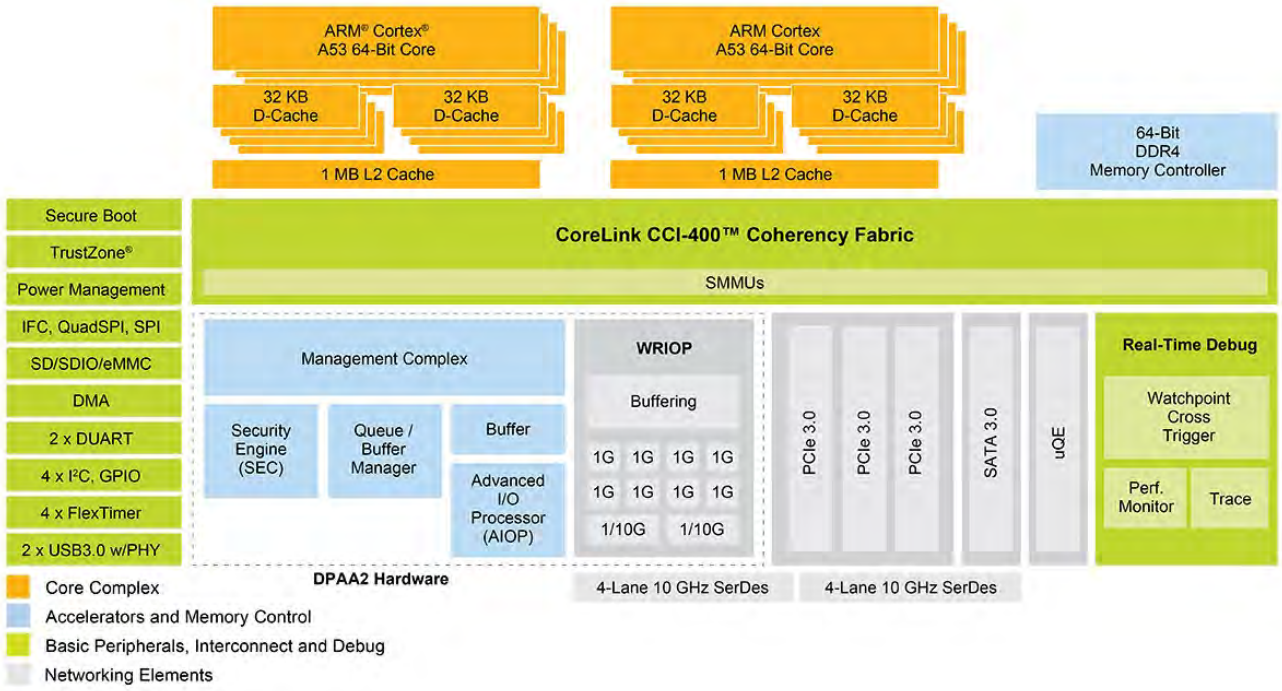


Figure 10. LS1088A Architecture

LS1088ARDB Port Layout

Figure 11. LS1088ARDB Port Layout

Label on Case	Physical Ports	Comment
ETH0	DPMAC.1	10G - Copper port

Table continues on the next page...

Table continued from the previous page...

ETH1	DPMAC.2	10G – SFP+ (Optical port)
ETH2	DPMAC.7	QSGMII port (1G)
ETH3	DPMAC.8	QSGMII port (1G)
ETH4	DPMAC.9	QSGMII port (1G)
ETH5	DPMAC.10	QSGMII port (1G)
ETH6	DPMAC.3	QSGMII port (1G)
ETH7	DPMAC.4	QSGMII port (1G)
ETH8	DPMAC.5	QSGMII port (1G)
ETH9	DPMAC.6	QSGMII port (1G)

9.1.1.1.6 LS2088A Reference Design Board (RDB)

LS2088A is a DPAA2 based platform. For more information on LS2088A, see www.nxp.com/LS2088ARDB.

Hardware specifications

LS2088A Reference Design Board

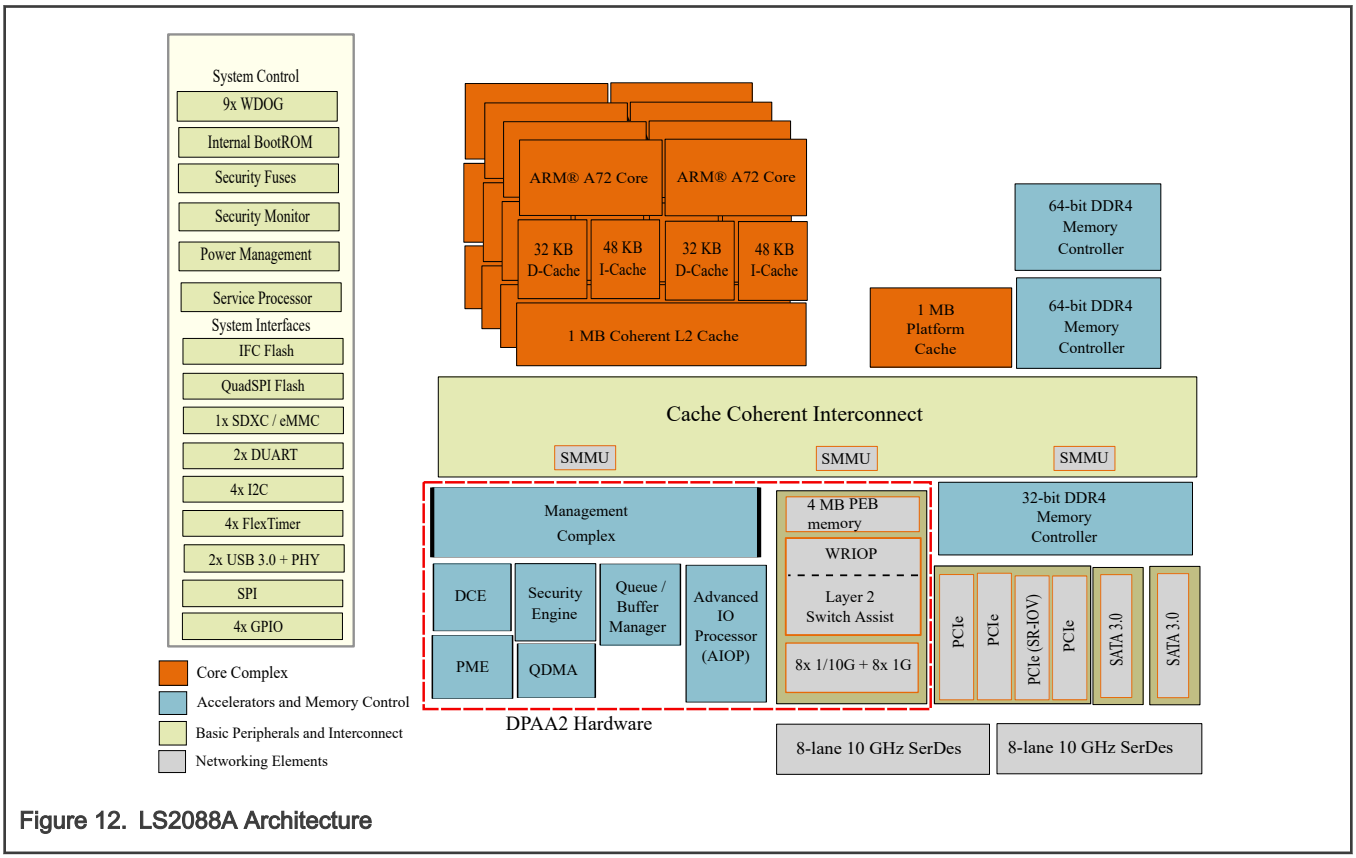


Figure 12. LS2088A Architecture

LS2088ARDB Port Layout

Figure 13. LS2088ARDB Port Layout

Label on Case	Physical Ports	Comment
ETH0	DPMAC.5	10G - Copper port
ETH1	DPMAC.6	10G - Copper port
ETH2	DPMAC.7	10G - Copper port
ETH3	DPMAC.8	10G - Copper port
ETH4	DPMAC.1	10G – SFP+ (Optical port)
ETH5	DPMAC.2	10G – SFP+ (Optical port)
ETH6	DPMAC.3	10G – SFP+ (Optical port)
ETH7	DPMAC.4	10G – SFP+ (Optical port)

9.1.1.1.7 LX2160A Reference Design Board (RDB)

LX2160A is a DPAA2 based platform. For more information on QorIQ LX2160A, see www.nxp.com/LX2160A.

Hardware specifications

LX2160A Reference Design Board

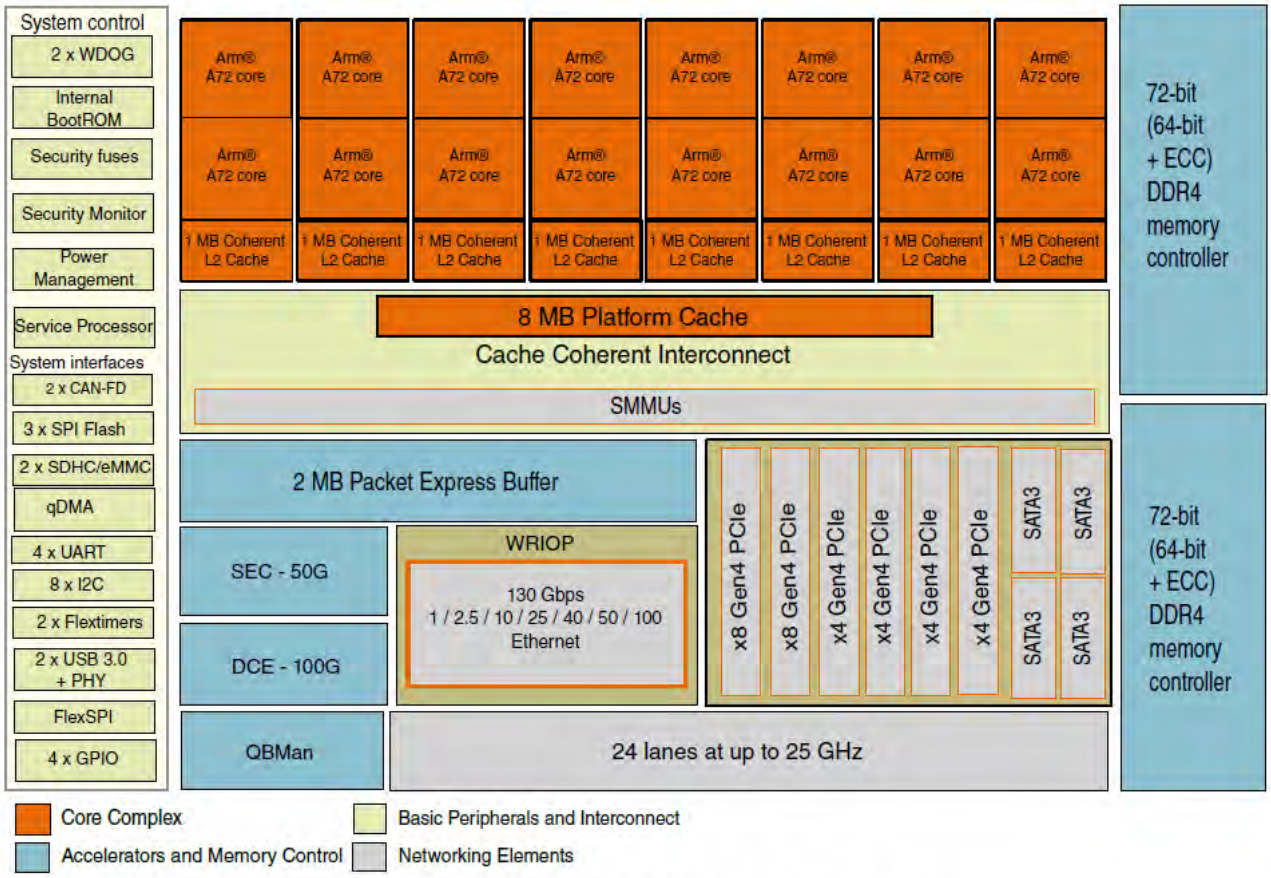


Figure 1-1. LX2160A block diagram

Figure 14. LX2160A Architecture

LX2160ARDB Port Layout

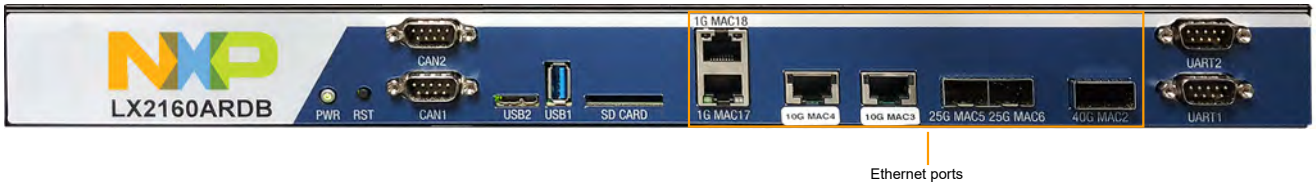


Figure 15. LX2160ARDB Port Layout

Table 11. Port Layout

Label on Case	Physical Ports	Comment
40G MAC2 (*)	dpmac.2	40G - Fiber port
10G MAC3	dpmac.3	10G - Copper port
10G MAC4	dpmac.4	10G - Copper port

Table continues on the next page...

Table 11. Port Layout (continued)

Label on Case	Physical Ports	Comment
25G MAC5	dpmac.5	25G - Fiber port
25G MAC6	dpmac.6	25G - Fiber port
10G MAC7 (*)	dpmac.7	10G - Fiber port
10G MAC8 (*)	dpmac.8	10G - Fiber port
10G MAC9 (*)	dpmac.9	10G - Fiber port
10G MAC10 (*)	dpmac.10	10G - Fiber port
1G MAC17	dpmac.17	1G - Copper port
1G MAC18	dpmac.18	1G - Copper port

NOTE

(*) Only one configuration between 40G or 4x10G would be available; therefore, depending on SerDes configuration, only one of {*dpmac.2*} port or {*dpmac.7, dpmac.8, dpmac.9, dpmac.10*} would be available. 4x10G is available by using port-splitter on the 40G port (*dpmac.2*). For 4x10G configuration, use SerDes Protocol 18.

SerDes configuration

Following table shows the SerDes protocol configuration application for LX2160A boards. Based on the configuration of the protocol, either 4x10G ports or 1x40G port is configured/visible.

18	USXGMII / XFI.3	USXGMII / XFI.4	25GE.5	25GE.6	USXGMII / XFI.7	USXGMII / XFI.8	USXGMII / XFI.9	USXGMII / XFI.10	SSFFSSSS
19	USXGMII / XFI.3	USXGMII / XFI.4	25GE.5	25GE.6	40GE.2				SSFFSSSS

NOTE

For detailed configurations and protocol information, see "LSDK Quick Start Guide for LX2160ARDB" section in "Layerscape SDK user guide" chapter of *Layerscape Software Development Kit User Guide* available at the following link:

https://www.nxp.com/design/software/embedded-software/linux-software-and-development-tools/layerscape-software-development-kit:LAYERSCAPE-SDK?tab=Documentation_Tab

9.1.1.2 References

Table 12. DPDK Application References

Sample Applications	DPDK Web Manual Link	Description
Layer-2 Forwarding (`l2fwd`)	l2fwd usage	Layer 2 Forwarding sample application setup and usage guide.
Layer-2 Forwarding with Crypto (`l2fwd-crypto`)	l2fwd-crypto	Layer 2 Forwarding with Crypto sample application setup and usage guide.
Layer-3 Forwarding (`l3fwd`)	l3fwd usage	Layer 3 Forwarding sample application setup and usage guide.

Table continues on the next page...

Table 12. DPDK Application References (continued)

IPsec Gateway (`ipsec-secgw`)	ipsec-secgw usage	IPsec Security Gateway sample application setup and usage guide.
PMD Test Application (`testpmd`)	testpmd usage	Guide for test application which can be used to test all PMD supported features.
DPDK Web Guide	DPDK Documentation	Link to DPDK Web Manual containing information about all supported PMD and Applications.

Table 13. Release References

Component	Base Upstream Release Versions
DPDK	19.11.0
OVS	2.13.0
PKTGEN	19.12.0

9.1.2 DPDK Overview

Key goal of the DPDK is to provide a simple, complete framework for fast packet processing in data plane applications. Using the APIs provided as part of the framework, applications can leverage the capabilities of underlying network infrastructure.

The framework creates a set of libraries for target environments, layered through an Environment Abstraction Layer (EAL) which hides all the device glue logic beneath a set of consistent APIs. These environments are created through the use of configuration files. Once the EAL library is created, the user may link with the library to create their own applications. Various other libraries, outside EAL, including the Hash, Longest Prefix Match (LPM) and rings libraries are also available for performing specific operations. Sample applications are also provided to help understand various features and uses of DPDK framework.

DPDK implements a run-to-completion model for packet processing where all resources must be allocated prior to calling data plane applications, running as execution units on logical processing cores. In addition, a pipeline model may also be used by passing packets or messages between cores via rings. This allows work to be performed in stages, resulting in more efficient use of code on cores.

More information on general working of DPDK can be found through DPDK website.

9.1.2.1 DPDK Platform Support

This section describes the NXP Data Path Acceleration Architecture, see the diagram below:

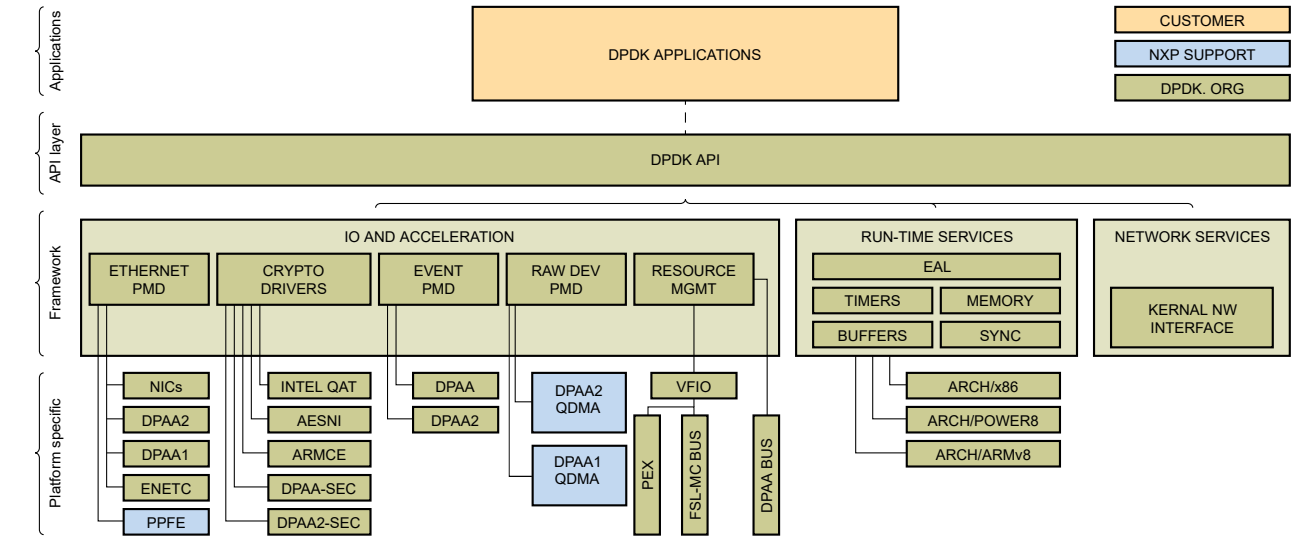


Figure 16. DPDK Architecture with NXP Components

The NXP Data Path Acceleration Architecture comprises of a set of hardware components, which are integrated via a hardware queue manager and use a common hardware buffer manager. Software accesses the DPAA via hardware components called *"Software Portals"*. These directly provide queue and buffer manager operations, such as enqueues, dequeues, buffer allocations, and buffer releases and indirectly provide access to all of the other DPAA hardware components via the queue manager.

NXP DPAA architecture-based *PMD (Poll Mode Drivers)* has been added to DPDK infrastructure to support seamless working on NXP platform. With the addition of these drivers, DPDK framework on NXP platforms permits Linux user space applications to be build using standard DPDK APIs in a portable fashion. The drivers directly access the DPAA queue and buffer manager software portals in a high performance manner and the internal details remains hidden from higher-level DPDK framework. Besides drivers for network interfaces, drivers (PMDs) for interfacing with Crypto (CAAM) block have also been included in the DPDK source code.

NOTE

Since this guide contains support for PPFE, DPAA2, ENETC, and DPAA platforms, the following markers are used throughout the guide:

- DPAA2 – This marker marks the steps/text applicable only for DPAA2 platforms, for example, LS2088
- DPAA – This marker marks the steps/text applicable only for DPAA platforms, for example, LS1043
- PPFE - This marker marks the steps/text applicable only for PPFE platforms, for example, LS1012
- ENETC - This marker marks the steps/text applicable only for ENETC platforms, for example, LS1028

All other steps which don’t have any marker are applicable for both the platforms.

NOTE

See DPDK Performance Reproducibility Guide to tune the system for best DPDK performance on NXP platforms.

NOTE

Multi-thread environment

DPDK was originally designed for Intel architectures, however efforts are underway to make it multiple architecture friendly. There are still some restrictions which should be taken care when used on NXP platforms.

1. Multiple pthreads

   DPDK usually pins one pthread per core to avoid the overhead of task switching. This allows for significant performance gains, but lacks flexibility and is not always efficient. DPDK is composed of several libraries - some of the functions in these libraries can be safely called from multiple threads simultaneously, while others cannot.

   The runtime environment of the DPDK is typically a single thread per logical core. It is best to avoid sharing data structures between threads and/or processes where possible. Where this is not possible, the execution blocks must access the data in a thread-safe manner. Mechanisms such as atomic variables or locking can be used to allow execution blocks to operate serially. However, this can affect the performance of the application.

2. Fast-path APIs

   Applications operating in the data plane are performance sensitive but certain functions within those libraries may not be safe to call from multiple threads simultaneously.

   The Hash, LPM, Mempool libraries, and RX/TX in the PMD are examples of such multi-thread unsafe functions. The RX/TX of the *PMD* are the most critical aspects of a DPDK application and it is recommended that no locking be used with these paths as it will impact performance. However, these functions can be safely used from multiple threads when each thread is performing I/O on a different NIC queue. If multiple threads are to use the same hardware queue on the same NIC port, then locking or some other form of mutual exclusion is necessary. In the NXP implementation, each thread has to use a software portal (DPIO) instance to access the underlying DPAA hardware. Thus, it is recommended that only one thread per logical core should be created for RX/TX and other I/O access to DPAA hardware.

9.1.2.2 DPAA: Supported DPDK Features

Following is the list of DPDK NIC features which DPAA driver supports:

- Allmulticast mode
- Basic and Extended stats
- Flow control
- Firmware Version information
- Jumbo frame
- L3 and L4 checksum offload
- Link status and update
- MTU update
- Promiscuous mode
- Queue start/stop
- Speed Capabilities
- Scattered RX and TX
- Unicast MAC filter
- RSS Hash
- Packet type parsing

- Interrupt mode
- Traffic splitting (VSP)
- Indirect and External buffers
- ARMv8

Following is the list of DPDK Crypto Device features which DPAA driver supports:

- Encryption/Descryption and Authentication
- Lookaside protocol offload support
- Multiple Algorithms as mentioned in `<dpdk>/doc/guides/cryptodevs/features/dpaa_sec.ini`
- PDCP protocol support
- IPSec RAW buffer support
- Scattered RX and TX

Other DPDK features supported on DPAA

- Event device support
- QDMA support

Applications validated on DPAA1

- dpdk-l2fwd
- dpdk-l3fwd
- dpdk-ipsec-secgw
- dpdk-testpmd
  - RX only, TX only, fwd modes
  - Jumbo Frame support
  - Scatter Gather support
  - Virtio net interfaces
  - Link bonding
  - RSS distributions
  - External buffer support
- dpdk-l2fwd-event
- dpdk-ethtool
- dpdk-ip_fragmentation
- dpdk-ip_reassembly
- dpdk-kni
- dpdk-l2fwd-qdma
- dpdk-l3fwd-power
- dpdk-link_status_interrupt
- dpdk-pdump
- dpdk-proc-info
- dpdk-test-crypto-perf

9.1.2.3 DPAA2: Supported DPDK Features

Following is the list of DPDK NIC features which DPAA2 driver supports:

- Allmulticast mode
- Basic stats
- Firmware Version information
- Flow control
- Jumbo frame
- L3 checksum offload
- L4 checksum offload
- Link Status
- Link Status Events
- MTU update
- Packet type parsing
- Promiscuous mode
- Queue start/stop
- RSS hash
- Unicast MAC filter
- VLAN filter and offload
- Speed capabilities
- Timesync
- Timestamp offload
- ARMv8
- Linux VFIO
- Extended stats

Following is the list of DPDK Crypto Device features which DPAA2 driver supports:

- Encryption/Decryption and Authentication
- Lookaside protocol offload support
- Multiple Algorithms as mentioned in `<dpdk>/doc/guides/cryptodevs/features/dpaa2_sec.ini`
- PDCP protocol support
- IPSec RAW buffer support
- Scattered RX and TX

Other DPDK features supported on DPAA2

- Event device support
- QDMA support
- Non-root user support

Applications validated on DPAA2

- dpdk-l2fwd
- dpdk-l3fwd

- dpdk-ipsec-secgw
- dpdk-testpmd
  - Rx only, TX only, fwd modes
  - Jumbo Frame support
  - Scatter Gather support
  - Flow Classification
  - Traffic Management
  - Virtio net interfaces
  - Link bonding
  - RSS distributions
  - External buffer support
- dpdk-l2fwd-event
- dpdk-ethtool
- dpdk-ip_fragmentation
- dpdk-ip_reassembly
- dpdk-kni
- dpdk-l2fwd-qdma
- dpdk-l3fwd-power
- dpdk-link_status_interrupt
- dpdk-mp_client
- dpdk-mp_server
- dpdk-ptpclient
- dpdk-qdma_demo
- dpdk-simple_mp
- dpdk-symmetric_mp
- dpdk-symmetric_mp_qdma
- dpdk-pdump
- dpdk-proc-info
- dpdk-test-crypto-perf

9.1.2.4 PPFE supported DPDK features

Following is the list of DPDK NIC features which PPFE driver supports:

- ALLmulticast mode
- Basic Stats
- MTU update
- Promiscuous mode
- Packet type parsing
- ARMv8

9.1.2.5 ENETC supported DPDK features

Following is the list of DPDK NIC features which ENETC driver supports:

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

Note:

ENETC-based DPDK features are not supported with Kernel 4.14.

9.1.3 Build DPDK

This section includes three subsections which detail:

1. Building DPDK binaries (libraries and sample applications) using the Yocto build system.
2. Building DPDK binaries as standalone package, through DPDK's own build system.
3. Building Pktgen application which can be used as a software packet generator using DPDK as underlying layer.

9.1.3.1 Build DPDK using Yocto

DPDK is one of the application packages of the Yocto build system. This section details method to build DPDK as a standalone package within the Yocto environment. It is assumed that the Yocto environment has already been configured before executing the commands below.

See Download Yocto layers for complete details of using the Yocto build system.

After the Yocto environment has been set up, the following commands can be used to build DPDK applications and libraries. Generated files (libraries and binaries) would be available in the `<yocto_sdk>/build_ls2088ardb/tmp/work/ls2088ardb-fsl-linux/dpdk/` folder. After the *rootfs* (root filesystem) is generated, the binaries would be merged into it.

```
bitbake dpdk         # it is assumed setup-env was run before running this command.
```

See Build Yocto images for packing these binaries into the target *rootfs* using the Yocto build system. Yocto environment by default compiles DPDK and place it in the rootfs when `bitbake fsl-image-networking` is run.

Layout of DPDK binaries

Single image of DPDK binary supports DPAA, DPAA2, ENETC, and PPFE platforms. Once the DPDK package has been installed, binaries would be available `/usr/share/dpdk` folder in the rootfs. Yocto system generates a single rootfs for all NXP platforms it supports.

```
/usr/share/dpdk/examples/        # Contains the sample applications listed in Table 12
```

At various places in this document, above binaries would be referred for representing execution as well as other information. It is assumed that execution is being done either using the `PATH` variable set, as explained above, or with absolute path to the binaries.

Besides the above folders, another set of files is also available in *rootfs* to support DPDK application execution. These files are available in the `/usr/share/dpdk` folder in the *rootfs*.

Table below depicts various DPDK artifacts which are available in the Yocto generated rootfs:

File/Image name relative to `/usr/share/dpdk/`	Description

`./examples/l2fwd`
`./examples/l3fwd`
`./examples/l2fwd-crypto`
`./examples/ipsec-secgw`
`./examples/testpmd`
	DPAA, DPAA2, ENETC, and PPFE
	DPDK Example applications and PMD test application.

`./dpaa/usdpaa_config_ls<PLAT>.xml`
`./dpaa/usdpaa_policy_hash_ipv4_1queue.xml`
`./dpaa/usdpaa_policy_hash_ipv4_2queue.xml`
`./dpaa/usdpaa_policy_hash_ipv4_4queue.xml`
	DPAA Only.
	FMC Configurations and Policy files.
	`<PLAT>` is platform name for DPAA platform, for example `ls1043` or `ls1046`.
	Each Policy file for defining the number of queues per port as mentioned in its name.

`./dpaa2/dynamic_dpl.sh` `./dpaa2/destroy_dynamic_dpl.sh`	DPAA2 Only. Dynamic DPL container creation and teardown script.
`../usertools/dpdk-setup.sh` `../usertools/dpdk_devbind.py`	PPFE only DPDK NIC binding utility. This is only applicable for executing DPDK applications in VM.
`./enable_performance_mode.sh` `./disable_performance_mode.sh`	When executing an Ubuntu OS over Layerscape board, performance on core 0 can become non-deterministic because of OS services and threads. These scripts allow a special setting wherein the DPDK application, which would run after running the enable script, would get real-time priorities. **NOTE** These scripts should not be used in general cases. For detailed use case, see Performance Reproducibility Guide.

`./examples/ipsec_secgw/ep0.cfg`
`./examples/ipsec_secgw/ep1.cfg`
`./ipsec/ep0_64X64.cfg`
`./ipsec/ep1_64X64.cfg`
	Configuration files for `ipsec-gw` example application.
	The `ep0` and `ep1` files are standard configurations for 2 tunnels for encryption and decryption, each. The `ep0_64X64` and

Table continues on the next page...

Table continued from the previous page...

`./ipsec/ep0_64X64_proto.cfg`
`./ipsec/ep0_64X64_sha256.cfg`
`./ipsec/ep1_64X64_proto.cfg`
`./ipsec/ep1_64X64_sha256.cfg`
	`ep1_64X64` are for 64 tunnels for encryption and decryption, each.

`/usr/bin/pktgen`	Packet generation application
`./debug_dump.sh`	Dumping the debug data for further analysis.

9.1.3.2 Build DPDK on host (native)

This section lists the steps required to build DPDK binaries (libraries and example applications) on the host environment. This environment is host enabled for building directly on the Layerscape target board.

NOTE

This section focuses on building of DPDK on a host machine for Layerscape boards as target. Notes are added to enable the compilation of DPDK applications directly on a host machine.

Setup proxies

Depending on the environment you are working in, proxy settings might require to have Internet connectivity. Use the following proxy commands:

```
$ export http_proxy=http://<proxy-server-name>.com:<port-number>
$ export https_proxy=https://<proxy-server-name>.com:<port-number>
```

Obtain the DPDK source code

The DPDK source code contains all the libraries for building example applications as well as test applications. The source code includes configurations and scripts for supporting build and execution. Obtain the DPDK source code using the link below:

```
git clone https://source.codeaurora.org/external/qoriq/qoriq-components/dpdk -b github.qoriq-os/19.11-qoriq
```

After the above repository is cloned, DPDK source code is available for compilation. This source is common for DPAA, DPAA2, ENETC, and PPFE platforms.

Compiling DPDK

Follow the below steps to compile DPDK. For direct compilation on target boards, it is assumed that prerequisites would be satisfied using the root filesystem. Execute the following command:

```
make T=arm64-dpaa-linuxapp-gcc install CONFIG_RTE_KNI_KMOD=n CONFIG_RTE_EAL_IGB_UIO=n -j 4
```

A directory named `arm64-dpaa-linuxapp-gcc` is created; binaries and libraries are also available in it.

The KNI and other kernel module compilation should be disabled as only limitation is kernel module like (KNI) native compilation is not supported due to build dependencies not met by Root File System.

NOTE

DPDK `arm64-dpaa-linuxapp-gcc` folder contains *.config* file for storing the build configuration. Another way of disabling or enabling support features, like KNI and software crypto drivers, is to edit this file before executing the `make` command. If this method is adopted, parameters to command line for disabling the feature are not required. There is one limitation that the kernel module (KNI) compilation is not supported because build dependencies at native are not satisfactory.

NOTE

If KNI or software crypto driver support is disabled using the `make` command line parameters, it would not modify the configuration file for DPDK in the `<target>` folder. Every subsequent compilation of DPDK or example application must include the same command line arguments to avoid failure because of missing features which were not compiled. Or, edit the *.config* folder in the `arm64-dpaa-linuxapp-gcc` build folder.

Compiling DPDK example applications

Once the DPDK source code is compiled, the DPDK example applications can be built independently as required:

1. Before the example applications can be built, the path to DPDK SDK must be set which includes the DPDK source code. The build system uses this path to look for compiled libraries and headers.

   ```
   export RTE_SDK=<path to DPDK source code, where compilation was done>
   ```

2. Target should be set to same value as done for compilation of DPDK.

   ```
   export RTE_TARGET=arm64-dpaa-linuxapp-gcc
   ```

3. Once the above variables are set, example applications can be compiled using the following commands:

   Some applications, such as `testpmd` are generated as part of default build. These applications are available in `<build folder>/app/` folder.

For other example applications which are part of the examples/ folder, one of following is applicable:

```
make -C examples/l3fwd             # for the L3 forwarding application
make -C examples/l2fwd             # for the L2 forwarding application
make -C examples/ip_fragmentation        # for the IP fragmentation application
make -C examples/ip_reassembly           # for the IP reassembly application
make -C examples/ipsec-secgw          # for the IPsec gateway application
```

Above are sample commands for a limited set of DPDK example applications. Other applications can be compiled using similar command pattern.

```
make -C examples/<Name of examples directory>
```

NOTE

All the example applications currently supported by DPDK are available as part of the DPDK source code in the `./examples/` folder. Other examples can also be compiled using the pattern stated above.

9.1.3.3 Standalone build of DPDK libraries and applications

This section details steps required to build DPDK binaries (libraries and example applications) in a standalone environment. This environment can either be on a host enabled for cross building for Layerscape boards or directly on the Layerscape target board.

NOTE

This section primarily focuses on standalone building of DPDK on a host machine using cross compilation for Layerscape boards as target. Though, necessary notes have been added to enable compilation directly on target boards. See Download Yocto layers for creating an environment suitable for building DPDK on Layerscape boards.

For instructions on how to build DPDK using Yocto system, see Build DPDK using Yocto.

Obtain the DPDK source code

The DPDK source code contains all the necessary libraries for build example applications as well as test applications. The source code also includes various configuration and scripts for supporting build and execution. Obtain the DPDK source code using the link below:

```
git clone https://source.codeaurora.org/external/qoriq/qoriq-components/dpdk -b github.qoriq-os/19.11-qoriq
```

After the above repository has been cloned, DPDK source code is available for compilation. This source is common for both, DPAA, DPAA2, ENETC, and PPFE platforms.

Prerequisites before compiling DPDK

Before compiling DPDK as a standalone build, following dependencies need to be resolved independently:

- Platform compliant and compiled Linux Kernel source code so that KNI modules can be built.
  - This is optional and if KNI module support is not required, this can be ignored
  - For details of compiling platform-compliant Linux kernel, see Download Yocto layers and Build Yocto images
  - For disabling KNI module, see notes below
- OpenSSL libraries required for building software crypto driver (OpenSSL PMD).
  - OpenSSL package needs to be separately compiled and libraries installed at a known path before DPDK build can be done
  - This is optional and if software crypto driver support is not required, this dependency can be ignored.

    Follow the steps below to build OpenSSL as a standalone package.

    ```
    git clone git://git.openssl.org/openssl.git  # Clone the OpenSSL source code
    cd openssl                                   # Change into cloned directory
    git checkout OpenSSL_1_1_1g                  # Checkout the specific tag supported by DPDK
    ```

    Export the Cross Compilation tool chain for building OpenSSL for target. The following step for exporting cross compilation toolchain is required only when compiling on Host. On a target board, it is assumed default build toolchain would be used.

    ```
    export CROSS_COMPILE=<path to uncompressed toolchain archive>/bin/aarch64-linux-gnu-
    ```

    Configure the OpenSSL build system with following command. The `--prefix` argument specifies a path where OpenSSL libraries would be deployed after build completes. This is also a path which would be provided to DPDK build system for accessing the compiled OpenSSL libraries.

    ```
    ./Configure linux-aarch64  --prefix=<OpenSSL library path> shared
    ```

    ```
    make depend
    make
    make install
    export OPENSSL_PATH=<OpenSSL library path>
    ```

NOTE

When building DPDK on target board, it is possible that OpenSSL libraries required by DPDK are already available as part of the *rootfs*, in which case external compilation of OpenSSL package would not be required.

— For disabling OpenSSL PMD support, see notes below

Compiling DPDK

Follow the below steps to compile DPDK once the above prerequisites are resolved. These steps are common for DPAA and DPAA2 targets and are needed only when cross compiling on a host for Layerscape boards as target. In case of direct compilation on target boards, it is assumed that prerequisites would be satisfied using the root filesystem. In case root filesystem doesn't contain necessary prerequisites, below steps would be required once prerequisites have been built/obtained independently.

1. Setup the environment for compilation:
   a. Setup Linux Kernel path. This is optional and required only for KNI and ixgb_uio module compilation. Skip it, if ixgb_uio or KNI module or KNI example application is not required.

```
export RTE_KERNELDIR=<Path to compiled Linux kernel to compile KNI kernel module>
```

   b. Setup cross compilation toolchain.

      This step is required only on the host environment where default toolchain is not for target boards. When compiling on a target board, this step can be skipped.

```
export CROSS=<path to cross-compile toolchain>
```

   c. Setup OpenSSL path for software crypto drivers (OpenSSL PMD). This is optional and can be skipped in case software crypto driver (OpenSSL PMD) support is not required.

```
export OPENSSL_PATH=<path to installed OpenSSL>
```

2. Use DPDK build system for compiling DPDK.

NOTE

DPDK binaries generated using below steps are compatible for DPAA, DPAA2, ENETC, and PPFE platforms. This is also valid when DPDK is build through Yocto build system.

   a. Execute the following command:

```
make T=arm64-dpaa-linuxapp-gcc install DESTDIR=<location to install DPDK>
```

      Where `DESTDIR=<location to install DPDK>` is an optional parameter to deploy all the DPDK binaries (libraries and example applications) to a standard Linux package specific layout within a directory represented by this parameter. Alternatively, a directory named `arm64-dpaa-linuxapp-gcc` is also created and binaries and libraries are also available in it.

   b. **Disabling KNI and other kernel module compilation**: In case DPDK kernel module is not required (`RTE_KERNELDIR` variable is not set), use the following command. `DESTDIR` can be added, as explained above, if required.

```
make T=arm64-dpaa-linuxapp-gcc CONFIG_RTE_KNI_KMOD=n CONFIG_RTE_EAL_IGB_UIO=n
CONFIG_RTE_EAL_IGB_UIO=n install
```

   c. **Enabling software crypto driver support**: Software crypto driver (OpenSSL PMD) is disabled by default. If it is required set `OPENSSL_PATH` variable, use the following command. `DESTDIR` can be added, as explained above, if required.

```
make T=arm64-dpaa-linuxapp-gcc CONFIG_RTE_LIBRTE_PMD_OPENSSL=y EXTRA_CFLAGS="-I$
{OPENSSL_PATH}/include/" EXTRA_LDFLAGS="-L${OPENSSL_PATH}/lib/" install
```

d. In case KNI is not required and software crypto support is required, use the following command. `DESTDIR` can be added, as explained above, if required.

```
make T=arm64-dpaa-linuxapp-gcc CONFIG_RTE_LIBRTE_PMD_OPENSSL=y EXTRA_CFLAGS="-I$
{OPENSSL_PATH}/include/" EXTRA_LDFLAGS="-L${OPENSSL_PATH}/lib/" CONFIG_RTE_KNI_KMOD=n
CONFIG_RTE_EAL_IGB_UIO=n install
```

e. In case of `dpdk-pdump`, an example of multiprocess application, following command pattern can be used after replacing the `EXTRA_*` variables with appropriate path.

```
make T=arm64-dpaa-linuxapp-gcc CONFIG_RTE_LIBRTE_PMD_PCAP=y CONFIG_RTE_LIBRTE_PDUMP=y
EXTRA_LDFLAGS="-L/path/to/compiled/LIBPCAP/lib" EXTRA_CFLAGS="-I"-L/path/to/compiled/
LIBPCAP/include" CONFIG_RTE_KNI_KMOD=n CONFIG_RTE_EAL_IGB_UIO=n install
```

NOTE

The LIBPCAP library and headers provided to above build command should have been cross-compiled for `aarch64` and should be copied over to the board before executing the binary.

NOTE

Currently, `dpdk-pdump` is supported only with `testpmd` application compiled using same build steps. Other applications would require modification for supporting packet capturing support before being run with `dpdk-pdump`.

NOTE

For more information about the DPDK build system, see DPDK Documentation.

NOTE

DPDK `arm64-dpaa-linuxapp-gcc` folder contains `.config` file for storing the build configuration. Another way of disabling or enabling support features, like KNI and software crypto drivers, is to edit this file before executing the `make` command. If this method is adopted, parameters to command line for disabling the feature are not required.

NOTE

If KNI or software crypto driver support is disabled using the `make` command line parameters, it would *not* modify the configuration file for DPDK in the `<target>` folder. Every subsequent compilation of DPDK or example application would need to include the same command line arguments to avoid failure because of missing features which were not compiled. Or, edit the `.config` folder in the `arm64-dpaa-linuxapp-gcc` build folder.

Compiling DPDK example applications

Once the DPDK source code has been compiled, the DPDK example applications can be built independently as required.

1. Before the example applications can be built, the path to DPDK SDK needs to be set which includes the DPDK source code. This would be used by build system to look for compiled libraries and headers.

```
export RTE_SDK=<path to DPDK source code, where compilation was done>
```

2. Target should be set to same value as done for compilation of DPDK.

```
export RTE_TARGET=arm64-dpaa-linuxapp-gcc
```

3. Once the above variables are set, example applications can be compiled using the following commands:

   Some applications, such as `testpmd`, `dpdk-procinfo`, and `dpdk-pdump` (last two are multiprocess examples) are generated as part of default build. These would be available in `<build folder>/app/` folder. Steps for these applications are defined in the Compiling DPDK section above.

For other example applications which are part of the `examples/` folder, one of following is applicable:

```
make -C examples/l3fwd            # for the L3 forwarding application
```

```
make -C examples/l2fwd            # for the L2 forwarding application
```

```
make -C examples/ip_fragmentation # for the IP fragmentation application
```

```
make -C examples/ip_reassembly    # for the IP reassembly application
```

```
make -C examples/ipsec-secgw      # for the IPsec gateway application

make -C examples/ipsec-secgw CONFIG_RTE_LIBRTE_PMD_OPENSSL=y EXTRA_CFLAGS="-I${OPENSSL_PATH}/
include/" EXTRA_LDFLAGS="-L${OPENSSL_PATH}/lib/" # for IPsec application with openssl PMD
```

```
make -C examples/l2fwd-crypto     # for the L2 forwarding with crypto support application

make -C examples/l2fwd-crypto CONFIG_RTE_LIBRTE_PMD_OPENSSL=y EXTRA_CFLAGS="-I${OPENSSL_PATH}/
include/" EXTRA_LDFLAGS="-L${OPENSSL_PATH}/lib/"  # for L2 forwarding crypto operations with
openssl PMD
```

Above are sample commands for a limited set of DPDK example applications. Other applications too be compiled using similar command pattern.

```
make -C examples/<Name of examples directory>
```

NOTE

All the example applications currently supported by DPDK are available as part of the DPDK source code in the `./examples/` folder. Other examples can also be compiled using the pattern stated above.

NOTE

`testpmd` is not supported on platforms with single core, for example LS1012 (PPFE). This is because `testpmd` requires one core for its CLI or management (timer) threads.

4. Once the example application are compiled, the binaries would be available in the following folder within the DPDK source code folder:

```
examples/<name of example application>/build/app/*
```

Besides the above example application, DPDK also provides a `testpmd` binary which can be used for comprehensive verification of DPDK driver (PMD) features for available and compatible devices. This binary is compiled by default during DPDK source compilation explained in Compiling DPDK section.

NOTE

Only a small set of DPDK example applications is currently deployed to root filesystem when compiling DPDK through Yocto build system. These are `l2fwd`, `l3fwd`, `l2fwd-crypto`, `ipsec-gw`, `cmdif_demo`, `ip_fragmentation`, `ip_reassembly`, `l2fwd-qdma`, and `testpmd`.

9.1.3.4 DPDK-based Packet Generator

Pktgen is a packet generator powered by DPDK. It requires DPDK environment for compilation and DPDK-compliant infrastructure for execution. Pktgen can use DPAA and DPAA2 DPDK PMD (Poll Mode Drivers) for building a packet generator using the DPAA infrastructure.

Prerequisites for compiling Pktgen

For compiling Pktgen, **libpcap** library is required. If compiling **Pktgen** as a cross-compiled target, then compile `libpcap` also against the same compiler. See Build Yocto images for more information.

NOTE

For libpcap library compilation and deployment, refer to Tcpdump and libpcap project pages. Libpcap current and past releases can be obtained from this link. Documentation for libpcap is included in its source code. Also note that libpcap should be compiled for target board if working in a cross compilation environment.

Obtaining the Pktgen source code

Fetch the **Pktgen** source code using the following clone command:

```
git clone http://dpdk.org/git/apps/pktgen-dpdk
git checkout pktgen-19.12.0
```

Compiling Pktgen

Compilation steps below assume that compiled DPDK binaries (libraries and headers) are available in build directory generated by DPDK. See DPDK Build Steps for compiling DPDK and creating the build (`arm64-dpaa-linuxapp-gcc`) directory. Further, it is expected that `libpcap` libraries and headers are also present in this build folder.

Export the path to DPDK build environment and build folder defined by the compilation target:

```
export RTE_SDK=<path to compiled DPDK source code containing build folder>
```

```
export RTE_TARGET=<arm64-dpaa-linuxapp-gcc or arm64-dpaa-linuxapp-gcc>  # Select the build folder
based on required DPAA or DPAA2 target>
```

Build the source code:

```
make
```

Before executing the Pktgen application

For executing the Pktgen application, *Pktgen.lua* file and *pktgen* binary are needed on the execution environment.

If build was done using a cross compiled environment, transfer these binaries to the target environment from the build host. If the compilation was done on the target board, skip this step.

```
cd <Pktgen compiled source code>
cp Pktgen.lua <target board>
cp app/app/arm64-dpaa-linuxapp-gcc/pktgen <target board>
```

9.1.3.5 Build OVS-DPDK using Yocto

OVS is a multilayer virtual switch for enabling massive network automation through programmatic extensions.

OVS-DPDK is one of the application packages of the Yocto build system that uses DPDK as underlying framework. This section explains the method to build OVS-DPDK as a standalone package within the Yocto environment. It is assumed that the Yocto environment is configured before executing the commands mentioned below.

See Download Yocto layers for details about using the Yocto build system.

NOTE

In the Yocto configurations, OVS-DPDK must be configured to '`y`' for enabling packaging of OVS-DPDK in Yocto generated root filesystem, if not already enabled. For more information, see Download Yocto layers.

After the Yocto environment has been set up, the following commands can be used to build OVS-DPDK package. Generated files (libraries and binaries) would be available in the `<yocto_sdk>/build_ls2088ardb/tmp/deploy/images/ls2088ardb/` folder. After the *rootfs* (root filesystem) is generated, the binaries would be merged into it.

```
$ bitbake ovs-dpdk
```

NOTE

OVS-DPDK is dependent on DPDK package as it is used as its underlying framework. Yocto is designed to compile DPDK before OVS-DPDK if not already built.

Layout of OVS-DPDK binaries

An OVS-DPDK binary image supports both the DPAA and DPAA2 platforms. After the OVS-DPDK package has been installed, binaries are available in `/usr/bin/ovs-dpdk` folder in the rootfs. Yocto system generates a single rootfs for all NXP platforms it supports.

NOTE

OVS-DPDK binaries are deployed into the root filesystem as per the default layout of installation target for OVS-DPDK build system.

The table below depicts various OVS-DPDK artifacts that are available in the Yocto generated rootfs.

File/image name related to `/usr/bin/ovs-dpdk`	Description
• `./ovs-ofctl` • `./ovs-vsctl` • `./ovsdb-tool` • `./ovsdb-server` • `./ovs-vswitchd` • And various other binaries installed by OVS package by default	Various OVS binaries for both DPAA and DPAA2 platforms

9.1.3.6 Virtual machine (VM or guest) images

This section describes steps for deploying a Virtual Machine and executing DPDK applications in it. Additionally, OVS-DPDK package is used for deploying a software switch on the host machine through which virtual machines communicate with other virtual machine or external network.

NOTE

For obtaining necessary artifacts (kernel image, rootfs) for booting up a virtual machine on a Layerscape board, see QEMU.

9.1.4 Executing DPDK Applications on Host

This section describes how to execute DPDK and related applications in both Host and VM environments.

NOTE

`IP_ADDR_BRD`, `IP_ADDR_IMAGE_SERVER`, and `TFTP_BASE_DIR` are not U-Boot or Linux environment variables. They are used in this document to represent:

1. **IP_ADDR_BRD**: IP address of target board in test setup.
2. **IP_ADDR_IMAGE_SERVER**: IP address of the machine where all the software images are kept. These images are transferred to the board using either tftp or scp.
3. **TFTP_BASE_DIR**: TFTP base directory of TFTP server running on the machine where images are kept.

9.1.4.1 Booting up target board

NOTE

To get the target board up and working, see "LSDK Quick Start Guide" section for the board in "Layerscape SDK user guide" chapter of *Layerscape Software Development Kit User Guide* available at the following link:

https://www.nxp.com/design/software/embedded-software/linux-software-and-development-tools/layerscape-software-development-kit:LAYERSCAPE-SDK?tab=Documentation_Tab

While bringing up various platforms, use the following boot arguments to obtain best performance. This can be done by appending the following string to the `othbootargs` environment variable in U-Boot. If `othbootargs` is not present, then create a new variable. While booting up, the boot scripts would append `othbootargs` to the `bootargs` variable.

```
For LS1012ARDB:
default_hugepagesz=2MB hugepagesz=2MB hugepages=256 iommu.passthrough=1

For LS1028ARB:
default_hugepagesz=2MB hugepagesz=2MB hugepages=256 isolcpus=1 iommu.passthrough=1

For LS1043ARDB:
default_hugepagesz=2MB hugepagesz=2MB hugepages=512 isolcpus=1-3 bportals=s0 qportals=s0
iommu.passthrough=1

For LS1046ARDB:
default_hugepagesz=2m hugepagesz=2m hugepages=448 isolcpus=1-3 bportals=s0 qportals=s0
iommu.passthrough=1

For LS1088ARDB:
default_hugepagesz=1024m hugepagesz=1024m hugepages=6 isolcpus=1-7 iommu.passthrough=1

For LS2088ARDB:
default_hugepagesz=1024m hugepagesz=1024m hugepages=8 isolcpus=1-7 iommu.passthrough=1

For LX2160ARDB:
default_hugepagesz=1024m hugepagesz=1024m hugepages=8 isolcpus=1-15 iommu.passthrough=1
```

The above settings ensure that the total number of available huge pages are available with the application depending on the platform. `isolcpus` ensures that Linux kernel does not use these CPUs for scheduling its tasks — this prevents context switching of any application running on these cores. If the installed memory is lesser, lower number of huge pages can be used.

`iommu.passthrough=1` makes Linux kernel to disable SMMU configuration which is ignored in case of DPDK user space application. Though, this setting impacts security context of environment and should be done after due diligence.

The `bportals` and `qportals` settings ensure that only 1 portal is available for kernel use (because only one core is for kernel), rest are available for user space. These settings are only needed for DPAA1 platforms.

NOTE

Depending on the memory available, huge pages can also be added to the system from command prompt:

```
echo 256 > /proc/sys/vm/nr_hugepages
Check it with:
cat /proc/meminfo
```

NOTE

To update boot arguments for UEFI, see section 5.4 ("UEFI") of *Layerscape Software Development Kit User Guide* available at the following link:

https://www.nxp.com/design/software/embedded-software/linux-software-and-development-tools/layerscape-software-development-kit:LAYERSCAPE-SDK?tab=Documentation_Tab

Update grub.cfg file for huge page and isolcpus related changes as follows:

- On DPAA2 platforms:

```
rootwait=20 default_hugepagesz=1024m hugepagesz=1024m hugepages=8 isolcpus=1-7
```

- On DPAA1 platforms:

```
rootwait=20 default_hugepagesz=2MB hugepagesz=2MB hugepages=512 isolcpus=1-3
bportals=s0 qportals=s0
```

User space mode for DPAA1: For the DPAA platform, DPDK-specific Device Tree file (for example, `fsl-ls1046a-rdb-usdpaa.dtb` for LS1046ARDB, `fsl-ls1046a-frwy-usdpaa.dtb` for FRWY-LS1046A, and `fsl-ls1043a-rdb-usdpaa.dtb` for LS1043ARDB) should be used for booting up the board. This device tree file is configured to provide user space applications with network interfaces.

NOTE

After the above mentioned Device Tree configuration is used, all FMan ports are only available in the user space. Changes to the Device Tree file might be required to assign some of the FMan ports to Linux Kernel.

Optionally, follow the below instructions to assign one of the FMan ports on LS104x (DPAA) RDB boards to Linux.

With standard Yocto generated dtb, all interfaces are assigned to either Linux or User space. When using `fsl-ls1043a-rdb-sdk.dtb` or `fsl-ls1046a-rdb-sdk.dtb`, all network interfaces are assigned to Linux. When using `fsl-ls1043a-rdb-usdpaa.dtb` or `fsl-ls1046a-rdb-usdpaa.dtb`, all network interfaces are assigned to user space. The example below shows the changes that are required to assign one network interface to Linux and configure FMan to support DPDK applications.

Example: Modify `fsl-ls1046a-rdb-usdpaa.dts` file to assign FMan ports to Linux by removing the following Ethernet node that corresponds to `fm0-mac3` (RGMII-1).

```
ethernet@2 {
        compatible = "fsl,dpa-ethernet-init";
        fsl,bman-buffer-pools = <&bp7 &bp8 &bp9>;
        fsl,qman-frame-queues-rx = <0x54 1 0x55 1>;
        fsl,qman-frame-queues-tx = <0x74 1 0x75 1>;
        };
```

Then modify the `usdpaa_config_ls1046.xml` file (located in `/usr/share/dpdk/dpaa`) by removing the corresponding port entry. For example, the below entry must be removed for `fm0-mac3` (RGMII-1):

```
<port type="MAC" number="3" policy="hash_ipsec_src_dst_spi_policy_mac3"/>
```

On DPAA1, the port numbers are decided in the sequence they are getting detected. If one or more ports are assigned to Linux kernel, then the user space port numbering will change. For example, after the above code change is done, `fm0-mac4` becomes Port 0 in DPDK / user space.

9.1.4.2 Prerequisites for running DPDK applications

This section describes the procedures after the target platform is booted up and logged into the Linux shell. This section is applicable to DPAA, DPAA2, ENETC, and PPFE platforms and is organized as follows:

- Generic setup contains common steps to be executed before executing any of DPDK sample application or external DPDK applications. One of these sections would be relevant depending on the platform DPAA, DPAA2, ENETC, or PPFE being used.
- Application-specific sections contain steps on how to execute the DPDK example and related applications.

  For more details, see the following topics:

  — Test Environment Setup

  — Generic Setup - DPAA

  — Generic Setup - DPAA2

  — Generic Setup - PPFE

  — Generic setup – ENETC

  — DPAA2: Multiple parallel DPDK applications

9.1.4.2.1 Test Environment Setup

Test Environment Setup

Various sample application execution steps are detailed in the following sections. Figure below describes the setup containing the DUT (Device Under Test) and the Packet Generator (Spirent, Ixia or any other software/hardware packet generator). This is applicable for the commands provided in following section.

The setup includes a one-to-one link between DUT and Packet generator unit. DPDK application running on the DUT is expected to forward the traffic from one port to another. The setup below and commands described in following sections can be scaled for more number of ports.

Figure 17. Test Setup

9.1.4.2.2 Generic Setup - DPAA

This section details steps required to set up necessary environment for execution of DPDK applications on DPAA platform. This section is applicable for sample as well as any external DPDK applications. For further details about the applicable configuration file for DPAA platform, refer to Build OVS-DPDK using Yocto. For DPAA2 platform specific setup, refer to Generic Setup - DPAA2.

DPAA Hardware Configuration files

NOTE

For automatic or dynamic FMan queue configuration, use the `export DPAA_FMCLESS_MODE=1` environment variable. If this environment variable is set, DPDK-based DPAA driver would automatically configure the number of queues as demanded by the application.

Default is non-dynamic mode, which requires user to run the *fmc* tool with exact queue configuration before running a DPDK application. This section provides details about this mode.

DPAA platform supports hardware acceleration of packet queues. These queues must be configured in the Frame Manager (FMan) prior to being used. This can be done by choosing the appropriate policy configuration file packaged along with Yocto rootfs or DPDK source code.

A 1-, 2-, or 4-queue-based policy file can be selected before application is executed. For example, 1 queue policy file would define single queue per physical interface of DPAA. Similarly, 2- or 4-queue-based files are for defining 2 or 4 queues for each defined interface, respectively.

NOTE

For switching between different number of queue configuration, *fmc* tool is required to be run each time with new policy files. Before running *fmc* tool, `fmc -x` should be executed to clean old configuration.

Following are the available platform-specific configuration files:

- `usdpaa_config_ls1043.xml` for LS1043ARDB board
- `usdpaa_config_ls1046.xml` for LS1046ARDB board
- `usdpaa_config_ls1046_frwy.xml` for LS1046AFRWY board

Following are the available policy files:

- `usdpaa_policy_hash_ipv4_1queue.xml` for 1 queue per port
- `usdpaa_policy_hash_ipv4_2queue.xml` for 2 queues per port
- `usdpaa_policy_hash_ipv4_4queue.xml` for 4 queues per port

NOTE

It is important to execute the applications using the same queue configuration as per the policy file used. This is because once the queue configuration is done, DPAA hardware would distribute packets across configured number of queues. Not consuming packets from any queue would lead to queue buildup eventually stopping the I/O.

Setting up the DPAA Environment

Configure number of queues using environment variable:

```
export DPAA_NUM_RX_QUEUES=<Number of queues>
```

Based on the number of queues defined in the above parameter, select the policy configuration file and execute the `fmc` binary:

```
fmc -x  # Clean any previous configuration/setting
fmc -c <Configuration file> -p <Policy File> -a
```

For example, the command for using 1 queue in the LS1043A platform is as follows:

```
export DPAA_NUM_RX_QUEUES=1
fmc -x
fmc -c ./usdpaa_config_ls1043.xml -p ./usdpaa_policy_hash_ipv4_1queue.xml -a
```

NOTE

It is important that value of `DPAA_NUM_RX_QUEUES` matches to the policy file being used. In case of mismatch, DPDK application may show unexpected behavior.

NOTE

LSDK 18.03 (or DPDK release 18.02) onwards DPAA platforms enables the push mode by default. That is, first 4 queues of an interface would be configured in Push mode, thereafter, all queues would use the default pull configuration. Push mode queues support higher performance configuration than standard pull mode queues, but are limited in numbers. To toggle the number of push mode queues, use the following environment variable:

```
#export DPAA_PUSH_QUEUES_NUMBER=0 <default value is 4>
```

Do note that configuring larger number of push mode queues than available (achievable), would lead to I/O failure. Max possible value of `DPAA_PUSH_QUEUES_NUMBER` on DPAA (LS1043A, LS1046A) is 8.

Set up hugepages for DPDK application to use for packet and general buffers. This step can be ignored if hugepages are already mounted. Use command `mount | grep hugetlbfs` to check if hugepages are already set up.

```
mkdir /dev/hugepages
```

```
mount -t hugetlbfs none /dev/hugepages
```

Now, DPDK sample applications are ready to be executed on the DPAA platform.

Cleanup of the DPAA Environment

To remove the configuration done using the `fmc` tool, use the `-x` parameter. It is a good practice to clean up the configuration before setting up a new configuration. Even in cases where change of configuration is required, for example, increasing the number of queues supported, following command can be used for cleaning up the previous configuration.

```
fmc -x
```

9.1.4.2.3 Generic Setup - DPAA2

This section details steps required to set up necessary environment for execution of DPDK applications over DPAA2 platform. This section is applicable for sample as well as any external DPDK applications. For further details about the applicable configuration file for DPAA2 platform, see Build OVS-DPDK using Yocto. For DPAA-platform-specific setup, see Generic Setup - DPAA.

These steps must be performed before running any DPDK application on the host.

Setting up the DPAA2 environment

For executing DPDK application on DPAA2 platform, a resource container needs to be created which contains all necessary interfaces to the DPAA2 hardware blocks. Necessary configuration scripts are provided with DPDK package for creating and destroying containers.

1. Configure the DPAA2 resource container with `dynamic_dpl.sh` script. This script is available under `/usr/share/dpdk/dpaa2` folder in the rootfs.

   ```
   cd /usr/share/dpdk/dpaa2     # Or, any other folder if custom installation of DPDK is done
   ./dynamic_dpl.sh <DPMAC1.id> <DPMAC2.id> ... <DPMACn.id>
   ```

   In the above command, `<DPMAC1.id>` refers to the DPAA2 MAC resource, for example, `dpmac.1` or `dpmac.2`. Modify the above command as per the number of physical MAC ports required by the application (constrained by availability and connectivity on the DUT).

Output of `dynamic_dpl.sh` command shows the name of the container created. This name is passed to DPDK applications using the `DPRC` environment variable. Following block shows sample output of the `dynamic_dpl.sh` command:

```
#################### Container  dprc.2  is created ####################
Container dprc.2 have following resources :=>

 * 16 DPBP
 * 5 DPCON
 * 4 DPSECI
 * 3 DPNI
 * 10 DPIO
 * 10 DPCI

######################### Configured Interfaces ##########################

Interface Name        Endpoint              Mac Address
==============        ========              ==================
dpni.1                dpmac.1               -Dynamic-
dpni.2                dpmac.2               -Dynamic-
```

The DPDK applications automatically assign the MAC addresses after fetching information from the firmware. These MAC addresses are same as the ones programmed by U-Boot. For creating flows, see the application output or note the MAC addresses during board bootup. Testpmd application can also be used to find the MAC address assigned.

NOTE

For using UEFI-ACPI as bootloader, run `export BOARD_TYPE=2160` or `2088` before running `dynamic_dpl.sh`.

NOTE

Number of interfaces (such as `DPBP`, `DPCON`, and `DPNI`) in a container can be modified by defining environment variable `COMPONENT_COUNT=<number>` before executing the script. For example, to set number of DPBP to 4, use `export DPBP_COUNT=4`.

NOTE

Although flexibility has been provided to modify the interfaces in the container, the resources must be balanced. Changing any count requires corresponding changes to other interfaces. Incorrect changes may cause the DPDK application fail to execute.

2. Set up the environment variable using the container name reported by `dynamic_dpl.sh` command:

```
export DPRC=dprc.2
```

After the above setup is complete, DPDK application can be executed on the DPAA2 platform.

Teardown of DPAA2 environment

It might be required to change the configuration of the resource contain to modify the components included in it. As the number of resources in the system are limited, number of containers which can be created as also limited. It is possible to remove an existing container and create another.

Execute the following command to teardown a container:

```
cd /usr/share/dpdk/dpaa2                  # Or, any other folder if custom installation of DPDK
is done
./destroy_dynamic_dpl.sh <Container Name>  # for example, "dprc.2"
```

9.1.4.2.4 Generic Setup - PPFE

This section provides steps required to set up necessary environment for execution of DPDK applications over PPFE platform.

These steps must be performed before running any of the DPDK applications on host.

Setting up the PPFE Environment

From LSDK 2012 onwards, PPFE is a builtin driver and by default enabled in kernel mode. For executing DPDK application on PPFE platform, `pfe.us=1` need to be added in the bootargs before booting linux to enable DPDK mode which will do the necessary initialization to run the DPDK applications. User must ensure the value of `/sys/module/pfe/parameters/us` is 1 to check pfe driver is loaded in user space mode. If `/sys/module/pfe/parameters/us` is not 1, then user shall reboot kernel and add `pfe.us=1` in kernel command line.

Additionally, user must run the below commands to fulfill DPDK applications huge pages requirements.

```
mkdir /dev/hugepages
mount -t hugetlbfs none /dev/hugepages
```

NOTE

For PPFE (LS1012 platform), when using Crypto drivers, CONFIG_RTE_LIBRTE_PMD_CAAM_JR_BE flag should be enabled while compiling DPDK.

NOTE

Also, for Crypto functionality, Linux CAAM Job Ring driver must be enabled. In case Linux CAAM driver has been built as a kernel module, it must be loaded (`insmod caam_jr.ko`) in kernel before running DPDK application.

Hereafter, DPDK sample applications are ready to be executed.

9.1.4.2.5 Generic setup – ENETC

This section details steps required to set up necessary environment for execution of DPDK applications over ENETC platform. This section is applicable for sample as well as any external DPDK applications.

These steps must be performed before running any DPDK application on the host.

Setting up ENETC environment

For executing DPDK application on ENETC platform, Ethernet devices must be bound to "vfio-pci" driver. Necessary configuration script is provided with DPDK package.

This script is available under /usr/share/dpdk/enetc folder in the rootfs.

```
cd /usr/share/dpdk/enetc     # Or, any other folder if custom installation of DPDK is done
./dpdk_configure_1028ardb.sh
```

This script enables two Ethernet devices to be used by DPDK applications by binding them to "vfio_pci" driver. These devices on case are labeled as "1G MAC0" and "1G SWP0".

9.1.4.3 DPDK example applications

DPDK example application binaries are available in the `/usr/share/dpdk/examples` folder in the Yocto generated rootfs.

NOTE

Command snippets below assume that commands are executed while being present in `/usr/share/dpdk/examples` or appropriate `PATH` variable has been set. Also, a DPDK binary can be executed on both, DPAA and DPAA2, platform without any modifications.

NOTE

Only a selected few DPDK example applications have been deployed in the root filesystem by default. For non-deployed example application, compilation must be done using DPDK source code. See Standalone build of DPDK libraries and applications for more details.

NOTE

For PPFE platform, since LS1012ARDB has only 1 core, so `-c` with 0x1 is only acceptable core mask for all DPDK applications. Additionally, user must provide the `—vdev` argument with value `net_pfe` to enable Ethernet device for DPDK applications.

NOTE

For ENETC platform, LS1028A has 2 cores. For performance numbers, -c with 0x2 is the only supported core mask for all supported DPDK applications. User can verify functionality on both cores.

NOTE

Throughout the document below, `-n 1` argument has been added to the commands. This argument represents the splitting of buffers across the channels/ranks on DDR, if available. This is useful for NUMA cases. But, in non-NUMA, as is the case with NXP SoCs, this might impact performance in case the channel/ranks of DDR vary from standard/verified environment. Performance benchmarking should be done after analyzing the impact of this configuration.

l2fwd – Layer 2 forwarding application

Sample application to show forwarding between multiple ports based on the Layer 2 information (switching).

```
l2fwd -c 0x2 -n 1 -- -p 0x1 -q 1 -T 0
```

In the above command: `-c` refers to the core mask for cores to be assigned to DPDK; `-p` is the port mask for ports to be used by application; `-q` defines the number of queues to serve on each port. Other command line parameters may also be provided - for a complete list, see L2 Forwarding Sample Application (in Real and Virtualized Environments).

NOTE

- `isolcpus` provided as boot argument to U-Boot assures that isolated cores are not scheduled by Linux kernel. Using Core 0 for DPDK application can lead to non-deterministic behavior, including drop in performance. It is recommended that DPDK application core mask values avoid using Core 0.
- L2fwd application periodically prints the I/O stats. To avoid CPU core to be interrupted because of these scheduled prints, `-T 0` option can be appended at the end of command line.
- Command to run `l2fwd` on LS1012ARDB:

  ```
  ./l2fwd -c 0x1 -n 1 --vdev 'net_pfe0' --vdev='net_pfe1' -- -p 0x3 -q 3
  ```

NOTE

For best performance on LS1046ARDB, use the following command. This includes an option `-b 7` which sets optimal I/O burst size:

```
l2fwd -c 0x2 -n 1 -- -p 0x1 -q 1 -T 0 -b 7
```

l2fwd-event – Event-based Layer 2 forwarding application

Sample application to show event-based forwarding between multiple ports based on the Layer 2 information (switching).

```
l2fwd-event -c 0x2 -n 1 --vdev=event_dpaa2 -- -p 0x1 -q 1 -T 0 --mode=eventdev --eventq-sched=atomic
```

In the above command: `-c` refers to the core mask for cores to be assigned to DPDK; `-p` is the port mask for ports to be used by application; `-q` defines the number of queues to serve on each port; change `--vdev=event_dpaa1` for DPAA devices; `--mode` can be `poll` or `eventdev`; `--eventq-sched` can be `ordered`, `atomic`, or `parallel`. Other command line parameters may also be provided - for a complete list, see L2 Forwarding Eventdev Sample Application

l2fwd-qdma – Layer 2 forwarding application using QDMA (DPAA2 only)

Sample application to show forwarding between multiple ports based on the Layer 2 information (switching) using QDMA. In this application, when a packet is received, a corresponding packet buffer is allocated for Tx. Data from the Rx packet is DMA copied over to the Tx buffer using the QDMA block. Then, the application releases Rx buffer; Tx buffer is transmitted out.

```
l2fwd-qdma -c 0x2 -n 1 -- -p 0x1 -q 1 -m 1 -T 0
```

In the above command: `-c` refers to the core mask for cores to be assigned to DPDK; `-p` is the port mask for ports to be used by application; `-q` defines the number of queues to serve on each port. `-m` mode specifies HW (-m is 0) or Virtual (-m is 1) mode for QDMA queues. Apart from `-m` parameter other parameters are similar to DPDK l2fwd application. For more details, see L2 Forwarding Sample Application (in Real and Virtualized Environments).

l3fwd – Layer 3 forwarding application

Sample application to show forwarding between multiple ports based on the Layer 3 information (routing).

```
l3fwd -c 0x6 -n 1 -- -p 0x3 --config="(0,0,1),(1,0,2)"
```

In the above command: `-c` refers to the core mask for cores to be assigned to DPDK; `-p` is the port mask for ports to be used by application; `--config` is *(Port, Queue, Core)* configuration used by application for attaching cores to queues on each port. Other command line parameters may also be provided - for a complete list, see L3 Forwarding Sample Application.

Other variations of the above command described below change the configuration of ports and queues, and cores service them.

1. 4 core - 2 Port, 2 queues per port:

   ```
   l3fwd -c 0xF -n 1 -- -p 0x3 -P --config="(0,0,0),(0,1,1),(1,0,2),(1,1,3)"
   ```

2. 4 core - 2 Port with destination MAC address:

   ```
   l3fwd -c 0xF -n 1 -- -p 0x3 -P --config="(0,0,0),(0,1,1),(1,0,2),(1,1,3)" --eth-
   dest=0,11:11:11:11:11:11 --eth-dest=1,00:00:00:11:11:11
   ```

3. 8 core - 2 Port with 4 queues per port:

   ```
   l3fwd -c 0xFF -n 1 -- -p 0x3 -P --config="(0,0,0),(0,1,1),(0,2,2),(0,3,3),(1,0,4),(1,1,5),
   (1,2,6),(1,3,7)"
   ```

NOTE

Although above command snippets use Core 0 for DPDK application; however, for best performance, Core 0 is not recommended as Linux OS schedules its tasks on it. It is also recommended that `isolcpus` should be used in Linux boot argument to prevent Linux from scheduling tasks on other Cores.

NOTE

Example command to run l3fwd on LS1012ARDB:

```
./l3fwd -c 0x1 --vdev='net_pfe0' --vdev='net_pfe1' -n 1 -- -p 0x3 --
config="(0,0,0),(1,0,0)" -P
```

NOTE

For best performance on LS1046ARDB, use the following command. This includes an option `-b 7` which sets optimal I/O burst size:

```
l3fwd -c 0xF -n 1 -- -p 0x3 -P -b 7 --config="(0,0,0),(0,1,1),(1,0,2),(1,1,3)"
```

This is valid for any configuration of cores, queues, and ports (that is, `--config` option).

For LX2 platform, while running on all available cores, the core mask parameters passed to `l3fwd` must be adjusted for 16 available cores. Following is an example of using all 16 cores on LX2, 2 Ports, 8 queues per port:

```
l3fwd -c 0xffff -n 1 -- -p 0x3 -
P --config="(0,0,0),(0,1,1),(0,2,2),(0,3,3),(0,4,4),(0,5,5),(0,6,6),(0,7,7),
(1,0,8),(1,1,9),(1,2,10),(1,3,11),(1,4,12),(1,5,13),(1,6,14),(1,7,15)"
```

l2fwd-crypto – Layer 2 forwarding using SEC hardware

This variation of Layer 2 forwarding application uses SEC block for encryption of packets.

- Layer 2 forwarding with Cipher only support:

```
l2fwd-crypto -c 0x2 -n 1 -- -p 0x1 -q 1  --chain CIPHER_ONLY  --cipher_algo aes-cbc --cipher_op
ENCRYPT --cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10
```

- Layer 2 forwarding with Cypher-Hash support:

```
l2fwd-crypto -c 0x2 -n 1 -- -p 0x1 -q 1 --chain CIPHER_HASH  --cipher_algo aes-cbc --
cipher_op ENCRYPT --cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --auth_algo sha1-
hmac --auth_op GENERATE --auth_key_random_size 64
```

- Layer 2 forwarding with Hash only support:

```
l2fwd-crypto -c 0x2 -n 1 -- -p 0x1 -q 1 --chain HASH_ONLY --auth_algo sha1-hmac --auth_op
GENERATE --auth_key_random_size 64
```

NOTE

For LS1028, `--iova-mode=pa` should also be added as command line parameter.

l2fwd-crypto – Layer 2 forwarding using OpenSSL software instructions

This variation of Layer 2 forwarding application uses OpenSSL library for performing software crypto operations. Internally, the OpenSSL library would use the ARMCE instructions specific for Arm CPUs. For DPDK, this application uses the OpenSSL PMD as its underlying driver.

NOTE

This command requires support of OpenSSL package while building the DPDK applications. See this section for details about toggling compilation of software crypto support, which includes the OpenSSL driver.

NOTE

In all the commands described below, `-T 0` has been appended which disables output on the console/terminal. This is important for performance reasons. Though, for debugging purposes or for knowing the number of packets transacted, remove the arguments or set a higher value in seconds.

- Cipher_only
  - For DPAA Platform

- 1 core: Depending on the platform being executed on, append the above blacklisting parameter to the end of this command:

```
l2fwd-crypto --vdev "crypto_openssl"  -c 0x2 -n 1 -- -p 0x1 -q 1
--chain CIPHER_ONLY --cipher_algo aes-cbc --cipher_op ENCRYPT --cipher_key
01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --cryptodev_mask 0x10 -T 0
```

- 2 core: Depending on the platform being executed on, append the above blacklisting parameter to the end of this command:

```
l2fwd-crypto --vdev "crypto_openssl0" --vdev "crypto_openssl1" -c 0x6 -n 1 -- -p
0x3 -q 1  --chain CIPHER_ONLY  --cipher_algo aes-cbc --cipher_op ENCRYPT --cipher_key
01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --cryptodev_mask 0x30 -T 0
```

- 4 core: Depending on the platform being executed on, append the above blacklisting parameter to the end of this command:

```
l2fwd-crypto --vdev "crypto_openssl0" --vdev "crypto_openssl1" --vdev "crypto_openssl2"
--vdev "crypto_openssl3" -c 0xf -n 1 -- -p 0xf -q 1  --chain CIPHER_ONLY  --cipher_algo
aes-cbc --cipher_op ENCRYPT --cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10
--cryptodev_mask 0xF0 -T 0
```

— For DPAA2 Platform

- 1 core: Depending on the platform being executed on, append the above blacklisting parameter to the end of this command:

```
l2fwd-crypto --vdev "crypto_openssl" -c 0x2 -n 1 -- -p 0x1 -q 1
--chain CIPHER_ONLY --cipher_algo aes-cbc --cipher_op ENCRYPT --cipher_key
01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --cryptodev_mask 0x100 -T 0
```

- 2 core: Depending on the platform being executed on, append the above blacklisting parameter to the end of this command:

```
l2fwd-crypto --vdev "crypto_openssl0" --vdev "crypto_openssl1" -c 0x6 -n 1 -- -p
0x3 -q 1  --chain CIPHER_ONLY  --cipher_algo aes-cbc --cipher_op ENCRYPT --cipher_key
01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --cryptodev_mask 0x300 -T 0
```

- 4 core: Depending on the platform being executed on, append the above blacklisting parameter to the end of this command:

```
l2fwd-crypto --vdev "crypto_openssl0" --vdev "crypto_openssl1" --vdev "crypto_openssl2"
--vdev "crypto_openssl3" -c 0xf -n 1 -- -p 0xf -q 1  --chain CIPHER_ONLY  --cipher_algo
aes-cbc --cipher_op ENCRYPT --cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10
--cryptodev_mask 0xF00 -T 0
```

- Cipher_hash

— For DPAA Platform:

- 1 core: Append the above blacklisting parameter to the end of this command:

```
l2fwd-crypto --vdev "crypto_openssl" -c 0x2 -n 1 -- -p 0x1 -q 1
--chain CIPHER_HASH --cipher_algo aes-cbc --cipher_op ENCRYPT --cipher_key
01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --auth_algo sha1-hmac --auth_op GENERATE
--cryptodev_mask 0x10 --auth_key_random_size 64 -T 0
```

— For DPAA2 Platform:

- 1 core: Append the above blacklisting parameter to the end of this command:

```
l2fwd-crypto --vdev "crypto_openssl" -c 0x2 -n 1 -- -p 0x1 -q 1
--chain CIPHER_HASH --cipher_algo aes-cbc --cipher_op ENCRYPT --cipher_key
01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --auth_algo sha1-hmac --auth_op GENERATE
--cryptodev_mask 0x100 --auth_key_random_size 64 -T 0
```

In the above commands, for scaling to multiple cores or ports, toggle the `-c` and `-p` arguments as described above.

- Hash_cipher
  - For DPAA Platform:
    - 1 core: Append the above blacklisting parameter to the end of this command:

```
l2fwd-crypto --vdev "crypto_openssl" -c 0x2 -n 1 -- -p 0x1 -q 1 --chain HASH_CIPHER
--auth_algo sha1-hmac --auth_op GENERATE --cipher_algo aes-cbc --cipher_op ENCRYPT
--cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --cryptodev_mask 0x10 --
auth_key_random_size 64 -T 0
```

  - For DPAA2 Platform:
    - 1 core: Append the above blacklisting parameter to the end of this command:

```
l2fwd-crypto --vdev "crypto_openssl" -c 0x2 -n 1 -- -p 0x1 -q 1 --chain HASH_CIPHER
--auth_algo sha1-hmac --auth_op GENERATE --cipher_algo aes-cbc --cipher_op ENCRYPT
--cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --cryptodev_mask 0x100 --
auth_key_random_size 64 -T 0
```

In the above commands, for scaling to multiple cores or ports, toggle the `-c` and `-p` arguments.

- Hash_only
  - For DPAA Platform:
    - 1 core: Append the above blacklisting parameter to the end of this command:

```
l2fwd-crypto --vdev "crypto_openssl" -c 0x2 -n 1 -- -p 0x1 -q 1 --chain HASH_ONLY --
auth_algo sha1-hmac --auth_op GENERATE --cryptodev_mask 0x10 --auth_key_random_size 64
-T 0
```

  - For DPAA2 Platform:
    - 1 core: Append the above blacklisting parameter to the end of this command:

```
l2fwd-crypto --vdev "crypto_openssl" -c 0x2 -n 1 -- -p 0x1 -q 1 --chain HASH_ONLY --
auth_algo sha1-hmac --auth_op GENERATE --cryptodev_mask 0x100 --auth_key_random_size 64
-T 0
```

  - For scaling to multiple cores or ports, toggle the `-c` and `-p` arguments as described above.

For more information on L2fwd-crypto application, see L2 Forwarding with Crypto Sample Application.

NOTE

Example command to run l2fwd-crypto with openssl on LS1012ARDB (cipher only):

```
./l2fwd-crypto -c 0x1 --vdev='net_pfe0' --vdev='crypto_openssl' -n 1 -- -p 0x1 -q 1 --chain
CIPHER_ONLY --cipher_algo aes-cbc --cipher_key 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f --
cipher_op ENCRYPT -T 0
```

ipsec-secgw – IPsec gateway using SEC hardware

For IPsec application, two DUTs must be configured as `endpoint 0 (ep0)` and `endpoint 1 (ep1)`. Assuming that `endpoint` have **4 ports** each:

- Connect Port 1 and Port 3 of the `ep0` and `ep1` to each other (back-to-back).
- Connect Port 0 and Port 2 of the `ep0` and `ep1` to packet generator (for example, Spirent).

The stream generated by packet generator must have IP addresses in following pattern:

```
EP0:
    port 0: 32 flows with destination IP:
192.168.1.XXX, 192.168.2.XXX, ..... ,192.168.31.XXX,192.168.32.XXX
    port 2: 32 flows with destination IP:
192.168.33.XXX, 192.168.34.XXX, ..... ,192.168.63.XXX,192.168.64.XXX
EP1:
    port 0: 32 flows with destination IP:
192.168.101.XXX, 192.168.102.XXX, ..... ,192.168.131.XXX,192.168.132.XXX
    port 2: 32 flows with destination IP:
192.168.133.XXX, 192.168.134.XXX, ..... ,192.168.163.XXX,192.168.164.XXX
```

The above lines of code represents default configurations for the endpoints in `ep0_64X64.cfg` and `ep1_64X64.cfg`. Custom port mappings, SA/SP, and the routes can be configured in the corresponding configuration files, `ep0.cfg` and `ep1.cfg`, for respective endpoint. These files are available in Yocto generated rootfs. For more details, see this table.

For more information, see IPsec Security Gateway Sample Application.

Endpoint 0 (`ep0`) configuration:

```
ipsec-secgw -c 0xf -n 1 -- -p 0xf -P -u 0xa --config="(0,0,0),(1,0,1),(2,0,2),(3,0,3)" -
f ep0_64X64.cfg
```

Endpoint 1 (`ep1`) configuration:

```
ipsec-secgw -c 0xf -n 1 -- -p 0xf -P -u 0xa --config="(0,0,0),(1,0,1),(2,0,2),(3,0,3)" -
f ep1_64X64.cfg
```

NOTE

For LS1028, `--iova-mode=pa` should also be added as command line parameter.

Running IPsec gateway application with hardware protocol offload

The DPAA/DPAA2 SEC hardware also supports IPsec protocol offload. The command and configurations are exactly same except the cfg files. For protocol offload, the cfg files are `ep0_64X64_proto.cfg` and `ep1_64X64_proto.cfg`. Performance with protocol offload would be much better than the standard case. For platforms that have 8 cores, the command for 8 cores is also exactly same as for non-offload case, except the names of the cfg files.

Endpoint 0 (`ep0`) configuration:

```
ipsec-secgw -c 0xf -n 1 -- -p 0xf -P -u 0xa --config="(0,0,0),(1,0,1),(2,0,2),(3,0,3)" -f
ep0_64X64_proto.cfg
```

Endpoint 1 (`ep1`) configuration:

```
ipsec-secgw -c 0xf -n 1 -- -p 0xf -P -u 0xa --config="(0,0,0),(1,0,1),(2,0,2),(3,0,3)" -
f ep1_64X64_proto.cfg
```

NOTE

For LS1028, `--iova-mode=pa` should also be added as command line parameter.

Running IPsec gateway application with 8 cores

For running IPsec application with multiple queues using 64X64 tunnels and with 8 cores, following command and configuration must be done:

Endpoint 0 (`ep0`) configuration: Sample endpoint 0 configuration for such application is available in `ep0_64X64.cfg` file placed in `/usr/share/dpdk/dpaa2/` folder in root filesystem.

```
ipsec-secgw -c 0xFF -n 1 -- -p 0xf -P -u 0xa --config="(0,0,0),(0,1,1),(1,0,2),(1,1,3),(2,0,4),
(2,1,5),(3,0,6),(3,1,7)" -f ep0_64X64.cfg
```

Endpoint 1 (`ep1`) configuration: Sample endpoint 1 configuration for such application is available in `ep1_64X64.cfg` file placed in `/usr/share/dpdk/dpaa2/` folder in root filesystem.

```
ipsec-secgw -c 0xFF -n 1 -- -p 0xf -P -u 0xa --config="(0,0,0),(0,1,1),(1,0,2),(1,1,3),(2,0,4),
(2,1,5),(3,0,6),(3,1,7)" -f ep1_64X64.cfg
```

Running IPsec gateway application with 16 cores on LX2 platform

For running IPsec application with multiple queues using 64X64 tunnels and with 16 cores, following command and configuration must be done:

Endpoint 0 (`ep0`) configuration: Sample endpoint 0 configuration for such application is available in `ep0_64X64_sha256_proto.cfg` file placed in `/usr/share/dpdk/dpaa2/` folder in root filesystem.

```
./ipsec-secgw -c 0xFFFF  -n 1 -- -p 0xf -P -u 0xa --config="(0,0,0),(0,1,1),(0,2,2),(0,3,3),(1,0,4),
(1,1,5),(1,2,6),(1,3,7),(2,0,8),(2,1,9),(2,2,10),(2,3,11),(3,0,12),(3,1,13),(3,2,14),(3,3,15)" -
f ep0_64X64_sha256_proto.cfg
```

Endpoint 1 (`ep1`) configuration: Sample endpoint 1 configuration for such application is available in `ep1_64X64_sha256_proto.cfg` file placed in `/usr/share/dpdk/dpaa2/` folder in root filesystem.

```
./ipsec-secgw -c 0xFFFF -n 1 -- -p 0xf -P -u 0xa --config="(0,0,0),(0,1,1),(0,2,2),(0,3,3),(1,0,4),
(1,1,5),(1,2,6),(1,3,7),(2,0,8),(2,1,9),(2,2,10),(2,3,11),(3,0,12),(3,1,13),(3,2,14),(3,3,15)" -
f ep1_64X64_sha256_proto.cfg
```

ipsec-secgw – IPsec gateway using OpenSSL PMD

The command, flow stream, and port configuration are similar to the ipsec-secgw – IPsec gateway using SEC hardware command, flow stream, and port configuration, except that it uses OpenSSL PMD for crypto operations. Internally, the OpenSSL PMD uses the ARMCE instructions for the Arm CPUs for performing crypto operations.

- For DPAA Platform:
  - Endpoint 0 configuration

    ```
    ipsec-secgw -c 0xf -n 1 --vdev "crypto_openssl" -- -p 0xf -P -u 0xa --config="(0,0,0),(1,0,1),
    (2,0,2),(3,0,3)" --cryptodev_mask 0x10 -f ep0_64X64.cfg
    ```

    Endpoint 1 configuration

    ```
    ipsec-secgw -c 0xf -n 1 --vdev "crypto_openssl" -- -p 0xf -P -u 0xa --config="(0,0,0),(1,0,1),
    (2,0,2),(3,0,3)" --cryptodev_mask 0x10 -f ep1_64X64.cfg
    ```

- For DPAA2 Platform:

— Endpoint 0 configuration

```
ipsec-secgw -c 0xf -n 1 --vdev "crypto_openssl" -- -p 0xf -P -u 0xa --config="(0,0,0),(1,0,1),
(2,0,2),(3,0,3)" --cryptodev_mask 0x100 -f ep0_64X64.cfg
```

Endpoint 1 configuration

```
ipsec-secgw -c 0xf -n 1 --vdev "crypto_openssl" -- -p 0xf -P -u 0xa --config="(0,0,0),(1,0,1),
(2,0,2),(3,0,3)" --cryptodev_mask 0x100 -f ep1_64X64.cfg
```

NOTE

Example command to run ipsec-secgw with openssl on LS1012ARDB:

```
./ipsec-secgw -c 0x1 -n 1 --vdev='net_pfe0' --vdev='net_pfe1' --
vdev='crypto_openssl'  -- -p 0x3 -P -u 0x2 --config="(0,0,0),(1,0,0)" -
f ep0_64X64.cfg
```

KNI - Using Kernel Network Interface Module

The Kernel NIC Interface (KNI) is a DPDK control plane solution that allows user space applications to exchange packets with the kernel networking stack. For more details, refer to: http://dpdk.org/doc/guides/sample_app_ug/kernel_nic_interface.html

Loading the KNI kernel module without any parameter. By default only one kernel thread is created for all KNI devices for packet receiving in kernel side:

```
#insmod rte_kni.ko
```

Affine the kni task to a single core, for example, core number #1:

```
#taskset -pc 1 `pgrep -fl kni_single | awk '{print $1}'`
```

Run the kni application:

Command syntax:

```
kni [EAL options] -- -P -p PORTMASK --config="(port,lcore_rx,lcore_tx[,lcore_kthread,..])
[,port,lcore_rx,lcore_tx[,lcore_kthread,..]]"
```

Command used:

```
./kni -c 0xf -n 1 -- -p 0x3 -P --config="(0,0,1),(1,2,3)"
```

where config is port, kni lcore Rx core, or kni lcore Tx core.

NOTE

Each core can either do Tx or Rx for one port only.

On another console, check the interfaces with:

```
#ifconfig -a
```

Enable the given interface and assign IP address (if any).

QDMA demo application (DPAA2 only)

NOTE

`qdma_demo` application is not available as default in the LSDK rootfs. For compiling this application, see Compiling DPDK Example Applications.

On DPAA2, DPDMAI block provides a generic DMA capability which has been exposed by DPDK for its application to use. `qdma-demo` application in DPDK is a demonstration application which does a memory-to-memory or memory to pci memory or pci memory to memory transaction using this QDMA block. It can be executed in following manner:

- For MEM-to MEM use case, run below command.

```
qdma_demo -c 0x3 -- --packet_size=512 --test_case=mem_to_mem
```

NOTE

`qdma_demo` requires more than 1 core to perform because it consumes at least one core for printing the output. Refer to examples/qdma_demo/readme in DPDK source code for more details on usages.

This would print to screen an output similar to:

```
Time Spend :4000.005 ms rcvd cnt:1310720 pkt_cnt:0
Rate: 1342.176 Mbps OR 327.680 Kpps
processed on core 7 pkt cnt: 1310720
```

This output demonstrates the count of memory chunks which have been moved through QDMA block by the application. It also shows the time spent, the performance achieved, and packets sent per core.

- For PCI-to-MEM and MEM-to-PCI, run following commands

Endpoint steps

- For LX2-EP PCI card: Boot it to U-Boot prompt only
- For standard PCI NIC card: Do nothing

HOST - LX2 root complex steps

1. Boot LX2 to Linux prompt.
2. Run 'lspci -v' to check the address of BAR whose memory is targeted memory for test:

```
$ lspci -v
        0000:01:00.0 Ethernet controller: Intel Corporation 82574L Gigabit Network Connection
       Subsystem: Intel Corporation Gigabit CT Desktop Adapter
        Flags: bus master, fast devsel, latency 0, IRQ 106
        Memory at 30460c0000 (32-bit, non-prefetchable) [size=128K]
        Memory at 3046000000 (32-bit, non-prefetchable) [size=512K]
        I/O ports at 1000 [disabled] [size=32]
        Memory at 30460e0000 (32-bit, non-prefetchable) [size=16K]
        Expansion ROM at 3046080000 [disabled] [size=256K]
Kernel driver in use: e1000e
```

3. Assign PCI device to userspace:
   - Load the UIO module, if not loaded already:

     ```
     #modprobe uio_pci_generic
     ```

   - Assign the device to userspace:

     ```
     #/usr/share/dpdk/usertools/dpdk-devbind.py --bind=uio_pci_generic 0000:01:00.1
     ```

4. Run qdma_demo application:

   NOTE

   At least two cores are required to run the test, one core is used for printing results/stats, other cores for running test.

```
$export DPDMAI_COUNT=48
$./dynamic_dpl.sh dpmac.3
$export DPRC=dprc.2
```

   mem to pci (using a Gen2-x1 - 1G PCI NIC)

```
  $./qdma_demo -c 0x81 -- --pci_addr=0x3046000000 --packet_size=512 --test_case=mem_to_pci
```

   NOTE

   The current QDMA demo code reads/writes only 4 KB area, which yields best bandwidth number. To test read/write large memory size, you can optionally pass `--pci_size` (size in byte), for example, for 2 MB PCI use: `--pci_size`=2147483648. Add "`--latency_test`" for testing latency.

Pktgen – DPDK-based software packet generator

Pktgen is a software packet generator based on DPDK. See DPDK-based Packet Generator for steps required for building *Pktgen*.

All the commands below assume that *Pktgen* application is either executed from current folder or appropriate path environment variable has been set.

1. 3 Port, 1 Core each:

```
pktgen -l 0-3 -n 1 --proc-type auto --file-prefix pg --log-level 8 -- -T -P -m "[1].0,
[2].1, [3].2"
```

2. 1 Port, 2 Core:

```
pktgen -l 0-3 -n 1 --proc-type auto --file-prefix pg --log-level 8 -- -T -P -m "[1:2].0"
```

3. To start or stop traffic on a specific port:

```
start 0 # start <port number>
stop 0  # stop <port number>
```

4. To start or stop traffic on all ports:

```
str
stp
```

9.1.4.4 DPAA2: Multiple parallel DPDK applications

This section describes steps for executing multiple parallel DPDK applications on DPAA2 platform.

For executing multiple DPDK applications, each application instance should run with its own resource container (*DPRC*). This constraint is because of the way DPDK framework is designed to use a given container for exclusive use, irrespective of resources within, and bind it using VFIO layer. This design prevents parallel access to single resource container from multiple parallel instances of a single DPDK application and multiple parallel execution of different DPDK applications.

Creating multiple DPRC instances

Using the resource container script documented in this section, create multiple resource container instances on host. Following command creates a resource container with 2 network interfaces (and all other resources necessary to run a DPDK application).

First DPRC: (assuming name as `dprc.2` through rest of the document)

```
cd /usr/share/dpdk/dpaa2                    # Or, any other folder if custom installation of DPDK
is done
./dynamic_dpl.sh <DPMAC1.id> <DPMAC2.id>    # For example, execute ./dynamic_dpl.sh dpmac.1 dpmac.2
```

Second DPRC: (assuming name as `dprc.3` through rest of the document)

```
cd /usr/share/dpdk/dpaa2                    # Or, any other folder if custom installation of DPDK
is done
./dynamic_dpl.sh <DPMAC3.id> <DPMAC4.id>    # For example, execute ./dynamic_dpl.sh dpmac.3 dpmac.4
```

Executing multiple DPDK Applications

Once the resource containers are created, on two separate terminals, execute the following commands to run **l2fwd** application, bridging traffic between both interfaces available in the container:

```
export DPRC=dprc.2
cd /usr/share/dpdk/examples
./l2fwd -c 0x3 -n 1 --file-prefix=p1 --socket-mem=1024 -- -p 0x3 -q 1
```

Some of the arguments, which are deviations from general **l2fwd** command, are explained below:

`--file-prefix`: Each DPDK Application attempts to allocate some hugepages for DMA'd area. This allocation is done in the hugepages using *hugepage* mount, by creating and mapping a file. This argument instructs the EAL to append a string to the filename. This way, multiple instances, having different such arguments, do not attempt to open same hugepage mapping file.

`--socket-mem`: Passed to EAL, this argument instructs the EAL to allocate only specified amount of memory from the hugepages. By default, if this argument is not provided, a DPDK application acquires all possible hugepages (all free pages) available on the Linux system.

For the second instance, command like following can be executed:

```
export DPRC=dprc.3
cd /usr/share/dpdk/examples
./l2fwd -c 0xc -n 1 --file-prefix=p2 --socket-mem=1024 -- -p 0x3 -q 1
```

Note the difference of values for **`-c`** and **`--file-prefix`** between the first and second command.

9.1.5 OVS-DPDK and DPDK in VM with VIRTIO Interfaces

DPDK example and DPDK-based applications can also run inside the virtual machine. This section describes steps to run these applications inside the virtual machine on both DPAA1 and DPAA2 platforms.

The virtual machine runs inside the host Linux system and is launched by an application called QEMU.

> **NOTE**
>
> While using the virtual machine, the console logs for the guest Linux do not appear on the host Linux console (i.e UART). The guest logs are exposed through `telnet`, and they can be accessed by doing `telnet` on the host board's IP Address (`IP_ADDR_BRD`) and `GUEST_CONSOLE_TELNET_PORT`. Each Virtual machine that is run on a single host is allocated a different `GUEST_CONSOLE_TELNET_PORT`, and this port number is specified by user running virtual machine through the QEMU command-line.

Following is the layout of the subsections of this chapter:

- Generic steps describing steps required for QEMU setup for both, DPAA1 and DPAA2 platforms.
- Configuring OVS describing steps necessary to launch OVS-DPDK on the host machine for switching traffic between VMs and external network.
- Various sections for launching a virtual machine and executing a DPDK application:

- Launch Virtual Machine for launching a virtual machine.
- Accessing virtual machine console for accessing a virtual machine console from a network connected machine over `telnet`.
- Launching two virtual machines for launching more than one virtual machine.
- Running DPDK applications in VM for running DPDK applications in the virtual machine.

- Multi-queue VIRTIO support describes steps for DPDK application using multiple queues.

9.1.5.1 Generic steps

See QEMU for detailed information about deploying virtual machines using KVM/QEMU on Layerscape boards.

The reference above serves as base for deploying Virtual Machines and DPDK application in them. All following sections assume that kernel image and virtual machine rootfs is available with DPDK sample application images in it.

NOTE

Give IP address to the board so that virtual machine console can be accessed using telnet.

```
ifconfig eth<x> <IP_ADDR_BRD>
```

9.1.5.2 Configuring OVS

OVS-DPDK application binary and configuration files are available in the `/usr/bin/ovs-dpdk/` folder in the Yocto generated rootfs.

It is assumed that before executing command snippets in this section, necessary steps mentioned in Generic steps have already been executed.

NOTE

Command snippets below assume that commands are executed while being present in `/usr/bin/ovs-dpdk/` folder. Or, appropriate `PATH` variable has been set. As the OVS commands are spread across multiple folders, each command snippet also shows the location of these binaries relative to above folder.

Command snippets below assume that commands are executed while being present in this folder or appropriate PATH variable has been set.

OVS is used as a back-end for VHOST USER ports. The physical ports on the target platform and the vhost user ports (virtio devices) are added to ovs-vswitch, and the flows in OVS are programmed to establish traffic switching between physical ports and vhost devices as follows:

- Incoming traffic Physical port1 => output to vhost-user port 1
- Incoming traffic on vhost-user port1 => output on physical port 1
- Incoming traffic on physical port 2 => output on vhost-user port 2
- Incoming traffic on vhost-user port 2 => output on physical port 2

The following steps must be followed to set up OVS as vhost switching back-end:

1. Reset the OVS environment.

```
pkill -9 ovs
```

```
rm /usr/local/etc/openvswitch/conf.db
```

```
rm -rf /usr/local/var/run/openvswitch/vhost-user-1
```

```
rm -rf /usr/local/var/run/openvswitch/vhost-user-2
```

2. Specify the initial Open vSwitch (OVS) database to use:

```
mkdir -p /usr/local/etc/openvswitch  # If the folder doesn't already exist
```

```
mkdir -p /var/log/openvswitch        # to ensure that OVS logging can be done
```

```
mkdir -p /usr/local/var/run/openvswitch
```

```
cd /usr/bin/ovs-dpdk/
./ovsdb-tool create /usr/local/etc/openvswitch/conf.db ./vswitch.ovsschema
```

```
./ovsdb-server --remote=punix:/usr/local/var/run/openvswitch/db.sock --
remote=db:Open_vSwitch,Open_vSwitch,manager_options --pidfile=/tmp/ovsdb-server.pid --detach --
log-file=/var/log/openvswitch/ovs-vswitchd.log
```

```
export DB_SOCK=/usr/local/var/run/openvswitch/db.sock
```

3. Configure the OVS to support DPDK ports:

```
./ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-init=true
```

4. Configure OVS to work with memory backed by hugepages:
   - For DPAA platform: Configure OVS to work with 200 M memory backed by hugepages:

```
export SOCK_MEM=200
./ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-socket-mem="$SOCK_MEM"
```

   - For DPAA2 platform: Configure OVS to work with 1G (1024 M) memory backed by hugepages:

```
export SOCK_MEM=1024
./ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-socket-mem="$SOCK_MEM"
```

5. Define Cores for OVS Operations

```
export OVS_SERVICE_MASK=0x1
export OVS_CORE_MASK=0x6
```

```
./ovs-vsctl  --no-wait set Open_vSwitch . other_config:dpdk-lcore-mask=$OVS_SERVICE_MASK
```

```
./ovs-vsctl  --no-wait set Open_vSwitch . other_config:pmd-cpu-mask=$OVS_CORE_MASK
```

NOTE

`OVS_CORE_MASK` should be chosen such that Core 0 is not included. `OVS_SERVICE_MASK` should be any core which is not already assigned to `OVS_CORE_MASK`. This way, OVS service threads (defined by `OVS_SERVICE_MASK`) do not compete for CPU scheduling with OVS I/O threads (`OVS_CORE_MASK`). `OVS_SERVICE_MASK` can be set to Core 0 as defined in example above.

6. Set Exact Match Cache (EMC) Insertion probability to 1 so that cache insertion is performed for every flow.

```
$ ovs-vsctl --no-wait set Open_vSwitch . other_config:emc-insert-inv-prob=1
```

7. Start the ovs-vswitchd daemon:

```
./ovs-vswitchd unix:$DB_SOCK --pidfile --detach
```

NOTE

`--detach` option makes the daemon run in background. If this option is given same shell can be used to run further commands, otherwise ssh to the target board and run further commands. Each time you reboot or there is an OVS termination, you must rebuild the OVS environment and repeat steps 1-6 of this section.

8. Create an OVS bridge.

```
./ovs-vsctl add-br br0 -- set bridge br0 datapath_type=netdev
```

9. Create DPDK port

   For creating DPDK ports with OVS, platform-specific port information must be provided to OVS.

   - ```
     ./ovs-vsctl add-port br0 dpdk0 -- set Interface dpdk0 type=dpdk options:dpdk-devargs=dpni.1
     ```

     ```
     ./ovs-vsctl add-port br0 dpdk1 -- set Interface dpdk1 type=dpdk options:dpdk-devargs=dpni.2
     ```

     Above commands attach the DPAA2 ports `dpni.1` and `dpni.2` with OVS. In case different ports are required, above command should be modified accordingly.

     NOTE

     For DPAA ports, replace `dpni.X` with `fm1-macX`. For example, `options:dpdk-devargs=fm1-mac3`.

     NOTE

     Another way to pass device names to OVS is to pass along with bus name. For example, for FSLMC/ DPAA2 devices, `options:dpdk-devargs=`**`fslmc:dpni.1`** can be used. For DPAA1, `options:dpdk-devargs=`**`dpaa:fm1-mac3`** can be used. DPDK would be able to parse either naming style, whether provided with bus name or without.

10. Create vhost-user port

```
./ovs-vsctl add-port br0 vhost-user1 -- set Interface vhost-user1 type=dpdkvhostuser
```

```
./ovs-vsctl add-port br0 vhost-user2 -- set Interface vhost-user2 type=dpdkvhostuser
```

11. Commands to Configure Multi Queues

```
./ovs-vsctl set Interface dpdk0 options:n_rxq=4
./ovs-vsctl set Interface dpdk1 options:n_rxq=4
./ovs-vsctl set Interface dpdk0 options:n_txq=4
./ovs-vsctl set Interface dpdk1 options:n_txq=4
```
```

NOTE

The above commands are required only for multi-queue use case (four queues are used in above reference commands). For single queue mode, no commands are needed as OVS is configured for single queue, by default.

12. Delete OVS flow:

```
./ovs-ofctl del-flows br0
```

13. Set OVS flow rules for external-to-external path:

NOTE

The commands below configure a hard-coded bidirectional data path between Port 1 and Port 2. Use this step only for OVS external-to-external testing. For OVS Host-to-VM configuration, skip and continue with next step.

```
./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=1,actions=output:2
```

```
./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=2,actions=output:1
```

14. Set OVS flow rules between Host to VM:

NOTE

The steps below configure OVS such that `Port 1 <=> Port 3` and `Port 2 <=> Port 4` are connected to each other. If a different configuration is required, the commands below should be altered as well as VM configurations.

```
./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=1,actions=output:3
```

```
./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=3,actions=output:1
```

```
./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=2,actions=output:4
```

```
./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=4,actions=output:2
```

NOTE

OVS Switch (`ovs-vswitchd`) must be run before launching the virtual machine using QEMU; otherwise, the virtual machine launch fails.

15. To enable emc-cache lookups in OVS, run the following command. This improves lookup speed and ensures better performance.

```
./ovs-vsctl --no-wait set Open_vSwitch . other_config:emc-insert-inv-prob=1
```

16. Verify the flows inserted:

```
./ovs-ofctl dump-flows br0
```

NOTE

Performance of OVS is highly dependent on the use case - which includes the configuration of flows, the flows being pumped, SMC/EMC configuration, and so on. It is important to analyze these dependencies before performance measurement or benchmarking can be done. For performance benchmarking, it is preferred that 256 flows are configured in the environment. If the number of flows is low, then distribution (RSS) may be impacted. On the other hand, if the number of flows is high, then cache usage is impacted.

9.1.5.3 Launch Virtual Machine

This section describes necessary environment setup and commands for launching a Virtual Machine (VM).

It is assumed that before executing command snippets in this section, necessary steps mentioned in Generic steps and Configuring OVS have already been executed.

Set up the environment

For accessing the VM, *telnet* is used. This environment variable defines the *telnet* port to be used.

```
export GUEST_CONSOLE_TELNET_PORT=4446    # Telnet port to be used for accessing the virtual machine
```

```
export ROOTFS_IMG=<VM_ROOTFS_IMG>
```

Define other environment variables which are used by the QEMU command to configure the virtual machine environment:

```
export VM_MEM=2048M
export VM_CORES=2
export NUM_QUEUES=1
```

NOTE

- VM_CORES are the number of cores to reserve for the virtual machine operation.

Export the following paths:

```
export VHOST1_PATH=/usr/local/var/run/openvswitch/vhost-user1
export VHOST2_PATH=/usr/local/var/run/openvswitch/vhost-user2
```

Launch QEMU and virtual machine

Launch the QEMU emulator using the following command.

```
qemu-system-aarch64 -nographic -object memory-backend-file,id=mem,size=$VM_MEM,mem-path=/mnt/
hugepages,share=on -cpu host -machine type=virt -kernel /boot/Image -enable-kvm  -serial
tcp::$GUEST_CONSOLE_TELNET_PORT,server,telnet -append 'root=/dev/vda rw console=ttyAMA0,115200
rootwait earlyprintk' -m $VM_MEM -numa node,memdev=mem -chardev socket,id=char1,path=$VHOST1_PATH
-netdev type=vhost-user,id=hostnet1,chardev=char1,vhostforce,queues=$NUM_QUEUES -
device virtio-net-pci,disable-modern=false,addr=0x3,netdev=hostnet1,id=net1,mrg_rxbuf=off
-chardev socket,id=char2,path=$VHOST2_PATH -netdev type=vhost-
user,id=hostnet2,chardev=char2,vhostforce,queues=$NUM_QUEUES -device virtio-net-pci,disable-
modern=false,addr=0x4,netdev=hostnet2,id=net2,mrg_rxbuf=off -smp $VM_CORES -S -drive
if=none,file=$ROOTFS_IMG,id=foo,format=raw -device virtio-blk-device,drive=foo
```

NOTE

For best performance, Core 0 in the VM should not be used for DPDK I/O threads.

Also, to avoid system services from using GPUs scheduled for DPDK I/O threads, it is recommended that `isolcpus` is used for isolating cores from Linux Kernel scheduling in VM. The exact configuration is dependent on number of CPU assigned by QEMU to VM using the `VM_CORES` environment variable.

Append `isolcpus=1-$VM_CORES` to the '`root=/dev/vda rw console=ttyAMA0,115200 rootwait earlyprintk`' string in the `qemu-system-aarch64` command given above.

NOTE

Extra care should be taken for value assigned to `mem-path` variable. It should point to a valid mounted hugepage filesystem. In case the value assigned to `mem-path` is not a valid hugepage filesystem, QEMU would create a mmaped file for its work which might negatively impact performance.

Following log appears on the host UART console:

```
QEMU 2.11.1 monitor - type 'help' for more information
(qemu) QEMU waiting for connection on: disconnected:telnet::4446,server
```

NOTE

Complete QEMU logs are visible only when `telnet` is used for logging into the guest machine, as described in Accessing virtual machine console.

The `-S` option mentioned in the qemu command stops the virtual machine bootup after initial setup. Run the `info cpus` command on QEMU CLI interface to see the QEMU threads.

```
(qemu) info cpus
* CPU #0: thread_id=2559
  CPU #1: (halted) thread_id=2560
```

SSH on the board (`telnet` to `IP address IP_ADDR_BRD`) from other console and affine the threads to the cores using the `taskset` command:

```
taskset -p 0x4 <tid1>
taskset -p 0x8 <tid1>
```

NOTE

It is recommended to affine the VCPUs to the cores on which OVS threads are not running. For better performance, VCPU threads should be given one physical CPU each, if possible.

Run the `c` command from the QEMU CLI to continue the VM boot-up:

```
(qemu) c
```

9.1.5.4 Launching two virtual machines

This section describes steps for launching 2 virtual machine simultaneously for multiple VM use cases.

NOTE

- Memory assigned to each virtual machine should not exceed the total number of huge pages assigned on system. In following example, 2048 Mbit to each virtual machine has been specified and verified to be working correctly.
- Console `telnet` port of both virtual machine must be different. In the below example, VM1 has port 4446 and VM2 has port 4447 configured for `telnet`. Modify the command accordingly if different values are required.

Launch VM1:

```
qemu-system-aarch64 -nographic -object memory-backend-file,id=mem,size=$VM_MEM,mem-path=/dev/
hugepages,share=on -cpu host -machine type=virt -kernel /boot/Image -enable-kvm -
serial tcp::4446,server,telnet -append 'root=/dev/vda rw console=ttyAMA0,115200 rootwait
earlyprintk' -m $VM_MEM -numa node,memdev=mem -chardev socket,id=char1,path=$VHOST1_PATH
-netdev type=vhost-user,id=hostnet1,chardev=char1,vhostforce,queues=$NUM_QUEUES -
device virtio-net-pci,disable-modern=false,addr=0x3,netdev=hostnet1,id=net1,mrg_rxbuf=off
-chardev socket,id=char2,path=$VHOST2_PATH -netdev type=vhost-
user,id=hostnet2,chardev=char2,vhostforce,queues=$NUM_QUEUES -device virtio-net-pci,disable-
modern=false,addr=0x4,netdev=hostnet2,id=net2,mrg_rxbuf=off -smp $VM_CORES -S -drive
if=none,file=$ROOTFS_IMG,id=foo,format=raw -device virtio-blk-device,drive=foo
```

Launch VM2:

```
qemu-system-aarch64 -nographic -object memory-backend-file,id=mem,size=$VM_MEM,mem-path=/dev/
hugepages,share=on -cpu host -machine type=virt -kernel /boot/Image -enable-kvm -
serial tcp::4447,server,telnet -append 'root=/dev/vda rw console=ttyAMA0,115200 rootwait
earlyprintk' -m $VM_MEM -numa node,memdev=mem -chardev socket,id=char1,path=$VHOST1_PATH
-netdev type=vhost-user,id=hostnet1,chardev=char1,vhostforce,queues=$NUM_QUEUES -
device virtio-net-pci,disable-modern=false,addr=0x3,netdev=hostnet1,id=net1,mrg_rxbuf=off
-chardev socket,id=char2,path=$VHOST2_PATH -netdev type=vhost-
user,id=hostnet2,chardev=char2,vhostforce,queues=$NUM_QUEUES -device virtio-net-pci,disable-
modern=false,addr=0x4,netdev=hostnet2,id=net2,mrg_rxbuf=off -smp $VM_CORES -S -drive
if=none,file=$ROOTFS_IMG,id=foo,format=raw -device virtio-blk-device,drive=foo
```

9.1.5.5 Running DPDK applications in VM

All the DPDK applications mentioned in this section have been tested in following configuration:

- Two Physical network interfaces.
- Two virtio-net devices in the virtual machine.

Following figure illustrates the test setup.

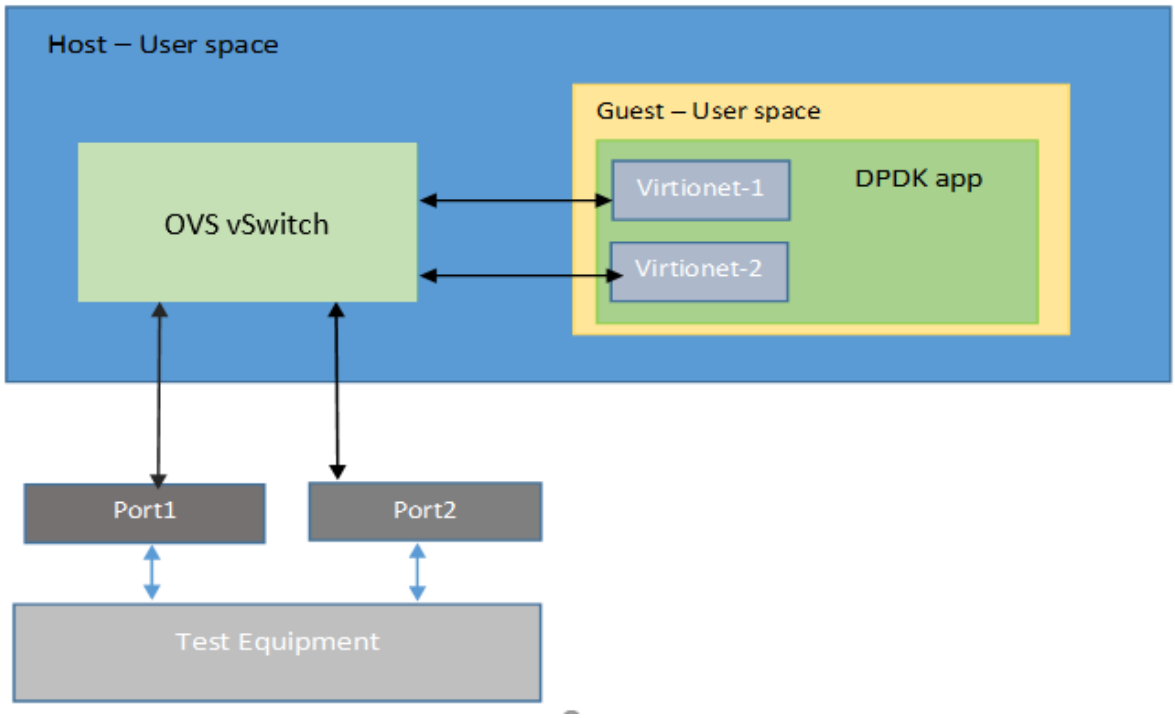


Figure 18. DPDK virtio-net test setup

Generic setup

DPDK example application binaries are available in the `/usr/share/dpdk/examples` folder in the Yocto generated rootfs.

- Set up Hugepages:

```
mkdir /dev/hugepages
mount -t hugetlbfs none /dev/hugepages
echo 512 > /proc/sys/vm/nr_hugepages ; for dpaa1 change change size as 256
```

NOTE

For the below commands, it is assumed that they are executed from `/usr/share` folder. Modify the commands for different path or `PATH` variable configuration.

- Set up the devices using DPDK scripts:

```
/usr/share/usertools/dpdk-devbind.py --status
/usr/share/usertools/dpdk-devbind.py -b uio_pci_generic 0000:00:03.0
/usr/share/usertools/dpdk-devbind.py -b uio_pci_generic 0000:00:04.0
```

Run DPDK applications

NOTE

Using Core 0 for DPDK application can lead to non-deterministic behavior, including drop in performance. It is recommended that DPDK application core mask values avoid using Core 0.

NOTE

`l3fwd` cannot work in VM with Virtio interfaces as offload mode for IP protocol is not supported by the DPDK Virtio driver.

NOTE

VM virtio is only functionally enabled and the performance is not comparable to the performance that we get on host.

Executing `l2fwd` application:

```
bin/l2fwd -c 0x2 -n 1 -- -p 0x1 -q 1 -T 0
```

Executing `testpmd` application:

- For Tx only:

```
./bin/testpmd -c 0x3 -n 1 -- -i --nb-cores=1  --portmask=0x1 --nb-ports=1 --forward-mode=txonly
--disable-hw-vlan --port-topology=chained
```

- For Rx only:

```
./bin/testpmd -c 0x3 -n 1 -- -i --nb-cores=1  --portmask=0x1 --nb-ports=1 --forward-mode=rxonly
--disable-hw-vlan --port-topology=chained
```

9.1.5.6 Multi-queue VIRTIO support

To scale the performance against the number of VM cores, the VIRTIO devices must be configured with multiple queues. This section explains the steps required to set up multi-queue VIRTIO devices.

See **Generic Setup** of DPAA platform including configuration necessary for defining multiple queues before DPDK application is executed. No special setup is required for DPAA2 before DPDK application start. Further, see Configuring OVS for setting OVS-DPDK on the host. Steps defined below build upon the configurations and steps provided in these sections for multi-queue support.

QEMU commands for multi-queue vhost devices are different and are shown later in the section.

Additional steps for setup of OVS

Besides the steps mentioned in Configuring OVS, following changes are required to modify the number of supported queues in the virtual machine.

Run following commands after adding DPDK and vhost-user ports to the bridge:

```
./ovs-vsctl set Interface dpdk0 options:n_rxq=2
./ovs-vsctl set Interface dpdk1 options:n_rxq=2
./ovs-vsctl set Interface dpdk0 options:n_txq=2
./ovs-vsctl set Interface dpdk1 options:n_txq=2
./ovs-vsctl set Interface vhost-user1 options:n_rxq=2
./ovs-vsctl set Interface vhost-user2 options:n_rxq=2
./ovs-vsctl set Interface vhost-user1 options:n_txq=2
./ovs-vsctl set Interface vhost-user2 options:n_txq=2
```

Launch VM with multi-queue VHOST devices

Similar to the steps mentioned in Launch Virtual Machine, following steps are required to start the virtual machine. Changes are highlighted with **bold**:

NOTE

Command snippets shown below are valid for DPAA2 platform. Replace **dpaa2** with **dpaa** for equivalent command on DPAA platform.

```
export GUEST_CONSOLE_TELNET_PORT=4446
export VM_MEM=2048M     # For DPAA1 use VM_MEM=650M
export VM_CORES=2
```

```
export NUM_QUEUES=2
```

```
export ROOTFS_IMG=<VM_ROOTFS_IMG>
```

```
export VHOST1_PATH=/usr/local/var/run/openvswitch/vhost-user1
export VHOST2_PATH=/usr/local/var/run/openvswitch/vhost-user2
```

```
qemu-system-aarch64 -nographic -object memory-backend-file,id=mem,size=$VM_MEM,mem-path=/mnt/
hugepages,share=on -cpu host -machine type=virt -kernel /boot/Image -enable-kvm  -serial
tcp::$GUEST_CONSOLE_TELNET_PORT,server,telnet -append 'root=/dev/vda rw console=ttyAMA0,115200
rootwait earlyprintk' -m $VM_MEM -numa node,memdev=mem -chardev socket,id=char1,path=$VHOST1_PATH
-netdev type=vhost-user,id=hostnet1,chardev=char1,vhostforce,queues=$NUM_QUEUES -device
virtio-net-pci,disable-modern=false,addr=0x3,netdev=hostnet1,mq=on,id=net1,mrg_rxbuf=off,vectors=6
-chardev socket,id=char2,path=$VHOST2_PATH -netdev type=vhost-
user,id=hostnet2,chardev=char2,vhostforce,queues=$NUM_QUEUES -device virtio-net-pci,disable-
modern=false,addr=0x4,netdev=hostnet2,mq=on,id=net2,mrg_rxbuf=off,vectors=6 -smp $VM_CORES -S -drive
if=none,file=$ROOTFS_IMG,id=foo,format=raw -device virtio-blk-device,drive=foo
```

DPDK applications in VM

Connect to VM terminal as explained in Accessing virtual machine console. Once logged-in as Guest, DPDK applications using multiple queues can be run in VM.

NOTE

If the number of queues defined for DPDK application in VM is *not* equal to number of queues (`NUM_QUEUES`) defined in QEMU command, the application may fail to start.

NOTE

For the below commands, it is assumed that they are executed from `/usr/share` folder. Modify the commands for different path or `PATH` variable configuration.

Besides the above steps, all steps are same as described in single queue VM usecase.

Set up the devices using DPDK scripts:

```
/usr/share/usertools/dpdk-devbind.py --status
```

```
/usr/share/usertools/dpdk-devbind.py -b uio_pci_generic 0000:00:03.0
```

```
/usr/share/usertools/dpdk-devbind.py -b uio_pci_generic 0000:00:04.0
```

Execute `l3fwd` application:

```
./examples/l3fwd  -c 0x3 -n 1 --  -p 0x3 --config="(0,0,0),(0,1,0),(1,0,1),(1,1,1)"  -P --parse-ptype
```

Execute `testpmd` application:

```
./bin/testpmd -c 3 -n 1 -- -i --nb-cores=1 --nb-ports=1 --total-num-mbufs=1025 --forward-mode=txonly
--disable-hw-vlan --rxq=2 --txq=2 --port-topology=chained
```

9.1.5.6.1 Accessing virtual machine console

Telnet to the `IP_ADDR_BRD` at port `GUEST_CONSOLE_PORT` from any machine, which can reach `IP_ADDR_BRD` over network:

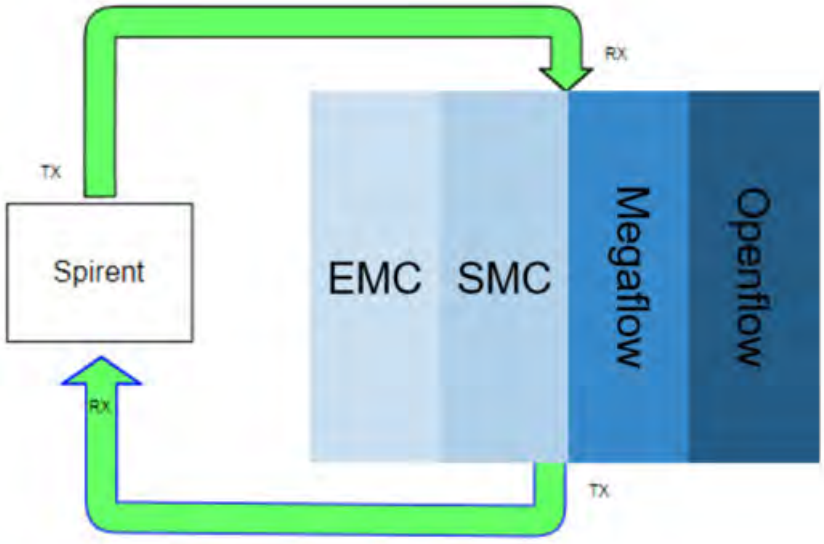
```
telnet  192.168.1.141 4446
Trying 192.168.1.141...
Connected to 192.168.1.141.
Escape character is '^]'.
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version 4.4.65 (root@dash1) (gcc version 5.4.0 20160609 (Ubuntu/Linaro
5.4.0-6ubuntu1~16.04.4) ) #1 SMP PREEMPT Fri Jun 23 07:34:43 IST 2017
```

Only a partial terminal output has been shown above.

9.1.5.7 OVS DPDK Performance Guide

OVS has a hierarchy of lookups. All the flows are initially added into the Openflow database (openflow block shown in below figure). When a flow is received, its entries get populated into EMC/SMC/Megaflow.

OVS Flow Table hierarchy



The exact-match cache (EMC) is the first and fastest mechanism Open vSwitch* (OVS) uses to determine what to do with an incoming packet. If the action for the packet cannot be found in the EMC, the search continues in the SMC cache followed by Megaflow classifier, and failing that the OpenFlow* flow tables are consulted. This can be thought of as similar to how a CPU checks increasingly slower levels of cache when accessing data.

By default EMC cache is enabled and SMC cache is disabled and both of them can be enabled or disabled via command-line only.

EMC cache can support up to max of 8K flows at a time, whereas SMC cache can support up to 100K entries.

Our recommendation w.r.t. flows for performance of OVS host cases:

- Use 256 flows for scenarios with 4 cores or less than 4 cores
- Use 2K flows for scenarios with more than 4 cores
- In case flows are more than 8K, disable EMC cache and enable SMC cache

To disable EMC cache and enable SMC cache:

```
ovs-vsctl --no-wait set Open_vSwitch . other_config:emc-insert-inv-prob=0
ovs-vsctl --no-wait set Open_vSwitch . other_config:smc-enable=true
```

To enable EMC cache and disable SMC cache:

```
ovs-vsctl --no-wait set Open_vSwitch . other_config:emc-insert-inv-prob=1
ovs-vsctl --no-wait set Open_vSwitch . other_config:smc-enable=false
```

It is also recommended to use per core memory pool using the below command:

```
ovs-vsctl set Open_vSwitch . other_config:per-port-memory=true
```

9.1.6 DPDK on Docker

9.1.6.1 Docker Overview

Docker provides an environment for a given image, over which any user space application can be executed. An image must contain/expose all the tools which are required to run any application.

For more information on Docker, see https://docs.docker.com/engine/userguide/.

9.1.6.2 DPAA1-Platform

9.1.6.2.1 Running Docker Container on DPAA1

To execute Docker, make sure you have completed the following prerequisites:

1. The Docker daemon must be running. If not, follow the instructions given at the link below to execute the daemon.

   https://docs.docker.com/engine/docker-overview/

2. The Docker tool must be installed, which works as the client to run the Docker container.

Download the required image, which should be run as an environment. Use the command below to get generic prebuilt images:

```
docker pull ubuntu:latest  # Command template is 'docker pull <distribution>:<tag>'
```

All downloaded images can be verified using the command below:

```
docker images
```

Once images are downloaded, the Docker container can be started using the steps below. Below commands execute a docker container, named **docker0**:

```
docker run --privileged --interactive --env LD_LIBRARY_PATH=/usr/local/lib --name=docker0 --
hostname=docker0 --detach --volume=/usr:/usr --volume=/sys:/sys --volume=/dev:/dev ubuntu:latest
```

Arguments provided to the command above have been explained below:

```
--privileged  # It provides privilege to docker container to access host completely
--interactive  # Docker container will be running state
--env LD_LIBRARY_PATH=/usr/local/lib  # Exporting host environment variable to docker container*/
--name=docker0 --hostname=docker0 # User defined name to docker container
--detach # container will be detached once it is launched and host prompt will be available for use
--volume=/XXX:/YYY # Exporting host partitions /XXX to docker container's mount point /YYY
```

Finally, following command attaches to the docker console which was run in previous command:

```
docker exec -it docker0 bash
```

9.1.6.2.2 Running the DPDK Application

Once Docker is launched and connected, then execute the DPDK application by running the respective command. The command below is a sample to run DPDK `l3fwd`:

```
export DPAA_FMCLESS_MODE=1
l3fwd -c 0x0C -n 1 - -p 0x30 --config="(4,0,2),(5,0,3)" -P
```

9.1.6.3 DPAA2-Platform

9.1.6.3.1 Traffic Multiplexer/De-Multiplexer

On the DPAA2 architecture, the MC provides various methods by which incoming traffic can be split of over the multiple DPNIs. The sections below provide more information.

Using DPDMUX

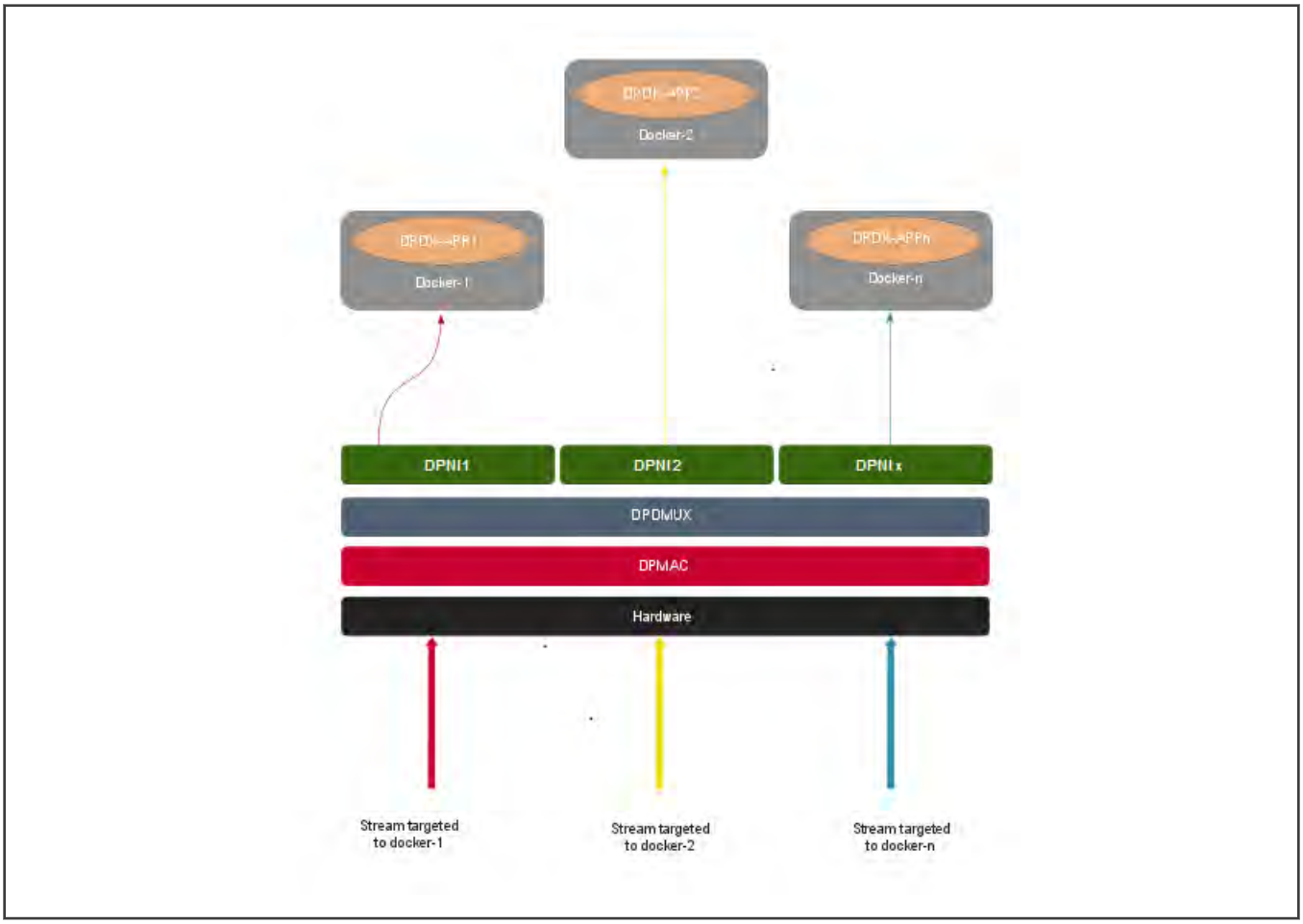
MC provides an object (DPDMUX) which splits incoming traffic over the multiple DPNIs based on following parameters:

1. MAC based classification

2. VLAN based classification
3. MAC + VLAN based classification
4. User defined key based classification.

DPDMUX has its own filter table which consists of default filtering rules. Default filtering rules are a combination of MAC address configured on DPNI and port information as a destination. Once the DPDMUX object is connected to a given DPNI, then the entry for a particular DPNI will be added to the filtering table. All incoming default traffic will be distributed based on the destination MAC address in the packet. You may add more entries to the filtering table as per your requirement.

The diagram below shows a sample use case for DPDMUX and associated links for a single DPMAC object. It can be extended up-to a maximum number of DPMACs, each having its own DPDMUX object.

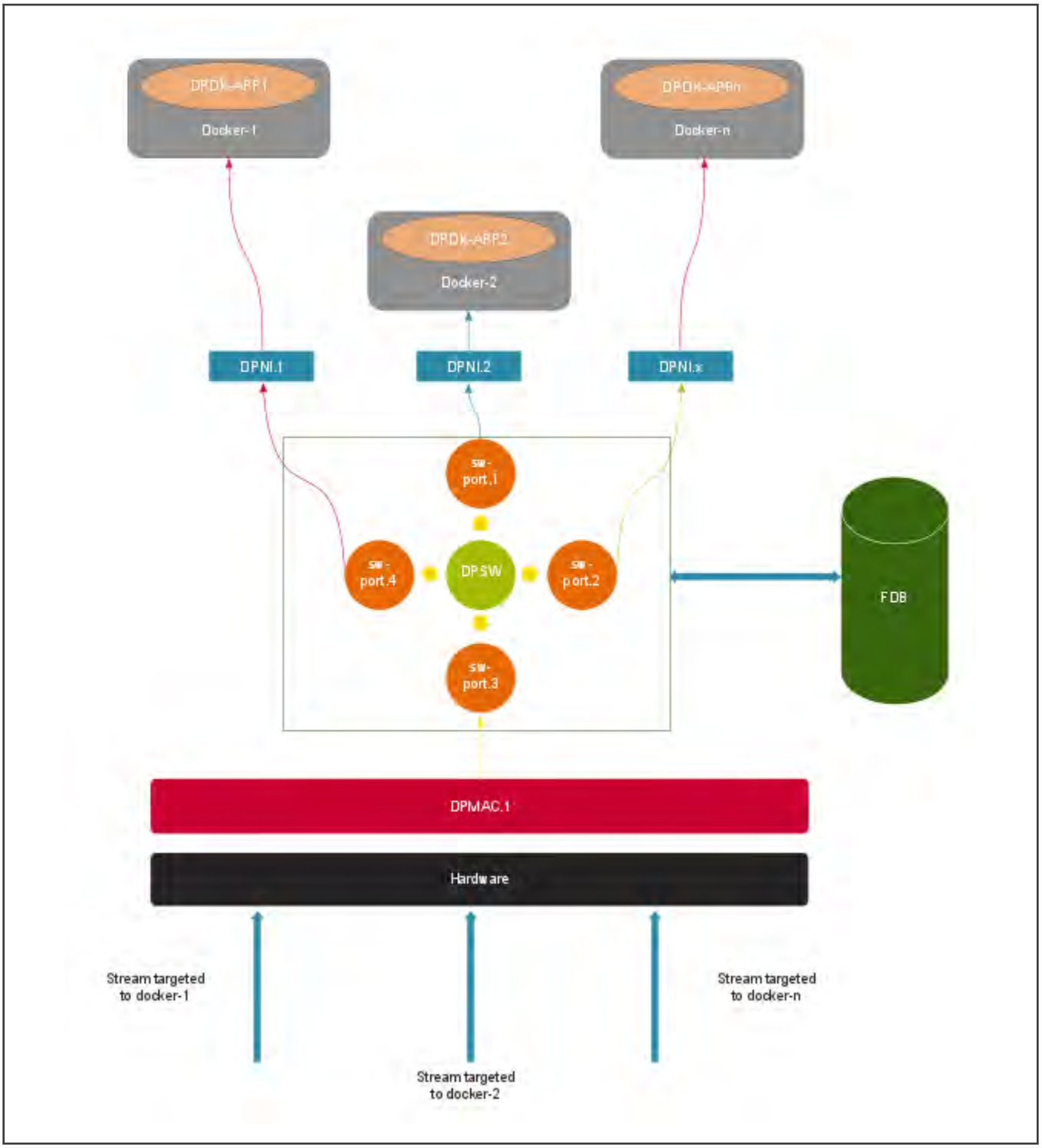


Using DPSW

MC also provides another object (DPSW) which internally implements DPAA2 H/W Switch. This Switch instance can also be used for traffic forwarding to multiple hosts. On LS2088, there is only once instance of DPSW that can be created and required ports will be connected to the same DPSW instance.

DPSW has its own filter table which populates dynamically with source MAC address and port on which packet is received. Default incoming traffic is flooded to all ports except ingress port and filtering rules are learned into filtering table. After learning, same packets are forwarded to the destined port only.

Below diagram shows a sample use case for DPSW and associated links for single DPMAC object. It can be extended up-to maximum number of DPMACs with same DPSW instance.



9.1.6.3.2 Single Docker Instance - Container Configuration (DPDMUX/DPSW)

For each Docker instance, a data path resource container (DPRC) must be created containing DPAA2 hardware blocks necessary for the Docker container.

NOTE

For more information on DPRCs, see "Data Path Resource Containers" section in "QorIQ networking technologies" chapter of *Layerscape Software Development Kit User Guide* available at the following link:

https://www.nxp.com/design/software/embedded-software/linux-software-and-development-tools/layerscape-software-development-kit:LAYERSCAPE-SDK?tab=Documentation_Tab

A helper script `dynamic_dpl.sh`, part of the LSDK rootfs, can be used for creating such DPRC. For example, following command snippet creates a DPRC containing 8 DPNI objects (logical network interfaces) which are not backed by any physical link (DPMAC) and have MAC addresses starting from `00:00:00:00:05:00`. For more details about creating DPRC, see "Creating DPRCs" subsection of "DPAA2 Quick Start Guide" section of *Layerscape Software Development Kit User Guide*.

Set the following environment variable which would be used by the `dynamic_dpl.sh` script:

```
export MAX_QOS=16
export DPNI_NORMAL_BUF=1 # This is optional
```

Execute the `dynamic_dpl.sh` script:

```
/usr/share/dpdk/dpaa2/dynamic_dpl.sh dpni dpni dpni dpni dpni dpni dpni dpni -b 00:00:00:00:05:00
```

The output of the above command would be similar to:

```
##################### Container  dprc.2  is created ####################

Container dprc.2 have following resources :=>

 * 16 DPBP
 * 8 DPCON
 * 8 DPSECI
 * 8 DPNI
 * 10 DPIO
 * 2 DPCI

 ######################### Configured Interfaces #########################

Interface Name        Endpoint             Mac Address
==============        ========             ==================
dpni.1                UNCONNECTED          00:00:00:00:05:01
dpni.2                UNCONNECTED          00:00:00:00:05:02
dpni.3                UNCONNECTED          00:00:00:00:05:03
dpni.4                UNCONNECTED          00:00:00:00:05:04
dpni.5                UNCONNECTED          00:00:00:00:05:05
dpni.6                UNCONNECTED          00:00:00:00:05:06
dpni.7                UNCONNECTED          00:00:00:00:05:07
dpni.8                UNCONNECTED          00:00:00:00:05:08
```

Each such DPRC is assigned to a Docker container. Therefore, multiple such DPRC would have to be created as per the use case and Docker instances required for it.

NOTE

A DPAA2 system has limited resources and assigning them to DPRC may result in error if requested resources are not available. For the above script output, if the script does not return any error and all the DPNIs have different MAC addresses, then the result can be considered a success. If an error or failure occurs during MAC address assignment, then resource assignment to the DPRCs must be restructured.

After this, based on whether DPDMUX or DPSW is used, one of the following configurations is applicable:

Configuration using DPDMUX

Create DPDMUX objects with total number of required links, that is, downlinks and uplinks both. Here, dpdmux.0 object is created:

```
restool dpdmux create --num-ifs=3 --method DPDMUX_METHOD_MAC --max-dmat-entries=8 --max-mc-groups=8 --
manip=DPDMUX_MANIP_NONE
```

Connecting downlinks and uplinks with above created DPDMUX:

```
restool dprc connect dprc.1 --endpoint1=dpmac.x --endpoint2=dpdmux.0.0
restool dprc connect dprc.1 --endpoint1=dpni.y --endpoint2=dpdmux.0.1
restool dprc connect dprc.1 --endpoint1=dpni.z --endpoint2=dpdmux.0.2
```

where, `x`, `y`, and `z` are object indexes created in resource containers.

Configuration using DPSW

Create DPSW object with total number of required links, that is, downlinks and uplinks both. Here, dpsw.0 object is created:

```
restool dpsw create --num-ifs=3
```

Connecting downlinks and uplinks with above created DPSW:

```
restool dprc connect dprc.1 --endpoint1=dpmac.x --endpoint2=dpsw.0.0
restool dprc connect dprc.1 --endpoint1=dpni.y --endpoint2=dpsw.0.1
restool dprc connect dprc.1 --endpoint1=dpni.z --endpoint2=dpsw.0.2
```

where, `x`, `y`, and `z` are object indexes created in resource containers.

9.1.6.3.3 Running Docker Container on DPAA2

Based on the explanation provided in the Running Docker Container on DPAA1, the command would be:

```
docker pull ubuntu:latest
export DPRC="dprc.<index>"
export VFIO_NO=`readlink /sys/bus/fsl-mc/devices/$DPRC/iommu_group | xargs basename`
docker run --privileged --interactive --env DPRC=$DPRC --device=/dev/vfio/vfio:/dev/vfio/vfio --
device=/dev/vfio/$VFIO_NO:/dev/vfio/$VFIO_NO --name=docker0 --hostname=docker0 --detach --volume=/
usr:/usr --volume=/sys:/sys --volume=/dev:/dev ubuntu:latest
docker exec -it docker0 bash
```

In the above, following is the explanation for arguments not applicable for DPAA:

```
export DPRC="dprc.<index>" # Where <index> is the DPRC container number created by
dynamic_dpl.sh execution
```

```
--device=/XXX:/YYY # Exporting host device /XXX to docker container device /YYY
```

9.1.6.3.4 Running the DPDK Application

Once Docker is launched and connected, then execute the DPDK application by running the respective command. The command below is a sample to run DPDK `l3fwd`:

```
l3fwd -c 0xFF -n 4 -- -p 0xFF -P --config="(0,0,0),(1,0,1),(2,0,2),(3,0,3),(4,0,4),(5,0,5),(6,0,6),
(7,0,7)" -P
```

9.1.6.3.5 Example Configuration for 2 Docker Instances: Using DPDMUX

Common Container settings:

```
export MAX_QOS=8
export DPNI_NORMAL_BUF=1
```

Create container for docker0:

```
./dynamic_dpl.sh dpni dpni dpni dpni dpni dpni dpni dpni -b 00:00:00:00:05:00

##################### Container  dprc.2  is created ####################

Container dprc.2 have following resources :=>
 * 16 DPBP
 * 8 DPCON
 * 8 DPSECI
 * 8 DPNI
 * 10 DPIO
 * 2 DPCI
######################### Configured Interfaces #########################

Interface Name        Endpoint              Mac Address
==============        ========              ==================
dpni.1                UNCONNECTED           00:00:00:00:05:01
dpni.2                UNCONNECTED           00:00:00:00:05:02
dpni.3                UNCONNECTED           00:00:00:00:05:03
dpni.4                UNCONNECTED           00:00:00:00:05:04
dpni.5                UNCONNECTED           00:00:00:00:05:05
dpni.6                UNCONNECTED           00:00:00:00:05:06
dpni.7                UNCONNECTED           00:00:00:00:05:07
dpni.8                UNCONNECTED           00:00:00:00:05:08
```

Create container for docker1:

```
./dynamic_dpl.sh dpni dpni dpni dpni dpni dpni dpni dpni -b 00:00:00:00:05:08

##################### Container  dprc.3  is created ####################

Container dprc.3 have following resources :=>
 * 16 DPBP
 * 8 DPCON
 * 8 DPSECI
 * 8 DPNI
 * 10 DPIO
 * 2 DPCI
######################### Configured Interfaces #########################
Interface Name        Endpoint              Mac Address
==============        ========              ==================
dpni.9                UNCONNECTED           00:00:00:00:05:09
dpni.10               UNCONNECTED           00:00:00:00:05:0a
dpni.11               UNCONNECTED           00:00:00:00:05:0b
dpni.12               UNCONNECTED           00:00:00:00:05:0c
dpni.13               UNCONNECTED           00:00:00:00:05:0d
dpni.14               UNCONNECTED           00:00:00:00:05:0e
dpni.15               UNCONNECTED           00:00:00:00:05:0f
dpni.16               UNCONNECTED           00:00:00:00:05:10
```

Create DPDMUX objects with downlinks and uplinks

```
restool dpdmux create --num-ifs=2 --method DPDMUX_METHOD_MAC --max-dmat-entries=8 --max-mc-groups=8 --
manip=DPDMUX_MANIP_NONE
restool dpdmux create --num-ifs=2 --method DPDMUX_METHOD_MAC --max-dmat-entries=8 --max-mc-groups=8 --
manip=DPDMUX_MANIP_NONE
restool dpdmux create --num-ifs=2 --method DPDMUX_METHOD_MAC --max-dmat-entries=8 --max-mc-groups=8 --
manip=DPDMUX_MANIP_NONE
restool dpdmux create --num-ifs=2 --method DPDMUX_METHOD_MAC --max-dmat-entries=8 --max-mc-groups=8 --
manip=DPDMUX_MANIP_NONE
```

Create uplink connections

```
restool dprc connect dprc.1 --endpoint1=dpdmux.0.0 --endpoint2=dpmac.1
restool dprc connect dprc.1 --endpoint1=dpdmux.1.0 --endpoint2=dpmac.2
restool dprc connect dprc.1 --endpoint1=dpdmux.2.0 --endpoint2=dpmac.3
restool dprc connect dprc.1 --endpoint1=dpdmux.3.0 --endpoint2=dpmac.4
```

Create downlink connections for docker0

```
restool dprc connect dprc.1 --endpoint1=dpni.1 --endpoint2=dpdmux.0.1
restool dprc connect dprc.1 --endpoint1=dpni.2 --endpoint2=dpdmux.1.1
restool dprc connect dprc.1 --endpoint1=dpni.3 --endpoint2=dpdmux.2.1
restool dprc connect dprc.1 --endpoint1=dpni.4 --endpoint2=dpdmux.3.1
```

Create downlink connections for docker1

```
restool dprc connect dprc.1 --endpoint1=dpni.5 --endpoint2=dpdmux.0.2
restool dprc connect dprc.1 --endpoint1=dpni.6 --endpoint2=dpdmux.1.2
restool dprc connect dprc.1 --endpoint1=dpni.7 --endpoint2=dpdmux.2.2
restool dprc connect dprc.1 --endpoint1=dpni.8 --endpoint2=dpdmux.3.2
```

NOTE

The above commands are for 1G test. In case 10G port is to be used append the above commands to create uplink and downlink with `--committed-rate=10000 --max-rate=10000`.

Running DPDK L2fwd on docker0

```
export DPRC="dprc.2"
export VFIO_NO=`readlink /sys/bus/fsl-mc/devices/$DPRC/iommu_group | xargs basename`
docker run --privileged --interactive --env DPRC=$DPRC --env LD_LIBRARY_PATH=/usr/local/lib --
device=/dev/vfio/vfio:/dev/vfio/vfio --device=/dev/vfio/$VFIO_NO:/dev/vfio/$VFIO_NO --name=docker0 --
hostname=docker0 --detach --volume=/usr:/usr --volume=/sys:/sys --volume=/dev:/dev ubuntu:latest
docker exec -it docker0 bash
l2fwd -c 0xF0 -n 1 --file-prefix=docker0 --socket-mem=2048 -- -p 0x0F -q 1
```

Running DPDK L2fwd on docker1

```
export DPRC="dprc.3"
export VFIO_NO=`readlink /sys/bus/fsl-mc/devices/$DPRC/iommu_group | xargs basename`
docker run --privileged --interactive --env DPRC=$DPRC --env LD_LIBRARY_PATH=/usr/local/lib --
device=/dev/vfio/vfio:/dev/vfio/vfio --device=/dev/vfio/$VFIO_NO:/dev/vfio/$VFIO_NO --name=docker1 --
hostname=docker1 --detach --volume=/usr:/usr --volume=/sys:/sys --volume=/dev:/dev ubuntu:latest
docker exec -it docker1 bash
l2fwd -c 0xF0 -n 1 --file-prefix=docker1 --socket-mem=2048 -- -p 0x0F -q 1
```

9.1.6.3.6 Example Configuration for 2 Docker Instances: Using DPSW

Common Container settings:

```
export MAX_QOS=8
export DPNI_NORMAL_BUF=1
```

Create container for docker0:

```
./dynamic_dpl.sh dpni -b 00:00:00:00:05:00

##################### Container  dprc.2  is created ####################

Container dprc.2 have following resources :=>
 * 16 DPBP
 * 8 DPCON
 * 8 DPSECI
 * 1 DPNI
 * 10 DPIO
 * 2 DPCI


######################### Configured Interfaces #########################

Interface Name        Endpoint              Mac Address
==============        ========              ==================
dpni.1                UNCONNECTED           00:00:00:00:05:01
```

Create container for docker1:

```
./dynamic_dpl.sh dpni -b 00:00:00:00:05:01

##################### Container  dprc.3  is created ####################

Container dprc.3 have following resources :=>

 * 16 DPBP
 * 8 DPCON
 * 8 DPSECI
 * 1 DPNI
 * 10 DPIO
 * 2 DPCI


######################### Configured Interfaces #########################

Interface Name        Endpoint              Mac Address
==============        ========              ==================
dpni.2                UNCONNECTED           00:00:00:00:05:02
```

Create DPSW objects

```
restool dpsw create --num-ifs=3
restool dprc connect dprc.1 --endpoint1=dpsw.0.0 --endpoint2=dpmac.1
```

Create downlink connections for docker0

```
restool dprc connect dprc.1 --endpoint1=dpni.1 --endpoint2=dpsw.0.1
```

Create downlink connections for docker1

```
restool dprc connect dprc.1 --endpoint1=dpni.2 --endpoint2=dpsw.0.2
```

Running DPDK L2fwd on docker0

```
export DPRC="dprc.2"
export VFIO_NO=`readlink /sys/bus/fsl-mc/devices/$DPRC/iommu_group | xargs basename`

docker run --privileged --interactive --env DPRC=$DPRC --env LD_LIBRARY_PATH=/usr/local/lib --
device=/dev/vfio/vfio:/dev/vfio/vfio --device=/dev/vfio/$VFIO_NO:/dev/vfio/$VFIO_NO --name=docker0 --
hostname=docker0 --detach --volume=/usr:/usr --volume=/sys:/sys --volume=/dev:/dev ubuntu:18.04

docker exec -it docker0 bash

cd /usr/share/dpdk/examples
./l2fwd -c 0x04 -n 1 --file-prefix=docker0 --socket-mem=2048 -- -p 0x01 -q 1
```

Running DPDK L2fwd on docker1

```
export DPRC="dprc.3"
export VFIO_NO=`readlink /sys/bus/fsl-mc/devices/$DPRC/iommu_group | xargs basename`

docker run --privileged --interactive --env DPRC=$DPRC --env LD_LIBRARY_PATH=/usr/local/lib --
device=/dev/vfio/vfio:/dev/vfio/vfio --device=/dev/vfio/$VFIO_NO:/dev/vfio/$VFIO_NO --name=docker1 --
hostname=docker1 --detach --volume=/usr:/usr --volume=/sys:/sys --volume=/dev:/dev ubuntu:18.04

docker exec -it docker1 bash

cd /usr/share/dpdk/examples
./l2fwd -c 0x08 -n 1 --file-prefix=docker1 --socket-mem=2048 -- -p 0x01 -q 1
```

9.1.7 Enabling DPAA2 direct assignment for DPDK

The DPAA2 architecture supports the assignment of direct DPAA2 resource access from the QEMU guest VM (Kernel or user space application). See **Direct assigned devices**.

This section describes necessary environment setup and commands for launching a Virtual Machine (VM) with VFIO device passthrough or direct device assignment support.

NOTE

Currently, Arm-V8 supports VM in NO-IOMMU mode only. It means, hardware access always happens using physical address mode. The default SDK code is build with virtual addressing mode only. You must rebuild the DPDK for `arm64-dpaa-linuxapp-gcc`, by manually setting `CONFIG_RTE_LIBRTE_DPAA2_USE_PHYS_IOVA=y` in `config/defconfig_arm64-dpaa-linuxapp-gcc`, and then build DPDK and example applications through standard compilation steps. `CONFIG_RTE_LIBRTE_DPAA2_USE_PHYS_IOVA=y` enables physical addressing mode (IOVA) which is required for direct assignment functionality.

Then, to build DPDK applications, follow the instructions in Standalone build of DPDK libraries and applications section.

You can transfer the applications manually to the virtual machine using the host-vm connections as suggested to configure in next section.

9.1.7.1 Launch Virtual Machine

1. Ensure that kernel is enabled for direct assignment mode (see "How to use DPAA2 direct assignment" in QEMU).
2. The default QEMU present in filesystem may not support the direct assignment feature (see QEMU).

To build QEMU 4.1 locally with VFIO passthrough support, execute the following commands:

```
git clone https://source.codeaurora.org/external/qoriq/qoriq-components/qemu
cd qemu
git checkout qemu-4.1
git submodule update --init dtc
```

Ensure that your machine has the required packages to build QEMU. Refer to the example below:

```
#update your ubuntu m/c with required packages.
apt-get install pkg-config
apt-get install libglib2.0-dev
apt-get install libpixman-1-dev
apt-get install libaio-dev
apt-get install libusb-1.0-0-dev
```

Now, build the QEMU:

```
./configure --prefix=/root/qemu-4.1 --target-list=aarch64-softmmu --enable-fdt --enable-kvm
make
make install
```

The new QEMU will be installed in /root/qemu-4.1 folder.

3. Create DPAA2 resources for the VM guest kernel and VM guest user space (DPDK) on the board.

   The dynamic scripts to support the dpaa2 resource creation are available in (/usr/share/dpdk/dpaa2) for LSDK rootfs. It is also part of the DPDK source code in "nxp" folder.

   ```
   export DPDK_SCRIPTS=/usr/share/dpdk/dpaa2
   ```

   Create a DPNI-based interface for file transfer and communication between host and VM guess kernel.

   ```
   ls-addni -n
   Output:---Created interface: eth0 (object:dpni.1, endpoint:)
   ```

   Next create the VM guest kernel container. See *How to use DPAA2 direct assignment* chapters for further information.

   A sample vm_linux conf file is provided in scripts to create the VM guest kernel container. In this case, the number of resources is good for 2 core VM. The previously created DPNI object is also passed to connect it with guest kernel container.

   ```
   source $DPDK_SCRIPTS/dynamic_dpl.sh -c $DPDK_SCRIPTS/vm_linux.conf <dpni.x>
       Where, <dpni.x> is DPNI interface created by "ls-addni" command.
   ```

   Next step is to create the container for VM guest user space for DPDK

   ```
   source $DPDK_SCRIPTS/dynamic_dpl.sh -c $DPDK_SCRIPTS/vm_dpdk.conf <dpmac.x> <dpmac.y>
       Where, <dpmac.x> & <dpmac.y> are required MAC interfaces.
   ```

   NOTE

   Ensure to enter the created parent DPRC into vm_dpdk.conf.

   For the rest of the chapter, it is assumed that VM guest kernel container is dprc.2 and VM guest user space child container is dprc.3 (nested).

Create an Ethernet connection between host and VM for communication/transfer. A connection was already created and passed during vm-linux container.

```
#assign IP to host interface created to communicate with VM (dprc.2, eth0)
ifconfig eth0 192.168.2.2
```

4. Create hugepages mount:

```
echo hugetlbfs /mnt/hugetlbfs hugetlbfs defaults,mode=0777 0 0 >> /etc/fstab
mkdir /mnt/hugetlbfs
mount /mnt/hugetlbfs
```

5. Launch QEMU (Version: 4.1.0) using the following command:

```
export ROOTFS_IMG=/ubuntu_bionic_arm64_rootfs.ext4.img # Telnet port to be used for accessing
this instance of virtual machine export GUEST_CONSOLE_TELNET_PORT=4446
export KERNEL_IMG=/root/Image-4.14 export KERNEL_IMG=/root/Image-4.14
```

NOTE

For generating a root filesystem image, see "Creating a guest Linux root filesystem" section in "Virtualization " chapter of *Layerscape Software Development Kit User Guide* available at the following link:

https://www.nxp.com/design/software/embedded-software/linux-software-and-development-tools/layerscape-software-development-kit:LAYERSCAPE-SDK?tab=Documentation_Tab

Assign the `ROOTFS_IMG` in above command with the absolute path to the generated image.

Define other environment variables which are used by the QEMU command to configure the virtual machine environment:

```
export VM_MEM=4096M(2048M for LS1088ARDB)
```

1. Add the device command below (for the GUEST KERNEL DPRC to be assigned) to the QEMU command line:

```
-device vfio-fsl-mc,host=dprc.2
```

Also, make sure to specify the appropriate number of cores for the guest VM. It should match the number of dpio objects created in the child container. In our case, 1 core.

```
-smp $VM_CORES
```

2. Start QEMU with -S option (the vcpu threads are not yet started) to make the Ethernet drivers in the guest bind the objects to the cores correctly.

```
# single core VM launch /root/qemu-4.1/bin/qemu-system-aarch64 -smp $VM_CORES -m
$VM_MEM -mem-path /mnt/hugetlbfs -cpu host -machine type=virt,gic-version=3 -kernel
$KERNEL_IMG -enable-kvm -display none -serial tcp::$GUEST_CONSOLE_TELNET_PORT,server,telnet
-drive if=none,file=$ROOTFS_IMG,id=foo,format=raw -device virtio-blk-device,drive=foo -append
'root=/dev/vda rw console=ttyAMA0 rootwait earlyprintk' -monitor stdio -device vfio-fsl-
mc,host=dprc.2 -S
```

Two core VM launch (check the isolcpus for core #1 in bootargs).

NOTE

- For best performance, Core 0 in the VM should not be used for DPDK I/O threads.
- To prevent system services from using GPUs scheduled for DPDK I/O threads, it is recommended that isolcpus is used for isolating cores from Linux Kernel scheduling in VM. The exact configuration is dependent on number of CPU assigned by QEMU to VM using the VM_CORES environment variable.

Append isolcpus=1-$VM_CORES to the 'root=/dev/vda rw console=ttyAMA0,115200 rootwait earlyprintk' string in the qemu-system-aarch64 command given above.

```
/root/qemu-4.1/bin/qemu-system-aarch64 -smp $VM_CORES -m $VM_MEM -mem-path /mnt/hugetlbfs
-cpu host -machine type=virt,gic-version=3 -kernel $KERNEL_IMG -enable-kvm -display none
-serial tcp::$GUEST_CONSOLE_TELNET_PORT,server,telnet -drive
if=none,file=$ROOTFS_IMG,id=foo,format=raw -device virtio-blk-device,drive=foo -append
'root=/dev/vda rw console=ttyAMA0 rootwait earlyprintk isolcpus=1' -monitor stdio
-device vfio-fsl-mc,host=dprc.2 -S
```

NOTE

Make sure to specify the appropriate number of cores for the guest VM. It should match the number of dpio objects created in. Also, make sure that the /mnt/hugetlbfs folder exists and is mounted when starting QEMU.

Following log appears on the host UART console:

```
QEMU 4.1.0 monitor - type 'help' for more information
(qemu) qemu-system-aarch64: -serial tcp::4446,server,telnet: QEMU waiting for connection
on: disconnected:telnet::4446,server
```

3. Launch VM using: telnet <Board ip addr> <GUEST_CONSOLE_TELNET_PORT> For example, telnet localhost 4446

   Ensure to assign each vcpu thread to one physical CPU only.

   Get the VM thread IDs entering QEMU shell.

```
(qemu) info cpus
* CPU #0: thread_id=7211
CPU #1: (halted) thread_id=7212
```

   Assign one vcpu thread to one core only. Also, apart from the first vcpu thread put all other threads in chrt priority for performance.

```
$ taskset -p 0x1 7211
pid 7211's current affinity mask: ff
pid 7211's new affinity mask: 1
$ taskset -p 0x2 7212
pid 7212's current affinity mask: ff
pid 7212's new affinity mask: 2
$ chrt -p 90 7212
```

   start the vcpu threads:

```
(qemu) c
```

9.1.7.2 Accessing the virtual machine console

```
root@ls2088ardb:~# telnet localhost 4446
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 4.14.16-00004-ga5a4b5d (b10814@bf-netperf1.idc) (gcc version 7.2.1
20171011 (Linaro GCC 7.2-2017.11)) #2 SMP PREEMPT Tue Apr 3 12:24:09 IST 2018
[    0.000000] Boot CPU: AArch64 Processor [410fd082]
[    0.000000] Machine model: linux,dummy-virt
[    0.000000] efi: Getting EFI parameters from FDT:
```

```
[    0.000000] efi: UEFI not found.
[    0.000000] cma: Reserved 16 MiB at 0x00000000ff000000
[    0.000000] NUMA: No NUMA configuration found
----\
Ubuntu 16.04.3 LTS localhost ttyAMA0
localhost login: root
Password:
Last login: Wed May  2 20:08:32 UTC 2018 on ttyAMA0
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.14.16-00004-ga5a4b5d aarch64)
```

Only a partial terminal output has been shown above.

Use "root" as login and password.

Execute the following commands:

```
echo 1000 > /proc/sys/vm/nr_hugepages
# child DPRC container for VM guest userspace
export DPRC=dprc.3
echo 1 > /sys/module/vfio/parameters/enable_unsafe_noiommu_mode
echo vfio-fsl-mc > /sys/bus/fsl-mc/devices/$DPRC/driver_override
echo $DPRC > /sys/bus/fsl-mc/drivers/vfio-fsl-mc/bind
```

The Host *core-index* which is used as First Physical core in VM should be used as *DPAA2_HOST_START_CPU* to run DPDK application. Where, *core-index* range is [0-7] for a 8 core platform.

For example, if you are running VM with two cores and Host core #4 and core #5 are given to VM, the first physical core for VM is core#4. So, we should set the Start CPU core as follows:

```
export DPAA2_HOST_START_CPU=4
```

Setup Hugepages

```
echo hugetlbfs /dev/hugepages hugetlbfs defaults,mode=0777 0 0 >> /etc/fstab
mkdir /dev/hugepages
mount /dev/hugepages
```

Configure the host connection for SCP, ssh, and file transfer.

```
ifconfig eth1 192.168.2.1
```

9.1.7.3 Running DPDK applications with direct device assignments

All the DPAA2-based DPDK applications work in VM similar to the host.

If the DPDK example applications are not present, you can bring them via scp/tftp using eth1 interface.

> **NOTE**
>
> Using Core 0 for DPDK application can lead to non-deterministic behavior, including drop in performance. It is recommended that DPDK application core mask values avoid using Core 0.

Some example test commands are given below:

```
#one core VM (core #0 for DPDK)
./l3fwd -c 0x1 -n 1 --log-level=bus.fslmc,8 -- -p 0x1 -P --config="(0,0,0)"
./l2fwd-crypto -c 0x1 -n 1 --log-level=bus.fslmc,8 -- -p 0x1 -q 1 --chain HASH_ONLY
--auth_algo sha2-256-hmac --auth_op GENERATE --auth_key_random_size 64
```

#two core VM (core #1 for DPDK)

```
./l3fwd -c 0x2 -n 1 -- -p 0x1 -P --config="(0,0,1)"
./l3fwd -c 0x2 -n 1 -- -p 0x3 -P --config="(0,0,1),(1,0,1)"
./testpmd -c 0x3 -n 1 -- -i --portmask=0x3 --nb-cores=1 --forward-mode=txonly
```

9.1.8 Known Limitations and Future Work

Generic Limitations:

1. Currently, only DPDK EAL framework threads are supported for packet I/O. The threads created by other methods (for example, pthread_create) may not work. The DPAAx hardware uses the hardware access portals internally for each thread doing packet I/O. Currently, this information is automatically assigned in thread local variable, when using DPDK thread framework.

   Additionally, the number of maximum I/O threads using DPDK are limited to the number of hardware portals available. The number of available portals is platform-dependent.
2. Not all features of DPDK framework are yet implemented through PPFE, ENETC, DPAA, and DPAA2 drivers (PMDs). For the list of supported features, refer to PPFE supported DPDK features, ENETC supported DPDK features, DPAA: Supported DPDK Features, and DPAA2: Supported DPDK Features.
3. Using Core 0 for I/O related work is known to impact performance - whether on host or in VM. Disabling services or RT prioritization can result in optimal performance but the results are non-deterministic. Affining Core 0 to I/O should be avoided as much as possible.
4. It has been observed that PCI NIC card events can lead to performance drop on certain platforms. The behavior is non-deterministic across platforms. For peak performance numbers, PCI NIC cards should be disabled.
5. DPDK docker support is only available for DPAA2 and DPAA platforms.
6. DPDK multiprocess mode (that is, using DPDK secondary processes) is not supported on DPAA1.
7. LS1088A platform has limited CTLU features. This limits the device hardware classification capabilities leading to reduced number of field combinations for flow matching/classification.

DPAA2-Specific Limitations:

1. IPsec Direct assignment performance does not scale beyond 4 cores.

DPAA-Specific Limitations:

1. Ports assigned to user space cannot be assigned dynamically to kernel space or vice versa.
2. Default configuration for DPAA platform is to expect execution of FMC tools (see manual) before application can be run. This adds a constraint on the number of queues the application can initialize which is exactly same as the number of queues the FMC tool has configured. If incorrect number of queues is used (less than configured by FMC tool), then RSS distribution can cause loss of packets or no I/O.
3. On the LS1043ARDB platform, performance may be lower for 6G setup as compared to 10G setup.

PPFE (LS1012A) Specific Limitations:

1. While using PPFE in user space, if the kernel mode PFE module is loaded before using the user space mode, then the HIF rings do not get cleaned sometimes and the user must restart the application again until the rings are cleaned.
2. Multiple buffer pools are not supported.
3. User-defined Rx/Tx queue configuration is not supported. Driver configures queue with default attributes only.

ENETC (LS1028A) Specific Limitations:

1. Link Negotiation and Link status update are not supported.

2. Switch port should be connected and link should be up before booting Linux.

9.1.9 Troubleshooting

Following are some common steps and suggestions outlined for best performance from DPDK applications:

1. To obtain best performance, ensure that the boot-up time command line arguments are similar to below:

   For DPAA2:

   ```
   console=ttyS1,115200 root=/dev/mmcblk0p3 earlycon=uart8250,mmio,0x21c0600
   default_hugepagesz=1024m hugepagesz=1024m hugepages=8 isolcpus=1-7
   iommu.passthrough=1 rcupdate.rcu_cpu_stall_suppress=1
   ```

   NOTE

   In the above, change the `isolcpus` as required based on the cores which would be used by DPDK applications.

   For DPAA:

   ```
   console=ttyS0,115200 root=/dev/mmcblk0p3 earlycon=uart8250,mmio,0x21c0500 default_hugepagesz=2m
   hugepagesz=2m hugepages=512 isolcpus=1-3 bportals=s0 qportals=s0
   iommu.passthrough=1 rcupdate.rcu_cpu_stall_suppress=1
   ```

   `isolcpus` in the above ensures that only Linux Kernel schedules its threads on Core 0 only. Core 1-x would be used for DPDK application threads.

   Hugepage count defined by `hugepages` should also be modified to maximum possible so as to allow DPDK applications to have larger buffers.

   NOTE

   The value of hugepages is dependent on the size of RAM available on the board. Value should be selected based on specific use case as any memory allocated for hugepage is not usable for Linux Kernel OS operations.

2. If there is an issue with reception of transmission of packets, verify the following points:
   a. Ensure that DPDK application reports no error at startup. If an error occurs, then the cause of error is usually described clearly in the output.
   b. Check the mapping of ports against the physical ports:
      - For DPAA platform, ensure that physical interfaces map correctly to the DPDK ports (see LS1043ARDB Port Layout or LS1046ARDB Port Layout).
      - For DPAA2 platform, ensure that correct `dpni.X` file has been used in the `dynamic_dpl.sh` script while creating the `dprc` containers. A common pitfall is to use an incorrect `dpni` as against the physical port being used for I/O.
   c. Ensure that traffic generator to board connectivity is proper. To validate if packets are going out on specific interfaces, you can run `testpmd` in `tx_only` mode. For information about `testpmd` application and its supported arguments, refer to the web documentation.
   d. Ensure that the traffic generator stream settings are correct and enough streams are being generated for proper distribution between DPDK application cores.
   e. Ensure that the MAC address of stream generated by traffic generator matches that of the `dpni` port, or the interface is in promiscuous mode.
3. If the performance is not as expected:
   a. Ensure that the stream configuration of the traffic generator is appropriate and that it can generate multiple streams. In case the streams have all same IP destination and/or source, the distribution of traffic across multiple cores would not happen.

NOTE

For obtaining best performance, it is important to configure the number of streams from packet generator adequately. If the number of streams generated by packet generator are not adequate, it would lead to improper distribution across the queues defined (especially for multi-queue setup) and eventually lack of performance.

b. Using standard process tools in Linux, for example `ps`, `top`, verify that all the DPDK application threads have been started (as per application configuration on command line) and busy looping.

c. For DPAA2, in case any DPAA2 ports are assigned to Linux kernel, assure that the interrupt affinity is not on any core which is assigned to DPDK. See the DPDK Performance Reproducibility Guide for details about how to check and affine cores to such interrupts.

4. For DPAA2 platform, certain tuning parameters are available which can be enabled, as required.

   - To offload the RX error packet drop (parsing error) handling in hardware, set:

     ```
     #export DPAA2_PARSE_ERR_DROP=1
     ```

   - To disable the TX congestion control, that is, infinite size of TX queues, set:

     ```
     #export DPAA2_TX_CGR_OFF=1
     ```

   - To configure the TQ queue congestion control - taildrop size in bytes (default is 64 KB), set:

     ```
     #export DPAA2_TX_TAILDROP_SIZE=<size>
     ```

5. System tuning parameters can be verified with "debug_dump.sh" script placed in "/usr/share/dpdk" directory. You can share the output with support team for further analysis.

6. DPAA2 port status can be checked from restool commands (for example, `restool dpni info dpni.1`).

7. DPAA2 - When using large number of buffers (> 1 Million), the application may hang. This is due to maximum limit of number of buffers configured by default for DPAA2 QBMAN. It is a configurable setting in DPC file. To configure number of buffers, following node must be added or modified with the correct number of buffers.

   ```
   {
       qbman  {
           ....
           total_bman_buffers=<number of buffers in HEX>;
        };
   };
   ```

9.1.10 DPDK Performance Reproducibility Guide

This chapter describes various cases and points that are important for obtaining best performance from DPDK software on NXP platforms. It includes some best practices and optimal configurations that can help extract maximum performance out of the NXP DPAA hardware.

NOTE

The practices mentioned in this chapter are based on tests in controlled environment. They are not intended for production or deployment without proper analysis of the impact on use cases.

This document is divided into two broad sections: Steps required before booting up the Linux Kernel and steps required before DPDK application execution.

Before booting up Linux

1. Use GCC toolchain 7.4.1 as recommended toolchain for compiling DPDK.
2. On kernel 5.4 and above, CONFIG_QORIQ_THERMAL flag is enabled by default which tracks temperature of the SOC. In some cases, such as LS1043A, the temperature may go up while running DPDK applications, which are CPU intensive

and the maximum CPU frequency is clamped to a lower value. It may result in lower performance numbers. To disable this feature, disable CONFIG_QORIQ_THERMAL flag while compiling kernel image.

3. **Choosing Optimal Board Support Packages (BSP)**
   - Choosing a compatible board support package is critical for functionality as well as performance of DPDK application.

     For DPAA and DPAA2 platforms, select the top frequency RCW/PBL binaries stably supported by boards. For example, for LS2088ARDB DPAA2, Rev 1.1 boards with frequency of **2100x800x2133** is known to perform best. Other frequency, though stable, would result in slower performance. Below table describes an indicative set of known BSP files for DPDK supported SoC.
4. **Disabling hardware prefetching through U-Boot**
   - For LS2088A DPAA2 platform, it is possible to disable hardware prefetching through U-Boot. It can enhance performance in multicore scenario.
   - For disabling hardware prefetching, following command should be used on U-Boot prompt:

     ```
     setenv hwconfig 'fsl_ddr:bank_intlv=auto;core_prefetch:disable=0xFE'
     ```

     NOTE

     Change the `disable=` parameter based on the platform being used. For example, for LS1046/LS1043, having 4 cores, use `disable=0xE`, and for LX2 having 16 cores, use `disable=0xFFFE`.

     After executing the above command, board bank must be reset for the setting to take place. In the above command, field `disable=0xFE` defines the mask for disabling prefetching on specific cores. For example, for disabling prefetching on 3rd and 4th core, use `disable=0x0C`.

     Also, it should be noted that disabling prefetching on Core 0 is not supported.

     NOTE

     This setting does not have impact on single core case. Maximum performance gain is observed when all 8 cores of LS2088 board are being used (of which 7 cores have prefetching disabled as Core 0 does not support this feature).
5. **Linux Boot Argument**
   - For DPAA platform, if the onboard memory is limited (for example, LS1043 RDB), following configuration should be appended to default boot arguments:

     ```
     default_hugepagesz=2m hugepagesz=2m hugepages=512 isolcpus=1-3 bportals=s0 qportals=s0
     iommu.passthrough=1 rcupdate.rcu_cpu_stall_suppress=1
     ```

     Through the above boot arguments, 1024 Mbit of hugepages have been assigned for all DPDK applications (512 pages of 2 M size each).

     `isolcpus` isolates the CPUs 1, 2, 3 from scheduling algorithm of Linux Kernel process schedulers. All System Service would be scheduled on Core 0 and that should be avoided in application configuration for I/O threads.

     `rcupdate.rcu_cpu_stall_suppress=1` is specifically for cases where Core 0 is also used for running DPDK I/O with `enable_performance_mode.sh` script - where because of Real-Time priority setting of the script, RCU stalls might be observed. That leads to screen dump which might impact performance.
   - For DPAA2 platform, following configuration should be appended to default boot arguments:

     ```
     default_hugepagesz=1024m hugepagesz=1024m hugepages=8 isolcpus=1-7
     iommu.passthrough=1 rcupdate.rcu_cpu_stall_suppress=1
     ```

     For DPAA2 platform, 1G huge page size is recommended.

`rcupdate.rcu_cpu_stall_suppress=1` is specifically for cases where Core 0 is also used for running DPDK I/O with `enable_performance_mode.sh` script - where because of Real-Time priority setting of the script, RCU stalls might be observed. That leads to screen dump which might impact performance.

NOTE

While running DPDK for all core cases, `isolcpus` parameter should not be set in bootargs. `enable_performance_mode.sh` script reserves 99.6% CPU for DPDK application and the rest is given to kernel on all cores, so `isolcpus` is not required. Ensure that links are up for all interfaces before running DPDK to avoid kernel tasks running slowly.

NOTE

Change the value of `isolcpus` parameter based on the platform being used. For example, for LX2 platform use `isolcpus=1-15`

- In case UEFI-based booting is used, the boot arguments are changed in the grub.cfg file. To update boot arguments for UEFI, see section 5.4 ("UEFI") of *Layerscape Software Development Kit User Guide* available at the following link:

  https://www.nxp.com/design/software/embedded-software/linux-software-and-development-tools/layerscape-software-development-kit:LAYERSCAPE-SDK?tab=Documentation_Tab

NOTE

Remember that CPU isolation configuration cannot be changed in a running Linux kernel whereas huge page configuration can be changed from Linux prompt by writing to `/proc/sys/vm/nr_hugepages` file. Therefore, CPU isolation should be carefully decided before booting up Linux kernel.

NOTE

`nousb` can be appended to boot arguments to disable USB in Linux Kernel. It prevents CPU cores from serving any interrupts from USB devices. It is especially important when Core 0 is being used for DPDK I/O performance. But, this option should only be used if there is no dependency of USB devices for system execution, for example, a USB mass storage which contains either the root filesystem or extra filesystem containing data necessary for execution.

6. For best performance, use the data cores as isolated CPUs and operate them in tickless mode on kernel version 4.4 above as follows:
   a. Compile the kernel with `CONFIG_NO_HZ_FULL=y`.
   b. Add bootargs with `'isolcpus=1-7 rcu_nocbs=1-7 nohz_full=1-7'` for 8 core platform and `'isolcpus=1-3 rcu_nocbs=1-3 nohz_full=1-3'` for 4 core platform.

NOTE

The CONFIG_NO_HZ_FULL Linux kernel build option is used to configure a tickless kernel. The idea is to configure certain processor cores to operate in tickless mode and these cores do not receive any periodic interrupts. These cores run dedicated tasks (and no other tasks can be scheduled on such cores, eliminating the need to send a scheduling tick). A CONFIG_HZ based timer interrupt invalidates L1 cache on the core which can degrade data plane performance by a few % points (to be quantified, but estimated to be 1-3%). Running tickless typically means getting 1 timer interrupt/sec instead of 1000/sec.

7. **Setup of the Performance Validation Environment**
   - It is important that the environment for performance verification uses a balanced core loading approach. Each core should be loaded with equal number of Rx/Tx queues, irrespective of their count. Images below describe some of the I/O scenario using an example setup containing a target board and a packet generator. In all the cases shown, it is assumed that each port has a single queue being serviced by a CPU core. Also, even though below images show 8 ports, it is a generic representation. DPAA boards may not have 8 equal ports (1G/10G) - this representation is assuming traffic is always distributed across equal capacity ports.

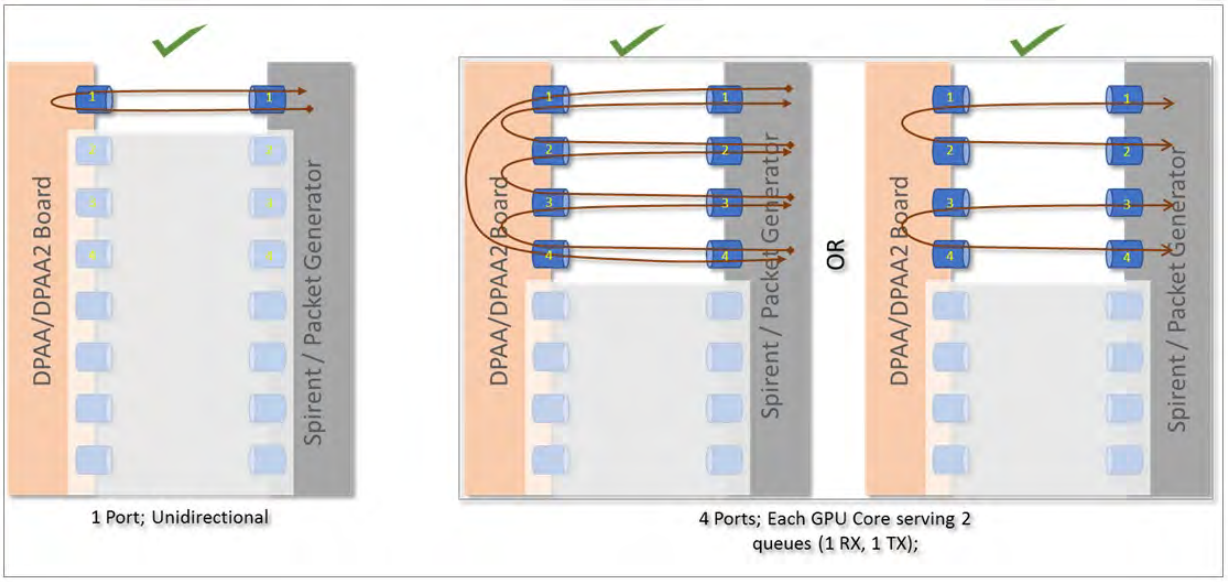


Image above describes 2 cases: One for single port and another for 4 ports. Note that all the cores are equally loaded (equal number of cores, irrespective numbers of ports being serviced). Further, the 4-port case shows that there is more than one way to move stream of packets (note the direction of arrows in each case).

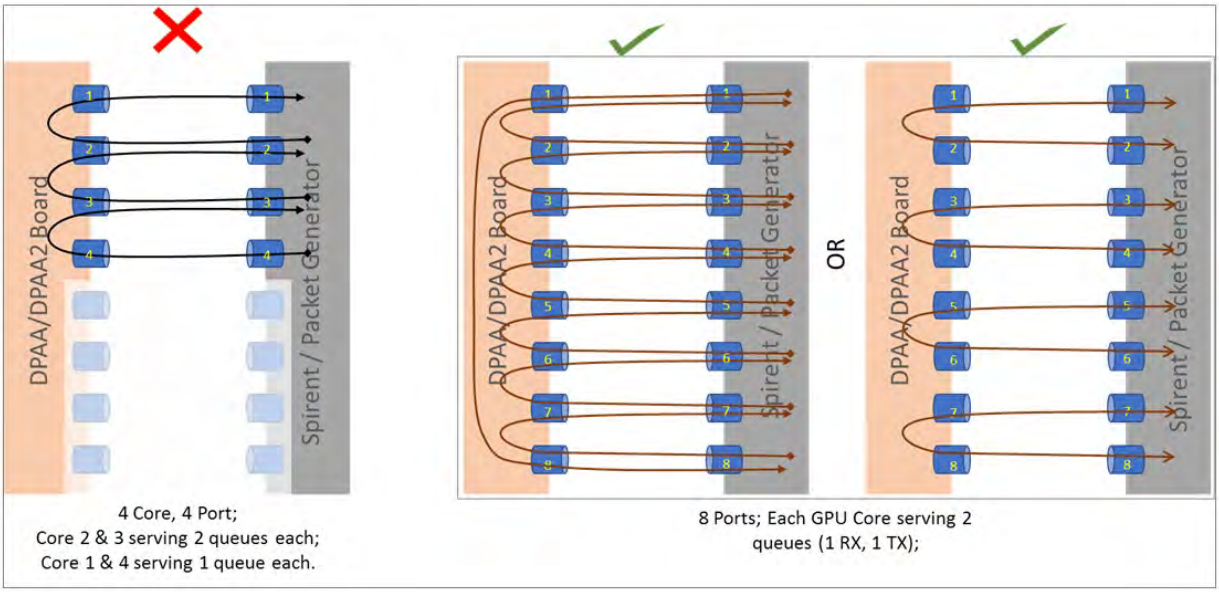


Image above describes a case with 4 ports where the CPU cores are not equally loaded. This is not a recommended combination as it would mean some streams being served (packet per second) slower than others. 8-port combination shown in the image above extends the mapping of 4 ports shown in image before. Once again, it should be noted that there are multiple ways to create a balanced set of streams. A performance setup should choose one baseline and all performance reports should be based on that baseline.

NOTE

For Performance measurement, performing I/O across non-equal capacity ports (`1G=>10G`, vice-verse) is not a valid case. This would lead to build up of queues on higher capacity links eventually stopping traffic when hardware is unable to obtain buffers for storing new incoming packets - eventually stopping traffic.

8. **Uninstalling PCI Ethernet (e1000) NIC Cards**

- It has been observed that when PCI Ethernet cards (for example, on DPAA/DPAA2 RDB boards Intel e1000) are installed, they poll the CPU cores frequently (Core 0 for isolation). This has adverse impact on the application performance if DPDK I/O threads are scheduled on same cores that service these interrupts.
- For best performance, such PCI Ethernet cards should be uninstalled from the hardware. If they cannot be uninstalled, then see the comments mentioned in section below to disable the interface by unlinking it from the Linux kernel.

NOTE

`nopci` can be appended to boot arguments to disable PCI devices completely from being detected by Linux kernel. It prevents PCI interrupts from being serviced by CPU. But, this option should not be used if there is dependency on any PCI device for system execution.

Before and during DPDK Application start

1. **Setting real-time priority for DPDK Application**
   - In full fledged distributions, such as Ubuntu, the root filesystem contains various system services by default. These services are targeted toward a generic environment. Many of these services require periodic CPU cycles. DPDK I/O threads execute as a run-to-completion process, infinitely looping over CPUs they are affined to. Services which require periodic CPU cycles can interrupt the DPDK I/O threads causing loss of packets and/or latency. Ideally, such services should be disabled or a rootfs without such services should be used for optimal performance. But, in case this cannot be done, real-time priority of application can also achieve desired results.
   - Execute the script `/usr/share/dpdk/enable_performance_mode.sh`. Care should be taken to run the DPDK application from same shell as the one on which script was executed. This is because the script sets some environment variables which are used by DPDK application to define real-time priorities for its threads. This script is also designed to set the CPU scaling governor to "performance" mode. It prevents the CPU from putting itself into lower power state when not busy. It causes loss of traffic in initial I/O streams when the CPU is expected to spin up to its maximum frequency.

     NOTE

     This script sets the real-time priorities for any DPDK application which is run after the script has been executed. It also applies to application configured to run on Core 0. Therefore, it is important to consider the implication. If the application is run on Core 0 and it is busy in I/O, it can lead to CPU stall causing complete lock-up. DPDK sample applications like `l2fwd, l3fwd, ipsec-secgw` are designed to relinquish the CPU when no I/O is being done. That way, using sample application, all core performance can be calculated. Similar care should be taken while developing custom DPDK applications. **As this script was primarily designed for host applications, it may require modification for it to be used with Virtualization cases (QEMU, VM) and OVS.**

     NOTE

     Though this script does not necessarily require core isolation and tickless kernel, it is still recommended that I/O cores be isolated and tickless kernel be used to get the best performance environment. Also, this script assumes that it is an Ubuntu environment with power governor support and that no other process is running in priority higher than DPDK application.

     NOTE

     An opposite script, `/usr/share/dpdk/disable_performance_mode.sh`, is also available. It puts the processor back in the "on-demand" scaling governor configuration and also removed the environment variables. It is important to run this script once performance verification of a DPDK sample application has been completed. This would avoid issues with inadvertently executing DPDK application on Core 0 and causing a lock-up.

2. **Using High-Performance (PEB) Buffer (Only for DPAA2)**
   - In DPAA2 platform, while creating the resource container using the `dynamic_dpl.sh` script, it is possible to toggle between high performance PEB buffers and normal buffers (DDR). By default, the high-performance buffers are enabled for LS2088A; for LS1088A, default configuration is normal buffers.

NOTE

For LS2088A, it is recommended to use high-performance buffers which are enabled by default. Though, there is caveat to this as described below.

PEB buffers are limited resources. Overusage of buffers, either through large number of queues or deep taildrop settings, can cause the PEB buffers to overflow causing an interruption of I/O. The hardware might also enter a state it cannot recover from until the board is reset.

Exact limitations of number of queues are based on various parameters and cannot be stated objectively without defining the use case. As a thumb-rule, refrain from using PEB buffers if configuration requires more than 1 queue per CPU core to be used, assuming all ports and CPU cores are being employed.

For toggling between normal and high performance buffers, use the following environment variable **before** executing the `dynamic_dpl.sh` script:

```
export DPNI_NORMAL_BUF=1
# disables high performance buffers; enables normal buffers
```

3. **Disabling PCI Ethernet (e1000) NICs**

   - As mentioned in the above section, it is preferable if no PCI Ethernet hardware (like e1000 on DPAA/DPAA2 boards) is installed. But, if a hardware device cannot be uninstalled, then the following command can be used to unlink the Ethernet card from PCI driver in the Linux Kernel; therefore, preventing the CPU cores from being interrupted with periodic interrupts. This is specially important when all core performance is to be recorded.

     ```
     echo 1 > /sys/bus/pci/devices/<PCI device BDF address>/remove
     ```

     In the above command, replace `<PCI device BDF address>` with appropriate BDF format bus address of the PCI device, for example `0000:01:00.0`, after properly bypassing the `:` character in the name to avoid failure reported by Linux Bash prompt. For example, `echo 1 > /sys/bus/pci/devices/0000\:01\:00.0/remove`.

     This command unlinks the PCI device with BDF address `0000:01:00.0` from the control of its PCI driver; therefore, disabling it from Linux kernel.

     NOTE

     Once the device is unlined from the PCI driver, it would not be usable through the Linux Kernel interface until bound to same or another PCI driver. It is out of scope for this document to record steps necessary for linking a PCI device to a PCI driver to bring it under Linux Kernel control.

4. **Interrupt Assignment for DPIO (Only for DPAA2)**

   With the Linux `cat /proc/interrupts` command, interrupts being serviced by each CPU core can be observed.

```
root@Ubuntu:~# cat /proc/interrupts r
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
...
113:          0          0          0          0          0          0          0          0
ITS-fMSI 230000 Edge      dpio.7
114:          0          0          0          0          0          0          0          0
ITS-fMSI 230001 Edge      dpio.6
115:          0          0          0          0          0          0          0          0
ITS-fMSI 230002 Edge      dpio.5
116:          0          0          0          0          0          0          0          0
ITS-fMSI 230003 Edge      dpio.4
117:          0          0          0          0          0          0          0          0
ITS-fMSI 230004 Edge      dpio.3
118:          0          0          0          0          0          0          0          0
ITS-fMSI 230005 Edge      dpio.2
119:          0          0          0          0          0          0          0          0
ITS-fMSI 230006 Edge      dpio.1
```

```
120:          0          0          0          0          0          0          0          0
ITS-fMSI 230007 Edge      dpio.0
...
```

This is especially important when DPDK uses some interrupts that CPUs are serving. For example, in the above representation, `DPIO` blocks are shown - these blocks are used by the Linux kernel assigned DPAA ports. Therefore, if a port is assigned to Linux (and some are assigned to DPDK) and I/O is performed on the ports assigned to Linux - there is a possibility that interrupts for that I/O spread across cores that DPDK is using. This should be avoided by setting the interrupt affinity. For example, if the DPIO.7 interrupt is considered in from the above output, following terminal snippet shows the affinity of that interrupt:

```
root@Ubuntu:~# cd /proc/irq/113/
root@Ubuntu:/proc/irq/113# ls -la
total 0
dr-xr-xr-x   3 root root 0 Mar  2 23:09 .
dr-xr-xr-x 111 root root 0 Mar  1 17:49 ..
-r--r--r--   1 root root 0 Mar  2 23:09 affinity_hint
dr-xr-xr-x   2 root root 0 Mar  2 23:09 dpio.7
-r--r--r--   1 root root 0 Mar  2 23:09 effective_affinity
-r--r--r--   1 root root 0 Mar  2 23:09 effective_affinity_list
-r--r--r--   1 root root 0 Mar  2 23:09 node
-rw-r--r--   1 root root 0 Mar  2 23:09 smp_affinity
-rw-r--r--   1 root root 0 Mar  2 23:09 smp_affinity_list
-r--r--r--   1 root root 0 Mar  2 23:09 spurious
root@Ubuntu:/proc/irq/113# cat smp_affinity
01
```

Output of `cat smp_affinity` is a mask for cores on which interrupt should be serviced. Affinity can be set by running following command:

```
cat 03 > smp_affinity # for enabling Core 0 and Core 1 for serving interrupts on DPIO.7
```

5. **DPDK Optimal Example Application Configuration**
   - Avoiding Core 0
     - As mentioned above, distributions like Ubuntu have large number of system services. Though some of these services can be disabled, there would always be cases of interrupts or uninterruptible services that require Core 0 cycles. Isolating the cores through Linux kernel can be done using Linux boot arguments. This allows isolated cores to be used exclusively for DPDK I/O threads.
     - Once a configuration of isolated cores is set, similar configuration should be done in DPDK application using the `-c` or `--coremask` command line option.
     - If 4 core (in LS1043A or LS1046A) or 8 core (LS1088A or LS2088A) performance is required, system services should be disabled. Though, it should be noted that performance number using Core 0 show undeterministic behavior of latency and packet losses. For example, LS2088A has been observed to perform fairly stable on 8 core configuration with services disabled, but same cannot be stated for LS1088A boards.
   - Avoiding Core 0 in case of Virtual Machine
     - Core 0 impact on the DPDK I/O performance is valid for host as well as for Virtual Machine (VM). While configuring DPDK application in VM, Core 0 should be avoided. The Qemu configuration should be such as to avoid using the Host's Core 0 for any VM logical core which is running DPDK I/O threads.
     - For a VM environment, OVS or similar switching stack maybe used on the host. Qemu configuration should be such as to avoid mapping the logical cores (VCPU) assigned to VM with any of the CPU cores which run the switching stack threads. `taskset` command is recommended for affining the Qemu threads (serving VM VCPUs) to a particular core. See Launch QEMU and virtual machine for more details.
   - Using Multi-queue configuration to spread load across multiple CPUs

— DPDK applications can utilize RSS based spreading of incoming frames across multiple queues servicing a particular port. This is especially helpful in obtaining better performance by utilizing 1:N mapping of ports to CPU cores. That is, more than 1 CPU core serves a single port.

It requires adequate configuration of Port-Queue-Core combination through DPDK application command line. For example, `l3fwd` application can be configured to use 8 ports on a LS2088A board for serving 2 ports using the following command:

```
l3fwd -c 0xFF -n 1 -- -p 0x3 --config="(0,0,0),(0,1,1),(0,2,2),(0,3,3),(1,0,4),(1,1,5),
(1,2,6),(1,3,7)"
```

In the above command, the `--config` argument takes multiple tuples of *(port, queue, core)*. Note that Core 0, 1, 2, and 3 serve Port number 0 using separate queue numbers.

Using similar configuration described for `l2fwd` application above, optimal utilization of Cores can be achieved. The command line options vary with DPDK application and DPDK online web manual should be referred for specific example applications.

Though the above command snippet utilizes Core 0, necessary care should be taken as described in text above.

— As mentioned above, DPDK uses RSS (Receive Side Scaling) to spread the incoming frames across multiple queues. Multi-queue setup must be supported by varying flows from the Packet Generator. The flows created should be such as to have varying Layer-2 or Layer-3 field values.

◦ As flow distribution is based on hash over Layer-2 and Layer-3 fields, it is possible that lower number of flows would distribute unevenly across queues. Number of flows created should be large enough to spread equally across all the configured queues.

- Consideration for CPU clusters

— SoC have multiple clusters housing one or more CPUs. Each cluster shares a L2 cache. In general, it allows threads sharing data over CPUs from same cluster to perform better than threads sharing data across CPUs from different clusters.

— For best performance, it is recommended that DPDK application configuration for selecting CPU cores should be such to either use all CPUs from same cluster or spread queues equally across clusters. When combined with Core 0 issue, it implies that using Cluster having Core 0 might perform slightly worse than using cluster that does not use Core 0.

- Using limited number of I/O buffers

— DPDK allows an application to change the number of maximum in-flight buffers. This is especially useful when there is memory constraint and DPDK application has limited resources.

— Each buffer, for processing, has to be fetched into the system caches (L2/L1). Larger the number of buffers in-flight simultaneously, more would be the flushing of buffer addresses. To avoid excessive pressure on the L2 caches (eviction, hit, miss cycle), lower number of buffers should be used. Exact numbers would depend on the use case and resources available.

For example, for `l3fwd` application, EAL argument, such as `--socket-mem=1025` can be provided to the application as shown in command snippet below. Note that the argument is provided before the `--` - these arguments are passed to DPDK framework rather than the application itself.

```
./l3fwd -c 0xFF -n 1 --socket-mem=1025 -- -p 0x1 --config="(0,0,0),(0,1,1),(0,2,2),
(0,3,3),(0,4,4),(0,5,5),(0,6,6),(0,7,7)"
```

- Degradation of OVS performance with increase in flows

— It has been observed that OVS does not perform well when the number of flows are large. This is due to the inherent design of OVS to use a flow matching table of size 8000. If larger than 8000 flows are used, the

overall performance degrades because of hash collisions. If more than 8000 flows are required, use the following command **after** OVS bridge has been created:

```
ovs-vsctl set bridge br0 other-config:flow-eviction-threshold=65535
```

This command would set the size of OVS internal flow table to 65535.

- Use `-n 1` as argument passed to DPDK EAL
  - `-n` argument for DPDK application is for defining number of DDR channels for the system - which is typically valid for NUMA architectures. This parameter is used for mempool memory alignments. For NXP SoCs, this argument should be set to "`1`". NXP SoCs supported by DPDK are non-NUMA.

9.1.11 Use cases

9.1.11.1 Traffic bifurcation using DPDMUX on DPAA2

9.1.11.1.1 Environment setup

NOTE

This section uses LS2088A Board as an example platform for demonstrating the use case. This use case would be applicable for all DPAA2 platforms including LS1088A, LX2160A.

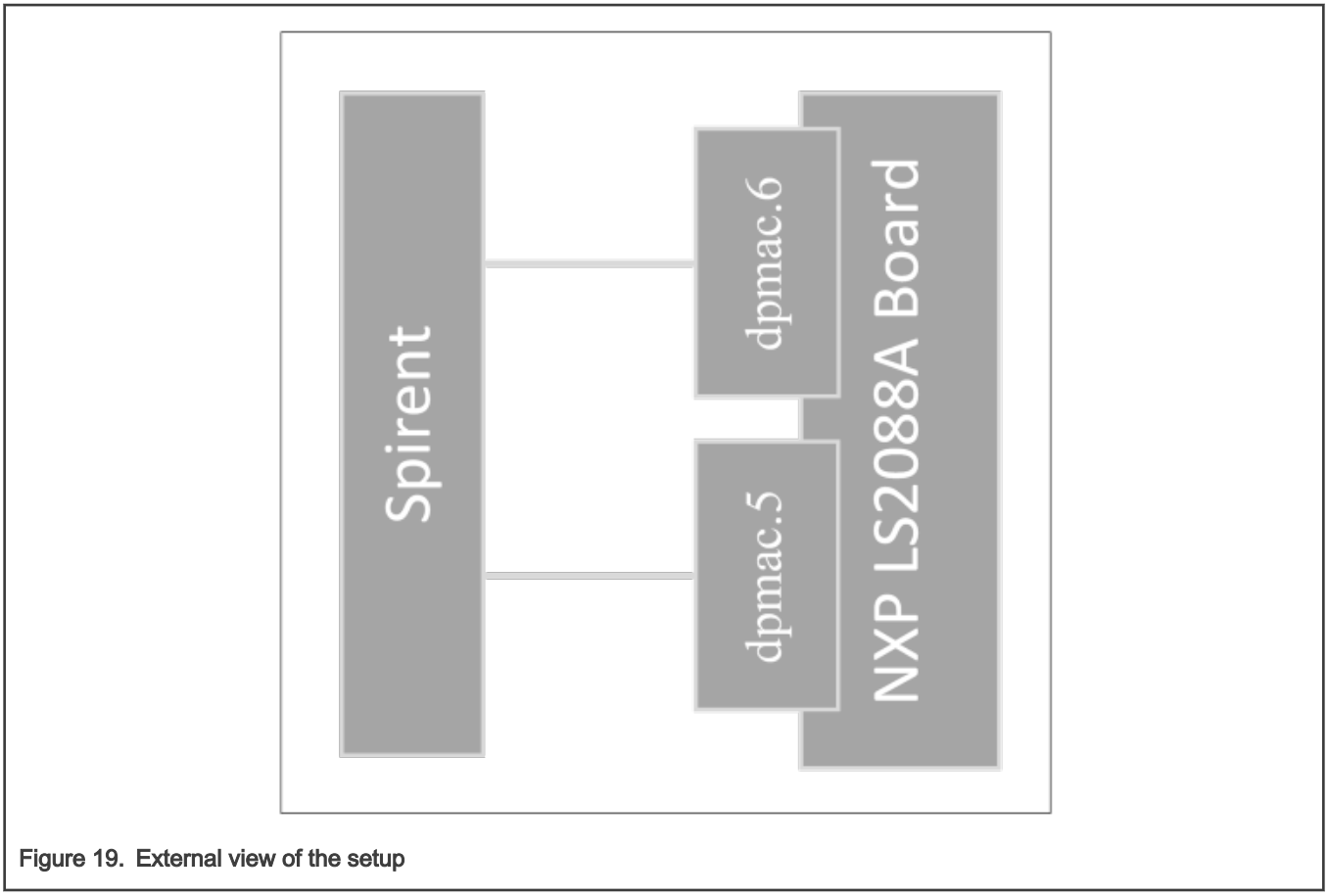


Figure 19. External view of the setup

In the above image, an NXP LS2088A board has been shown connected to a packet generator (Spirent).

NOTE

Though the example uses Spirent as packet generator, any other source of controlled packet transmission can also be used.

NOTE

The image uses `dpmac.5` and `dpmac.6` interfaces for demonstration. Any other other interface can also be used - in which case, the commands described below would have to be altered accordingly.

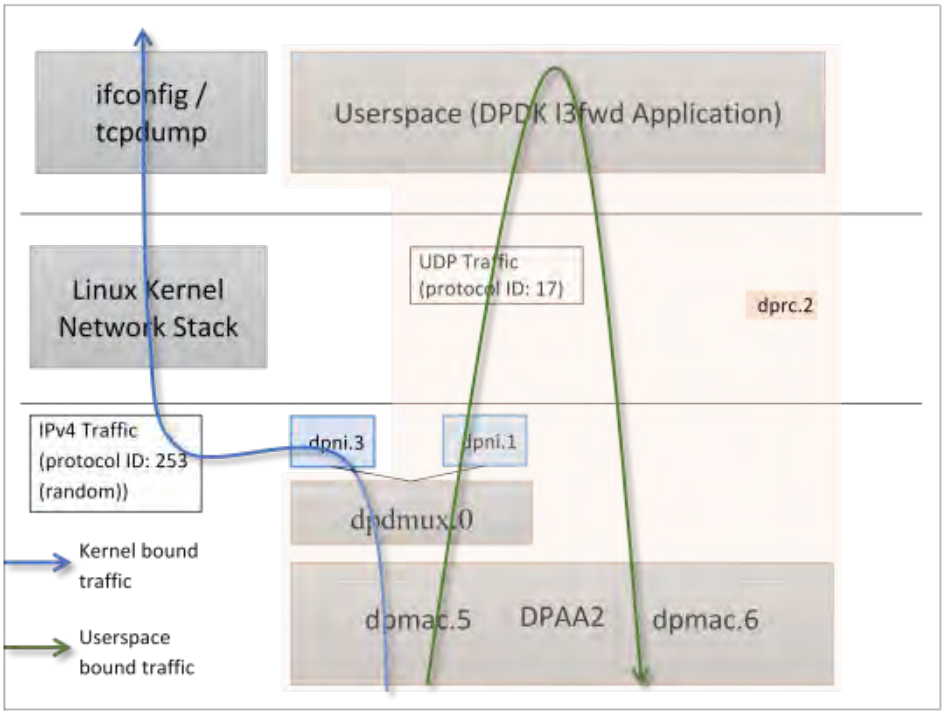


Figure 20. NXP LS2088A Internal Block for traffic bifurcation setup

In the above environment setup, a DPRC container (dprc.2) is created containing DPAA2 *dpmac.5* and *dpmac.6* interfaces. DPDMUX *dpdmux.0* is created with *dpni.1* and *dpni.3*, while *dpni.2* is connected with *dpmac.2*.

NXP LS2088A board has 8 10G links – 4 Fiber ports, and 4 Copper ports.

Figure 21. LS2088ARDB ports

On a standard LSDK configuration, these ports are represented using dpmac.X naming. Corresponding to the image above describing the ports, following is the naming convention:

- dpmac.1, dpmac.2, dpmac.3, and dpmac.4 are ETH4, ETH5, ETH6, and ETH7, respectively
- dpmac.5, dpmac.6, dpmac.7, and dpmac.8 are ETH0, ETH1, ETH2, and ETH3, respectively

Following are the commands to create the above setup:

Though this section uses dpmac.5 and dpmac.6 as interfaces; similar setup can be created using any other ports of LS2088A (or any other DPAA2 DPDMUX supporting board). Replace dpmac.X in commands below with equivalent port name.

1. Create DPRC with dpmac.5 and dpmac.6 attached. This creates dpni.1 and dpni.2 internally.

```
/usr/share/dpdk/dpaa2/dynamic_dpl.sh dpmac.5 dpmac.6
```

Output log:

```
##################### Container  dprc.2  is created ####################

Container dprc.2 have following resources :=>

* 1 DPMCP
* 16 DPBP
* 8 DPCON
* 8 DPSECI
* 2 DPNI
* 18 DPIO
* 2 DPCI
* 2 DPDMAI

######################### Configured Interfaces #########################
Interface Name        Endpoint              Mac Address
==============        ========              ==================
dpni.1                dpmac.5               -Dynamic-
dpni.2                dpmac.6               -Dynamic-
```

2. Create a DPNI for assigning to Linux kernel. This is used for forwarding the UDP traffic.

```
ls-addni --no-link
```

Output log:

```
Created interface: eth0 (object:dpni.3, endpoint: )
```

NOTE

It is important to note the dpni.X naming which is dynamically generated by the dynamic_dpl.sh script and ls-addni command. In case they are different from what is described in this section, corresponding changes should be done in the commands below.

3. Unplug the DPRC from VFIO, create a DPDMUX, assign DPNIs (`dpni.1` and `dpni.3`) to it, and then plug the DPRC back again to VFIO so that user space application can use it. This was already in plugged state because of the `dynamic_dpl.sh` script.

```
# Unbinding dprc.2 from VFIO
echo dprc.2 > /sys/bus/fsl-mc/drivers/vfio-fsl-mc/unbind

# Remove dpni.2 from dprc.2 so that it can be assigned to dpdmux
restool dprc disconnect dprc.2 --endpoint=dpni.1

# Create dpdmux with CUSTOM flow creation; Flows would be created
# from the Userspace (DPDK) application
restool dpdmux create --default-if=1 --num-ifs=2 --method DPDMUX_METHOD_CUSTOM --
manip=DPDMUX_MANIP_NONE --option=DPDMUX_OPT_CLS_MASK_SUPPORT --container=dprc.1

# Create DPDMUX with two DPNI connections and one DPMAC connection
restool dprc connect dprc.1 --endpoint1=dpdmux.0.0 --endpoint2=dpmac.5
restool dprc connect dprc.1 --endpoint1=dpdmux.0.1 --endpoint2=dpni.3
restool dprc connect dprc.1 --endpoint1=dpdmux.0.2 --endpoint2=dpni.1
restool dprc assign dprc.1 --object=dpdmux.0 --child=dprc.2 --plugged=1
```

NOTE

The default queue has been configured as `0.1` in DPDK DPMUX driver. In the above commands, `dpni.3` has been configured to `--endpoint1=dpdmux.0.1`. Therefore, all traffic which is not filtered would be sent by `dpdmux.0` to `dpni.3`. Further, the l2fwd application has currently configured UDP traffic (IPv4 Protocol Header field value 17) to be sent to `--endpoint1=dpdmux.0.2`, which corresponds to `dpni.1`.

```
# Bind the DPRC back to VFIO
echo dprc.2 > /sys/bus/fsl-mc/drivers/vfio-fsl-mc/bind

# Export the DPRC
export DPRC=dprc.2
```

If necessary, IP address can be assigned to `eth0`, which would appear in Linux OS to represent the dpni.3. Thereafter, external packet generator or a device can send ICMP traffic to confirm the bifurcation of traffic.

```
root@Ubuntu:~# ifconfig eth0 10.0.0.10/24 up
root@Ubuntu:~# ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.10  netmask 255.255.255.0  broadcast 10.0.0.255
        inet6 fe80::dce6:feff:fe3a:e105  prefixlen 64  scopeid 0x20<link>
        ether de:e6:fe:3a:e1:05  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 6  bytes 516 (516.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@Ubuntu:~# restool dpni info dpni.3
dpni version: 7.8
dpni id: 3
plugged state: plugged
endpoint state: 1
endpoint: dpdmux.0.1, link is up
link status: 1 - up
mac address: de:e6:fe:3a:e1:05
dpni_attr.options value is: 0
```

4. Run the l3fwd application

```
l3fwd -c 0xF0 -n 1 -- -p 0x3 --config="(0,0,4),(1,0,5)" -P --traffic-split-proto 17:2
```

NOTE

- In the above command, `-c 0xF0` corresponds to the cores being used by the DPDK Application. In case they are different, the mask should be changed.
- Further, `--config="(0,0,4),(1,0,5)"` represents **(Port, Queue, Core)** – which should align with the core masks provided. The Port value is 0 and 1 assuming only `dpmac.5` and `dpmac.6` have been assigned to the DPRC `dprc.2`. Only single queue per device has been considered. Numbering for all elements of this tuple starts from 0.
- `--traffic-split-proto 17:2` conveys to the application that protocol number `17` (UDP) should be sent to DPDMUX port `2` - which would be `dpni.1` (according to order of creation of ports in DPDMUX).

Output log:

```
EAL: Detected 8 lcore(s)
EAL: Probing VFIO support...
EAL: VFIO support initialized
EAL: PCI device 0000:01:00.0 on NUMA socket -1
```

```
EAL:   Invalid NUMA socket, default to 0
EAL:   probe driver: 8086:10d3 net_e1000_em
PMD: dpni.1: netdev created
PMD: dpni.2: netdev created
PMD: dpsec-0 cryptodev created
PMD: dpsec-1 cryptodev created
PMD: dpsec-2 cryptodev created
PMD: dpsec-3 cryptodev created
PMD: dpsec-4 cryptodev created
PMD: dpsec-5 cryptodev created
PMD: dpsec-6 cryptodev created
PMD: dpsec-7 cryptodev created
^[[6~L3FWD: Promiscuous mode selected
L3FWD: LPM or EM none selected, default LPM on
Initializing port 0 ... Creating queues: nb_rxq=1 nb_txq=4...  Address:00:00:00:00:00:01,
Destination:02:00:00:00:00:00, Allocated mbuf pool on socket 0
LPM: Adding route 0x01010100 / 24 (0)
LPM: Adding route 0x02010100 / 24 (1)
LPM: Adding route IPV6 / 48 (0)
LPM: Adding route IPV6 / 48 (1)
txq=4,0,0 txq=5,1,0 txq=6,2,0 txq=7,3,0
Initializing port 1 ... Creating queues: nb_rxq=1 nb_txq=4...  Address:DA:CA:B2:78:68:19,
Destination:02:00:00:00:00:01, Allocated mbuf pool on socket 0
txq=4,0,0 txq=5,1,0 txq=6,2,0 txq=7,3,0
Initializing rx queues on lcore 4 ... rxq=0,0,0
Initializing rx queues on lcore 5 ... rxq=1,0,0
Initializing rx queues on lcore 6 ...
Initializing rx queues on lcore 7 ...

Checking link statusdone
Port0 Link Up. Speed 1000 Mbps -full-duplex
Port1 Link Up. Speed 10000 Mbps -full-duplex
L3FWD: entering main loop on lcore 5
L3FWD:  -- lcoreid=5 portid=1 rxqueueid=0
L3FWD: lcore 7 has nothing to do
L3FWD: lcore 6 has nothing to do
L3FWD: entering main loop on lcore 4
L3FWD:  -- lcoreid=4 portid=0 rxqueueid=0
```

5. Send following packet streams from the Packet generator (in this case, Spirent)
    a. Packets sent to `dpmac.5`
        i. UDP Traffic: IPv4 Packet with Protocol ID field (next protocol) = 0x11 (hex) or 17 (decimal); Size greater than 82 bytes.
        ii. IPv4 Only Traffic: IPv4 Traffic with any random Protocol ID (next protocol) = 253 (Experimental); Size greater than equal to 64 bytes. Src IP: 1.1.1.1; Dst IP: 2.1.1.1 (so that packets can be forwarded by l3fwd application from `dpmac.5 to dpmac.6`.
    b. Packets sent to dpmac.2
        i. IPv4 Only Traffic: IPv4 Traffic with any random Protocol ID (next protocol) = 253 (Experimental); Size greater than equal to 64 bytes. Src IP: 2.1.1.1; Dst IP: 1.1.1.1 (so that packets can be forwarded by l3fwd application from `dpmac.6 to dpmac.5`.

9.1.11.1.2 Expected results

Following is the expected output:

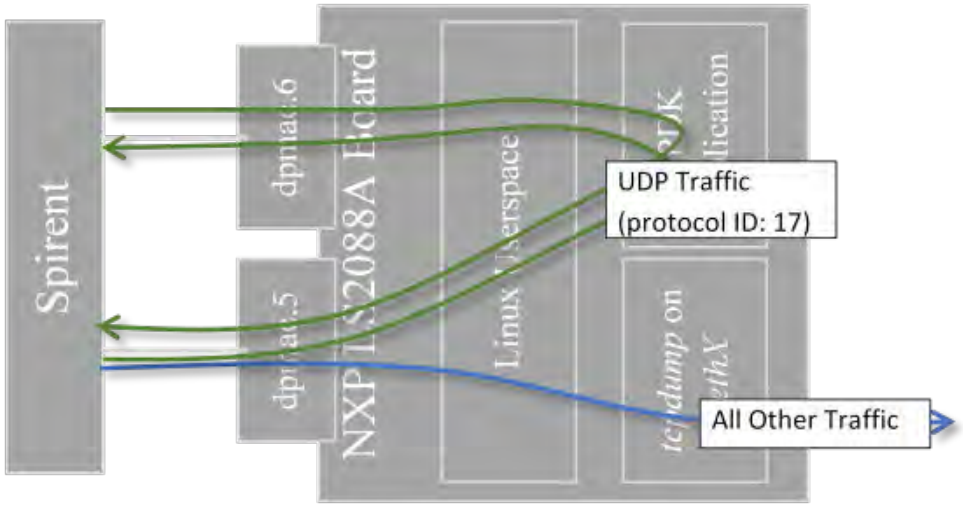


Figure 22. Expected output on Linux User space and Packet Generator

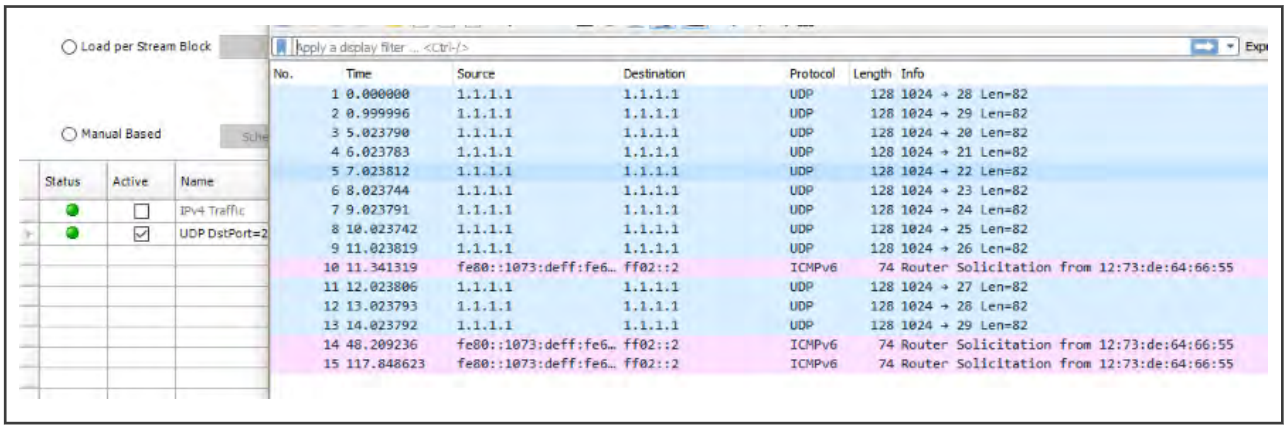
1. All traffic with UDP Protocol set in IPv4 header would be sent to Linux Kernel network stack and would be eventually available on the Ethernet interface (backed by *dpni.3*). Application like tcpdump would be able to demonstrate the packets coming in:

```
root@localhost:~# ifconfig
…
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
inet 10.0.0.10  netmask 255.255.255.0  broadcast 10.0.0.255
inet6 fe80::5885:a5ff:fe1c:76af  prefixlen 64  scopeid 0x20<link>
ether 5a:85:a5:1c:76:af  txqueuelen 1000  (Ethernet)
RX packets 0  bytes 0 (0.0 B)
RX errors 0  dropped 0  overruns 0  frame 0
TX packets 5  bytes 426 (426.0 B)
TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
…

tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
22:39:10.286502 IP 10.0.0.11 > Ubuntu.ls2088ardb:  ip-proto-200 90
22:39:11.286385 IP 10.0.0.11 > Ubuntu.ls2088ardb:  ip-proto-201 90
22:39:12.286286 IP 10.0.0.11 > Ubuntu.ls2088ardb:  ip-proto-202 90
22:39:13.286172 IP 10.0.0.11 > Ubuntu.ls2088ardb:  ip-proto-203 90
22:39:14.286075 IP 10.0.0.11 > Ubuntu.ls2088ardb:  ip-proto-204 90
22:39:15.285958 IP 10.0.0.11 > Ubuntu.ls2088ardb:  ip-proto-205 90
22:39:16.285845 IP 10.0.0.11 > Ubuntu.ls2088ardb:  ip-proto-206 90
22:39:17.285757 IP 10.0.0.11 > Ubuntu.ls2088ardb:  ip-proto-207 90
22:39:18.285636 IP 10.0.0.11 > Ubuntu.ls2088ardb:  ip-proto-208 90
22:39:19.285541 IP 10.0.0.11 > Ubuntu.ls2088ardb:  ip-proto-209 90
^C
10 packets captured
10 packets received by filter
0 packets dropped by kernel
root@Ubuntu:~#
```

In the above output, it can be observed that packets of different IPv4 Protocol fields are being received in Linux. (This setting can be configured in Spirent). `Ubuntu.ls2088ardb` refers to the local machine IP `10.0.0.10` which was configured using `ifconfig`.

2. All other traffic would be visible in the packet generator being reflected by ‘l3fwd’ application. Below is the screen-grab of Wireshark output of packet captured by Spirent which were reflected by the l3fwd application:



9.1.11.1.3 Application Limitation

Currently, the application has been designed to only bifurcate traffic on the basis of the protocol number matched in the IP header (provided through `--traffic-split-proto 17:2` argument). Also, only a single matching criteria can be provided for now. For enhanced cases, the application will have to be modified.

9.1.11.2 DPDK multi-process

9.1.11.2.1 DPDK Multiprocess Support

Supported Platforms (and their derivatives):

- DPAA2: LS108x, LS208x, LX2160

NXP DPDK provides a set of data plane libraries and network interface controller driver for Layerscape platforms. This section provides information about multiprocess support in DPDK for NXP platforms.

- Multiprocess: In DPDK context, this is a deployment model where multiple independent processes are executed each of which can functionally behave as threads of a parent process.
- Parent/Primary Process: The first DPDK process which is run. In the DPDK multiprocess model, this process is responsible for configuration of the devices and any other common configuration to be used by the secondary processes. This process can also perform I/O on the devices. While executing the process, if `--proc-type=primary` is used as an EAL argument, the process is expected to be primary. In case this is not the first DPDK process, then this would result in error.
- Child/Secondary Process: Every next DPDK process which is started with `--proc-type=secondary` EAL argument. Another way is to add `--proc-type=auto` as EAL argument which automatically selects between primary or secondary based on order of execution.

NOTE

In case another instance of DPDK application is started but it is not expected to be part of a Multiprocess model (separate DPDK instance), then adequate configuration of hugepages need to be done. By default, DPDK maps all available hugepages which only be consumed by a single process, and its secondary processes.

9.1.11.2.2 Various Multiprocess Models

NOTE

This section is only applicable for DPAA2 as this involves I/O in the secondary process. Refer to following sections for DPAA support.

Based on the functionality of the processes, the multiprocess model can be categorized into two broad spectrums:

- Symmetric: A model in which the Primary and Secondary processes have similar functionality. For example, when Primary process is performing I/O over one `eth` device, while one or more secondary processes are also performing I/O on separate `eth` devices. Or, if each device is equally shared across multiple processes for I/O.

```
+----------+    +------------+   +------------+    +------------+
| Primary  |    | Secondary1 |   | Secondary2 |... | SecondaryN |
+-----XV---+    +------XV----+   +------XV----+    +-----XV-----+
      ||              ||              ||              ||
+-----||---+    +------||----+   +------||----+    +-----||-----+
| Device 1 |    | Device 2   |   | Device 3   |... | Device N   |
+----------+    +------------+   +------------+    +------------+
```

Where, { X = Rx and V = Tx } signifying I/O (RX/TX, both). Another way to visualize is where I/O (RX/TX) is performed by each process on same device, maybe through separate queues:

```
+----------+    +------------+   +------------+    +------------+
| Primary  |    | Secondary1 |   | Secondary2 |... | SecondaryN |
+-----XV---+    +------XV----+  +------XV----+    +------XV-----+
      ||              ||             ||                ||
      ||  .-------------`             ||                ||
      ||  ||   .-------------------------`              ||
      ||  ||   ||    .-----------------------------------`
      ||  ||   ||    ||
+-----||--||---||----||--+
|         Device 1       |
+------------------------+
```

- Asymmetric: A model in which the primary and secondary processes have dis-similar functionality in terms of I/O. For example, primary process performing RX on a device, transferring data to a secondary process through some internal process mechanism (IPC, for example, Ring), which in turn does TX on the same device.

```
+----------+    +------------+
| Primary  >----X Secondary1 |
+-----X-V-V+    +------V-----+
      | | |            |         +------------+
      | |  `-----------)---------X Secondary2 |
      | |              |         +------V-----+
      | |              |                |
      | |              |                |           +------------+
      |  `-------------)----------------)-------...-X SecondaryN |
      |                |                |           +-----V------+
      |                |                |                 |
      |.---------------'                |                 |
      ||.-------------------------------`                 |
      |||.--------------------------------------------------`
      ||||
+-----||||-+
| Device 1 |
+----------+
```

Current implementation of NXP DPDK supports both mode on the supported platforms. The design of the application drives the mode being used.

9.1.11.2.3 Environment Setup

NOTE

This section is applicable for DPAA2 only.

NOTE

For all DPDK multiprocess use cases, disable ASLR - Address Space Layout Randomization[1]This is the default setting on various Linux distributions for preventing stack and other malicious address manipulation attacks. This works by randomizing the address-space layout of the ELF binary. This impact secondary process because the second or further process would attempt to find the same address space as the primary process (hugepage). This should be disabled using:

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

DPRC or DPAA2 Resource Container contains a number of resources which need to be segregated between the primary and secondary process. Initializing all the I/O devices (dpni, dpseci, dpdmai, etc) is done by primary - secondary process is not expected to initialize any I/O device. Only control devices like `dpio`, `dpmcp` need to be initialized by secondary for its own work.

While executing the primary or secondary processes, list of devices to blacklist (those which are not to be configured) need to be passed. Alternatively, a list of all devices which are to be configured can be passed. This list is important as overlap would result in incorrect configuration.

1. Create enough `dpmcp` devices: While creating the DPRC (through `dynamic_dpl.sh` script), create as many `dpmcp` as the number of processes (primary and secondary) expected to use the DPRC.

```
$ export DPMCP_COUNT=3              # for 1 Primary, 2 Secondary
$ ./dynamic_dpl.sh dpmac.1 dpmac.2
```

2. Create enough `dpio` devices to suffice the total number of cores being used across primary and secondary, plus one additional for each process. For example, in case primary is to be run with 2 cores, and secondary with 2 Cores, total `dpio` required are: (Total Process = 3) x (3 `dpio` per process) = 9

NOTE

A large number of `dpio` devices are already created in default container created by `dynamic_dpl.sh`.

```
$ export DPIO_COUNT=10              # A larger number to accommodate conf changes
$ ./dynamic_dpl.sh dpmac.1 dpmac.2
```

Assuming that following DPRC is created:

```
$ restool dprc show dprc.2
```

```
dprc.2 contains 58 objects:
object          label           plugged-state
dpni.3                          plugged
dpni.2                          plugged
dpni.1                          plugged
dpbp.16                         plugged
dpbp.15                         plugged
dpbp.14                         plugged
dpbp.13                         plugged
dpbp.12                         plugged
dpbp.11                         plugged
dpbp.10                         plugged
dpbp.9                          plugged
dpbp.8                          plugged
dpbp.7                          plugged
dpbp.6                          plugged
dpbp.5                          plugged
dpbp.4                          plugged
```

[1] https://en.wikipedia.org/wiki/Address_space_layout_randomization

```
    dpbp.3                          plugged
    dpbp.2                          plugged
    dpbp.1                          plugged
    dpci.1                          plugged
    dpci.0                          plugged
    dpseci.7                        plugged
    dpseci.6                        plugged
    dpseci.5                        plugged
    dpseci.4                        plugged
    dpseci.3                        plugged
    dpseci.2                        plugged
    dpseci.1                        plugged
    dpseci.0                        plugged
    dpdmai.1                        plugged
    dpdmai.2                        plugged
    dpmcp.23                        plugged
    dpmcp.22                        plugged
    dpmcp.21                        plugged
    dpio.17                         plugged
    dpio.16                         plugged
    dpio.15                         plugged
    dpio.14                         plugged
    dpio.13                         plugged
    dpio.12                         plugged
    dpio.11                         plugged
    dpio.10                         plugged
    dpio.9                          plugged
    dpio.8                          plugged
    dpcon.8                         plugged
    dpcon.7                         plugged
    dpcon.6                         plugged
    dpcon.5                         plugged
    dpcon.4                         plugged
    dpcon.3                         plugged
    dpcon.2                         plugged
    dpcon.1                         plugged
```

Ignore `dpni`, `dpbp`, `dpci`, `dpseci`, `dpcon` - as they are all configured by the primary process only. Secondary process is designed to skip them. But, `dpio` and `dpmcp` are important considerations.

3. Start primary application with EAL arguments for blacklisting

```
# Only allowing dpio.8, dpio.9, dpio.10, dpmcp.21 in primary; blacklisting all others

$ ./primary_process -c 0x3 -b fslmc:dpio.11 -b fslmc:dpio.12 -b fslmc:dpio.13 \
      -b fslmc:dpio.14 -b fslmc:dpio.15 -b fslmc:dpio.16 -b fslmc:dpio.17 \
      -b fslmc:dpmcp.22 -b fslmc:dpmcp.23 -- <application arguments>

# Only allowing fslmc:dpio.11, fslmc:dpio.12, fslmc:dpio.13, fslmc:dpmcp.22 in secondary
process 1

$ ./secondary_process1 -c 0x3 -b fslmc:dpio.8 -b fslmc:dpio.9 -b fslmc:dpio.10 \
      -b fslmc:dpio.14 -b fslmc:dpio.15 -b fslmc:dpio.16 -b fslmc:dpio.17 \
      -b fslmc:dpmcp.21 -b fslmc:dpmcp.23 -- <application arguments>

# Only allowing fslmc:dpio.14, fslmc:dpio.15, fslmc:dpio.16, fslmc:dpmcp.23 in secondary process
2; ignoring dpio.17

$ ./secondary_process1 -c 0x3 -b fslmc:dpio.8 -b fslmc:dpio.9 -b fslmc:dpio.10 \
```

```
      -b fslmc:dpio.11 -b fslmc:dpio.12 -b fslmc:dpio.13 -b fslmc:dpio.17 \
      -b fslmc:dpmcp.21 -b fslmc:dpmcp.22 -- <application arguments>
```

NOTE

- In the above format `<bus>:<device>` is the way to provide the device identifier for blacklisting/whitelisting.
- Another method would be to whitelist all devices - but, that would require listing even the `dpni, dpci, dpseci, and dpcon` devices. That would increase the length of the argument to unmanageable lengths.

9.1.11.2.4 Executing DPDK example application

NOTE

This section is applicable for DPAA2 only.

NOTE

Applications used in the snippets below are not available on the LSDK rootfs. For standalone compilation of these applications, refer Compiling DPDK Example Applications

NOTE

It important to note that before any secondary application execution, ASLR support should be disabled by using `echo 0 > /proc/sys/kernel/randomize_va_space`.

DPDK provides two sample applications which can be used for I/O using multiprocess model.

```
./examples/multi_process/symmetric_mp             # symmetric model example
./examples/multi_process/client_server_mp         # asymmetric model example
```

Some other examples are also provided from NXP which use the available hardware support in DPAA2:

```
./examples/multi_process/symmetric_mp_qdma        # symmetric model with QDMA example
```

Detailed explanation can be seen from DPDK documentation: https://doc.dpdk.org/guides/sample_app_ug/multi_process.html

Using the same DPRC shown as sample above.

1. Executing `dpdk-symmetric_mp` with 1 Primary, 1 Secondary:

   ```
   # Running primary process with single Core, 2 ports; assigning 1 dpmcp and 5 dpio to it.

   dpdk-symmetric_mp -c 0x1 -n 1 -b fslmc:dpio.13 -b fslmc:dpio.14 -b fslmc:dpio.15 \
           -b fslmc:dpio.16 -b fslmc:dpio.17 -b fslmc:dpmcp.22 -b fslmc:dpmcp.23 \
           -- -p 0x3 --num-procs=2 --proc-id=0
   ```

   In the above, `--num-procs=2` signifies 2 processes in total. `--proc-id=0` is the identifier for the primary process.

   ```
   # Running secondary process with single core (not overlappping with primary), 2 ports (same as
   primary); Assigning 1 dpmcp and 5 dpio to it. (We can ignore the extra dpmcp preserved for 3
   process, if any)

   dpdk-symmetric_mp -c 0x2 -n 1 --proc-type=secondary -b fslmc:dpio.8 \
           -b fslmc:dpio.9 -b fslmc:dpio.10 -b fslmc:dpio.11 -b fslmc:dpio.12 \
           -b fslmc:dpmcp.21 -b fslmc:dpmcp.23 -- -p 0x3 --num-procs=2 --proc-id=1
   ```

   In case more than one instance of application is to be executed, similar blacklisting has to be done to distribute resources. Further, `--num-procs` and `--proc-id` too need to be changed.

   Send I/O to the ports assigned to the processes and observe traffic being reflected back.

2. Executing `dpdk-client_server_mp` with 1 Primary, 1 Secondary:

   This sample application has two different applications which are executed as server and client. Server process is responsible for RX from interfaces (all) and distributing to Client for TX (one queue per device).

```
# Running server (primary) process with 1 Core, 2 ports; assigning 1 dpmcp and 5 dpio to it.

dpdk-mp_server -c 0x1 -n 1 -b fslmc:dpio.13 -b fslmc:dpio.14 -b fslmc:dpio.15 \
        -b fslmc:dpio.16 -b fslmc:dpio.17 -b fslmc:dpmcp.22 -b fslmc:dpmcp.23 \
        -- -p 0x3 -n 1
```

```
 # Running client (secondary) process with 1 Core, 2 ports; assigning 1 dpmcp and 5 dpio to it.

dpdk-mp_client -c 0x2 -n 1 --proc-type=secondary -b fslmc:dpio.8 \
        -b fslmc:dpio.9 -b fslmc:dpio.10 -b fslmc:dpio.11 -b fslmc:dpio.12 \
        -b fslmc:dpmcp.21 -b fslmc:dpmcp.23 -- -n 0
```

   Send I/O to the ports assigned to the processes and observe traffic being forwarded.

3. Executing QDMA example application:

   `dpdk-symmetric_mp_qdma` application is similar to `dpdk-symmetric_mp` except that the mbuf (buffer) transfers between RX and TX are done using the NXP DPAA2 QDMA (`dpdmai`) hardware block.

```
# Running primary process with single Core, 2 ports; assigning 1 dpmcp and 5 dpio, 1 dpdmai
to it:
# Assuming that there are two DPDMAI objects in the DPRC: dpdmai.1 and dpdmai.2, the command is
very similar to the symmetric_mp example:

dpdk-symmetric_mp_qdma -c 0x1 -n 1 -b fslmc:dpio.13 -b fslmc:dpio.14 -b fslmc:dpio.15 \
        -b fslmc:dpio.16 -b fslmc:dpio.17 -b fslmc:dpmcp.22 -b fslmc:dpmcp.23 \
        -b fslmc:dpdmai.1 -- -p 0x3 --num-procs=2 --proc-id=0
```

   Notice the extra parameter `-b fslmc:dpdmai.1` as compared the `symmetric_mp` command. This extra parameter conveys this process to ignore the `dpdmai.1` block and use `dpdmai.2` in Primary.

```
# Running secondary process with single core (not overlappping with primary), 2 ports (same as
primary); Assigning 1 dpmcp and 5 dpio, 1 dpdmai to it. (We can ignore the extra dpmcp preserved
for 3 process, if any)

dpdk-symmetric_mp_qdma -c 0x2 -n 1 --proc-type=secondary -b fslmc:dpio.8 \
        -b fslmc:dpio.9 -b fslmc:dpio.10 -b fslmc:dpio.11 -b fslmc:dpio.12 \
        -b fslmc:dpmcp.21 -b fslmc:dpmcp.23 -b fslmc:dpdmai.2 \
        -- -p 0x3 --num-procs=2 --proc-id=1
```

4. Executing `dpdk-l2fwd-crypto` example application:

   `dpdk-l2fwd-crypto` is a standard DPDK crypto demonstration application, which also supports multiprocess model.

   Primary differences with usual execution of `dpdk-l2fwd-crypto` are the `mp-emask` and `mp-cmask` arguments passed - signifying Ethernet ports and crypto ports, respectively, which would be used by l2fwd-crypto instance.

   Following can be a possible primary application command, using the same DPRC as used for examples described above:

```
dpdk-l2fwd-crypto -c 0x3 -n 1 -b fslmc:dpio.13 -b fslmc:dpio.14 -b fslmc:dpio.15 \
        -b fslmc:dpio.16 -b fslmc:dpio.17 -b fslmc:dpmcp.22 -b fslmc:dpmcp.23 \
        -- -p 0x3 -q 1 --mp-emask 0x1 --mp-cmask 0x1 --chain CIPHER_ONLY \
    --cipher_algo aes-cbc --cipher_op ENCRYPT \
    --cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 \
    --cipher_iv 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10
```

Secondary application can be executed like:

```
dpdk-l2fwd-crypto -c 0xc -n 1 --proc-type=secondary -b fslmc:dpio.8 \
        -b fslmc:dpio.9 -b fslmc:dpio.10 -b fslmc:dpio.11 -b fslmc:dpio.12 \
        -b fslmc:dpmcp.21 -b fslmc:dpmcp.23 -b fslmc:dpdmai.2 \
        -- -p 0x3 -q 1 --mp-emask 0x2 --mp-cmask 0x2 --chain CIPHER_ONLY \
    --cipher_algo aes-cbc --cipher_op ENCRYPT \
    --cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 \
    --cipher_iv 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10
```

5. Non-I/O performing applications: `dpdk-pdump` and `dpdk-procinfo`

   Both, `dpdk-pdump` and `dpdk-procinfo` are examples of secondary applications can query the primary application (the first one) for information, without performing any actual I/O over the network devices. Both these are compiled as default as part of the DPDK framework, just like `testpmd` and can be obtained from `app/` folder in the compiled output.

   a. `dpdk-pdump` for capturing packets

      NOTE

      For PCAP support, 'libpcap' library has to be provided to DPDK during compilation. This needs to be compiled for ARM64 target.

      Execute the primary application (only `testpmd` currently supports interfacing with `dpdk-pdump`):

      ```
      dpdk-testpmd -c 0xff -n 1 -- -i --portmask=0x3 --nb-ports=2
      ...
      # Start I/O on application
      testpmd> set fwd io
      testpmd> start
      ```

      Execute the secondary application, `dpdk-pdump`. Just like the `dpdk-symmetric_mp` and other multiprocess application, isolation of DPRC resources like `dpio` and `dpmcp` needs to be done.

      ```
      dpdk-pdump -n 1 -b fslmc:dpio.8 -b fslmc:dpio.9 -b fslmc:dpio.10 -b fslmc:dpio.11 \
      -b fslmc:dpio.12 -b fslmc:dpio.13 -b fslmc:dpio.14 -b fslmc:dpio.15 \ -b fslmc:dpio.16
      -b fslmc:dpmcp.29 --mbuf-pool-ops-name="ring_mp_mc" \ -- --pdump "port=0,queue=*,rx-
      dev=./rx.pcap"
      ```

      In the above command, `port=0, ...` parameters convey `dpdk-pdump` app that capture should be done on port 0 (for example, `dpni.1`) and packets being received on all queues `queue=*`. Further, all the captured packets can be dumped to a PCAP file using `rx-dev=<path to pcap file>` if LIBPCAP was enabled.

      In the above example, only Rx'd packets are being written to PCAP. For Tx'd packets, use something similar to `port=0,queue=*,tx-dev=./...` where the output PCAP file is different from Rx'd packets. Both, RX and TX, options can be simultaneously provided.

      Stop `dpdk-pdump` using Ctrl+C and copy the PCAP file for reading through external application like Wireshark

   b. `dpdk-procinfo` for dumping application information like memory or port statistics

      `dpdk-procinfo` is an inbuilt application which allows dumping information like memory and statistics of a running (primary) DPDK application.

      Execute the primary application: (Unlike `dpdk-pdump`, `dpdk-procinfo` can be run along with any primary application):

      ```
      dpdk-testpmd -c 0xff -n 1 -- -i --portmask=0x3 --nb-ports=2
      ...
      # Start I/O on application
      ```

```
testpmd> set fwd io
testpmd> start
```

Execute the secondary application, `dpdk-procinfo`:

NOTE

It is a good practice to isolate the resources of the application (`dpio`, `dpmcp` and `dpdmai`) for all cases of secondary application. Though `dpdk-procinfo` can work without that isolation because it doesn't necessarily access device-specific data, it is a good practice to do the isolation in this case as well.

```
dpdk-procinfo -- -m
```

Above command dumps the memory layout of the primary DPDK application to the screen. There are other switches also available like `-s` which can dump the statistics.

NOTE

- `dpdk-l2fwd/dpdk-l3fwd` in their current design are not suited for multiprocess execution and therefore, would not work with substantial modification of segregating the queues and RX/TX processing.
- For exiting the application, it is advisable to send SIGKILL to all the application. Similar to "killall <application name>". If not, all secondary should be killed first (Ctrl+C or SIGKILL) before the primary process.

9.1.11.3 Traffic Policing in DPAA

On the DPAA SoCs (such as LS1043A, LS1046A), traffic policing can be done using simple configuration using the FMC tool. This is part of the Ingress Traffic Management in the FMan block which sits between the QMan and the hardware in the overall vertical block layout of DPAA. Once the frames are ingressed from WRIOP into FMan, post the Parser and Classify block, the Policer block can be configured to color (and drop) frames based on the policy. Policer blocks pass along any non-dropped frame toward the QMan through the FMan<=>QMan interface. FMan supports up to 256 policy profiles.

NOTE

A sample XML has been added to DPDK source folder `/usr/share/dpdk/dpaa/usdpaa_policy_hash_ipv4_1queue_policer_ls1046.xml`. This section uses snippets from this file. This file is only applicable to the LS1046A boards.

1. Define a Policer policy XML. In this example, a copy of `usdpaa_policy_hash_ipv4_1queue_policer_ls1046.xml` has been used.

```
<policer name="policer9">
    <algorithm>rfc2698</algorithm>
    <color_mode>color_aware</color_mode>
    <CIR>5000000</CIR>
    <EIR>5500000</EIR>
    <CBS>5000000</CBS>
    <EBS>5500000</EBS>
    <unit>packet</unit>
    <action condition="on-red" type="drop"/>
</policer>
```

In the above configuration, a **RFC2698** (Two Rate Three Color Marker) policer has been defined. This policy is based on 2 token buckets representing two rates - **PIR/EIR** or Peak/Exceed Information Rate and **CIR** or Committed Information Rate - and 3 colors - Red, Yellow and Green. Based on the information configured above for **CBS** (Committed Burst Size) and **EBS** (Peak/Exceed Burst Size), streams are marked as being colored for one of the 3 colors.

NOTE

Based on the standard trTCM (Three Color Marker), CIR is rate of filling the committed bucket and CBS being its initial size, EIR is the rate of filling the exceed bucket and EBS being its initial size. Therefore, if a flow of packets is received and it exceeds the EIR, then it is marked as Red; if it exceeds CIR but does not exceed EIR, then it is marked as Yellow; otherwise, it is marked as Green.

The configuration above has following elements:

- `policer name` is the name of the Policer which would be used for assigning to the distribution policy records
- `algorithm` which has to be defined to `rfc2968`, or `rfc4115` for trTCM for differentiated services or `pass-through` to disable policing (default)
- `color_mode` which can be set to either of `color_aware` or `color_blind`. `color_aware` uses the pre-colored information, if any, to make decisions, while `color_blind` ignores the existing color information.
- `CIR`, `EIR` - for Committed Information Rate and Exceed Information Rate, respectively. Metric for these elements is defined by `unit` per second (explained below).
- `CBS`, `EBS` - for Committed Burst Size and Exceed Burst Size, respectively. Metric for these elements is defined by `unit` per second (explained below).
- `unit` - defines the metric for all the four configuration parameters, namely `CIR, CBS, EIR, EBS`. For information rate, it would be `unit/second` whereas for burst size it would `unit`.

2. Apply the policy to one or more distribution policies:

```
    <distribution name="hash_ipv4_src_dst_dist9">
        <queue count="1" base="0xd00"/>
        <key>
            <fieldref name="ipv4.src"/>
            <fieldref name="ipv4.dst"/>
        </key>
        <action type="policer" name="policer9"/>
    </distribution>
```

3. Apply the policy file using the FMC tool

```
root@LS1046ARDB:~# fmc -x
root@LS1046ARDB:~# fmc -c /usr/share/dpdk/dpaa/usdpaa_config_ls1046.xml -p /usr/share/dpdk/dpaa/
usdpaa_policy_hash_ipv4_1queue_policer_ls1046.xml -a
```

Now, perform I/O to see the impact of policing being implemented.

9.1.11.4 Precision Time Protocol (IEEE1588)

The Precision Time Protocol (PTP) is a protocol used to synchronize clock throughout a computer network. PTP was originally defined in IEEE1588-2002 standard.

To test ptp functionality in DPDK, one can use DPDK example application “ptpclient” present in DPDK source code. ptpclient application uses DPDK IEEE1588 API to communicate with a PTP master clock to synchronize the time on NIC and, optionally, on the Linux system.

NOTE

ptpclient application is based on assumption that it is single-threaded and it always works in slave mode.

.

9.1.11.4.1 Supported platforms

PTP is supported for DPAA2 based LS1088A and LS2088A family of SoCs.

9.1.11.4.2 Build procedure

1. By default, IEEE1588 is kept disabled in DPDK config file. To enable, set 'CONFIG_RTE_LIBRTE_IEEE1588' as `true` in `config/arm/meson.build` file.
2. Build DPDK using steps mentioned in Build DPDK section.
3. 'dpdk-ptpclient' executable will be generated in `arm64-build/examples` directory

9.1.11.4.3 Test setup and prerequisite to test with ptpclient

dpdk-ptpclient test application works in slave mode. It can be tested with ptp4l test application running in Linux on another machine in master mode.

Two machines are required to be connected back-to-back:

- **Tester Machine**: to run ptp4l test application. ptp4l test application can be directly installed on tester machine via apt-install type commands or can be built by downloading linuxptp package.
- **DUT (Board NXP platform)**: to run ptp4client test application.

DPAA2 port of DUT board on which ptpclient test application will be tested should be connected to one of Ethernet port of Tester Machine (Tester_port).

9.1.11.4.4 Test procedure with ptpclient

1. **Tester machine**: Ensure the Tester_port is up and connected to DPAA2_port of DUT board. Confirm this by testing ping. If the tester machine is NXP DPAA2 based board and the Tester_port does not show up in ifconfig -a command, run command like below to create the interface:

   ```
   #ls-addni dpmac.1
   Created interface: eth1 (object:dpni.0, endpoint: dpmac.1)
   ```

   For more details on interface creation, see "Creating a DPAA2 network interface" subsection of "DPAA2 Quick Start Guide" section of *Layerscape Software Development Kit User Guide*.

   Start ptp server on tester machine. It acts as PTP Master. Suppose eth1 is tester_port that is connected to DUT.

   ```
   #./ptp4l -i eth1  -m -2
   ptp4l[581.659]: selected /dev/ptp1 as PTP clock
   ptp4l[581.718]: port 1: INITIALIZING to LISTENING on INIT_COMPLETE
   ptp4l[581.718]: port 0: INITIALIZING to LISTENING on INIT_COMPLETE
   ptp4l[587.813]: port 1: LISTENING to MASTER on ANNOUNCE_RECEIPT_TIMEOUT_EXPIRES
   ptp4l[587.813]: selected local clock b26433.fffe.beb68c as best master
   ptp4l[587.813]: assuming the grand master
   role
   ```

2. **DUT machine**: This machine acts as ptp slave. Create DPRTC instance and attach DPAA2 port to DPDK using dynamic_dpl script.

   ```
   #export
   DPRTC_COUNT=1

   #source ./dynamic_dpl.sh dpmac.1
   ```

   Confirm from dynamic_dps.sh output that one DPRTC object is created.

   Run ptpclient application on DUT_port

a. Synchronize DUT PTP Hardware clock with Tester Machine PTP Hardware clock:

```
#./ptpclient -l 1 -n 1 -- -p 0x1 -T 0
```

b. Synchronize DUT PTP Hardware clock with Tester Machine PTP Hardware clock and additionally update system kernel clock:

```
#./ptpclient -l 1 -n 1 -- -p 0x1 -T 1
```

c. Verify if System kernel clock is updated, read time before and after execution of above ptpclient command using date command

```
#date
```

```
root@localhost:~# export DPRTC_COUNT=1
root@localhost:~# source ./dynamic_dpl.sh dpmac.5
parent - dprc.1
Creating Non nested DPRC
NEW DPRCs
dprc.1
dprc.2
Using board type as 2088
Using High Performance Buffers
#################### Container  dprc.2  is created ####################
Container dprc.2 have following resources :=>
 * 1 DPMCP
 * 16 DPBP
 * 8 DPCON
 * 8 DPSECI
 * 1 DPNI
 * 18 DPIO
 * 2 DPCI
 * 2 DPDMAI
 * 1 DPRTC
######################### Configured Interfaces #########################
Interface Name        Endpoint              Mac Address
==============        ========              ==================
dpni.1                dpmac.5               -Dynamic-
root@localhost:~# date
Mon Jul  1 21:41:26 UTC 2019
root@localhost:~# ./ptpclient -l 1 -n 1 -- -p 0x1 -T 0
EAL: Detected 8 lcore(s)
EAL: Detected 1 NUMA nodes
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket
fslmc: Skipping invalid device (power)
EAL: Probing VFIO support...
EAL: VFIO support initialized
EAL: PCI device 0000:01:00.0 on NUMA socket -1
EAL:   Invalid NUMA socket, default to 0
EAL:   probe driver: 8086:10d3 net_e1000_em
PMD: dpni.1: netdev created
dpaa2_net: Rx offloads non configurable - requested 0x0 ignored 0x2000
dpaa2_net: Tx offloads non configurable - requested 0x18000 ignored 0x1c000
Core 1 Waiting for SYNC packets. [Ctrl+C to quit]
Master Clock id: 32:70:3e:ff:fe:ff:a6:59
T2 - Slave  Clock.  207s 560468378ns
T1 - Master Clock.  19324s 999662036ns
T3 - Slave  Clock.  0s 0ns
```

```
T4 - Master Clock.  19324s 999702684ns
Delta between master and slave clocks:19221219448171ns

Comparison between Linux kernel Time and PTP:
Current PTP Time: Thu Jan  1 05:23:48 1970 780202685 ns
Current SYS Time: Mon Jul  1 21:42:10 2019 317847 ns
Delta between PTP and Linux Kernel time:-1561997901537542450ns
[Ctrl+C to quit]

root@localhost:~# date
Mon Jul  1 21:42:18 UTC 2019
root@localhost:~# ./ptpclient -l 1 -n 1 -- -p 0x1 -T 1
EAL: Detected 8 lcore(s)
EAL: Detected 1 NUMA nodes
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket
fslmc: Skipping invalid device (power)
EAL: Probing VFIO support...
EAL: VFIO support initialized
EAL: PCI device 0000:01:00.0 on NUMA socket -1
EAL:   Invalid NUMA socket, default to 0
EAL:   probe driver: 8086:10d3 net_e1000_em
PMD: dpni.1: netdev created
dpaa2_net: Rx offloads non configurable - requested 0x0 ignored 0x2000
dpaa2_net: Tx offloads non configurable - requested 0x18000
ignored 0x1c000

Core 1 Waiting for SYNC packets. [Ctrl+C to quit]
Master Clock id: 32:70:3e:ff:fe:ff:a6:59
T2 - Slave  Clock.  20845s 385135978ns
T1 - Master Clock.  19339s 999998152ns
T3 - Slave  Clock.  0s 0ns
T4 - Master Clock.  19340s 23532ns
Delta between master and slave clocks:8917307442853ns

Comparison between Linux kernel Time and PTP:
Current PTP Time: Thu Jan  1 08:16:02 1970 692874689 ns
Current SYS Time: Thu Jan  1 08:16:02 1970 692915 ns
Delta between PTP and Linux Kernel time:52105ns
[Ctrl+C to quit]
root@localhost:~# date
Thu Jan  1 05:16:17 UTC 1970
root@localhost:~#
```

Figure 23. LS2088ARDB DUT logs

9.2 QEMU

This section provides some use case scenarios to highlight the differences between Yocto and Ubuntu userlands. See *Layerscape Software Development Kit User Guide* for more details.

The examples provided in this section assume that the host Linux kernel is booted and has a working network interface and the following images are present in the host root filesystem:

- Guest kernel image (*/boot/Image*)
- Guest root filesystem (*/boot/fsl-image-networking-ls2088ardb.ext2.gz*)
- QEMU (*/usr/bin/qemu-system-aarch64*)

You can get these images from Yocto image, *fsl-image-networking-full*.

How to mount HugeTLB filesystem on host

1. Mount the HugeTLB filesystem on the host as follows:

```
$: echo 512 > /proc/sys/vm/nr_hugepages
$: mkdir /mnt/hugetlbfs #any mount point can be used
$: mount -t hugetlbfs none /mnt/hugetlbfs
```

How to start QEMU guest

1. Start QEMU guest:

```
$: qemu-system-aarch64 -smp 8 -m 1024 -mem-path /mnt/hugetlbfs -cpu host -machine type=virt,gic-
version=3 -kernel /boot/Image -enable-kvm -display none -serial tcp::4446,server,telnet -
initrd /boot/fsl-image-networking-ls2088ardb.ext2.gz -append 'root=/dev/ram rw console=ttyAMA0
ramdisk_size=1000000 rootwait earlyprintk' -monitor stdio
```

2. Connect to QEMU via Telnet to start booting the virtual machine (in this example, the IP address of the target board is 192.168.4.100):

```
$: telnet 192.168.4.100 4446
```

NOTE

ls1043a and ls1046a do not have a GICv3 implementation, rather they have GICv2 implementation. For those platforms, `gic-version=3`should be omitted (see *Layerscape Software Development Kit User Guide* for more details).

How to use virtual network interfaces using virtio

1. Enable virtio networking in the host and guest Linux kernels.
2. On the host Linux system, create a bridge to the physical network interface to be used by the virtual network interface in the virtual machine:

```
$: brctl addbr br0
$: ifconfig br0 192.168.3.30 netmask 255.255.248.0
$: ifconfig eth2 0.0.0.0
$: brctl addif br0 eth2
```

3. Create a qemu-ifup script on the host Linux system:

```
#!/bin/sh
# TAP interface will be passed in $1
bridge=br0
guest_device=$1
ifconfig $guest_device 0.0.0.0 up
brctl addif $bridge $guest_device
```

4. When starting QEMU, specify that the network device type is “virtio” and specify the path to the script created in step 3:

```
$: qemu-system-aarch64 -smp 8 -m 1024 -mem-path /mnt/hugetlbfs -cpu host -machine type=virt,gic-
version=3 -kernel /boot/Image -enable-kvm -display none -seriatrd /boot/fsl-image-networking-
ls2088ardb.ext2.gz -netdev tap,id=tap0,script=qemu-ifup,downscript=no,ifname="tap0" -device
virtio-net-pci,netdev=tap0 -append 'root=/dev/ram r=1000000 rootwait earlyprintk' -monitor stdio
```

If you want to use vhost-net with virtio, then perform the above steps and add the `vhost=on` parameter as follows:

```
$: qemu-system-aarch64 -smp 8 -m 1024 -mem-path /mnt/hugetlbfs -cpu host
-machine type=virt,gic-version=3 -kernel /boot/Image -enable-kvm -display none -
serial tcp::4446,server,telnet -initrd /boot/fsl-image-networking-ls2088ardb.ext2.gz -netdev
tap,id=tap0,script=qemu-ifup,downscript=no,ifname="tap0",vhost=on -device virtio-net-pci,netdev=tap0 -
append 'root=/dev/ram rw console=ttyAMA0 ramdisk_size=1000000 rootwait earlyprintk' -monitor stdio
```

How to use virtual disks using virtio

1. On the host Linux system, create a binary image to represent the guest disk:

   ```
   $: dd if =/dev/zero of=my_guest_disk bs=4K count=4K
   ```

2. Start QEMU, specifying the name of the virtual disk file for the `-drive` argument:

   ```
   $: qemu-system-aarch64 -smp 8 -m 1024 -mem-path /mnt/hugetlbfs -cpu host
   -machine type=virt,gic-version=3 -kernel /boot/Image -enable-kvm -display none -
   serial tcp::4446,server,telnet -initrd /boot/fsl-image-networking-ls2088ardb.ext2.gz -drive
   if=none,file=my_guest_disk,cache=none,id=user,format=raw -device virtio-blk-pci,drive=user -
   append 'root=/dev/ram rw console=ttyAMA0 ramdisk_size=1000000 rootwait earlyprintk' -monitor
   stdio
   ```

How to use virtual disks using virtio-blk data plane

1. Start QEMU as follows:

   ```
   $: qemu-system-aarch64 -smp 8 -m 1024 -mem-path /mnt/hugetlbfs -cpu host -machine type=virt,gic-
   version=3 -kernel /boot/Image -enable-kvm -display none -serial tcp::4446,server,telnet
   -initrd /boot/fsl-image-networking-ls2088ardb.ext2.gz -object iothread,id=iothread0 -
   drive if=none,file=/dev/mmcblk0,cache=none,id=drive0,format=raw,aio=native -device virtio-blk-
   pci,drive=drive0,scsi=off,iothread=iothread0 -append 'root=/dev/ram rw console=ttyAMA0
   ramdisk_size=1000000 rootwait earlyprintk' -monitor stdio
   ```

How to use DPAA2 direct assignment

1. Create the child container that will be assigned to the guest virtual machine and bind it to the vfio-fsl-mc driver.
2. Add the following `device` command to the QEMU command line:

   ```
   -device vfio-fsl-mc,host=dprc.2
   ```

3. Start QEMU as follows:

   ```
   $: qemu-system-aarch64 -smp 8 -m 1024 -mem-path /mnt/hugetlbfs -cpu host -machine type=virt,gic-
   version=3 -kernel /boot/Image -enable-kvm -display none -serial tcp::4446,server,telnet -
   initrd /boot/fsl-image-networking-ls2088ardb.ext2.gz -device vfio-fsl-mc,host=dprc.2 -append
   'root=/dev/ram rw console=ttyAMA0 ramdisk_size=1000000 rootwait earlyprintk' -monitor stdio -S
   ```

4. Assign each vCPU thread to one physical CPU only:
   a. Get the virtual machine thread IDs by entering QEMU shell:

      ```
      (qemu) info cpus
      ```

   b. Assign one vCPU thread to one core only using *taskset*.

c. Start the vCPU threads:

```
(qemu) c
```

How to use PCIe direct assignment

1. Select the PCIe device that will be assigned to the virtual machine. For example, PCIe device is e1000e PCIe network device (0000.01.00.0).
2. Bind the PCIe device to the VFIO driver.
3. List all devices in the same iommu-group and bind them to VFIO using step 1.
4. Add the following `device` command to the QEMU command line for all the devices in the iommu-group:

```
-device vfio-pci,host=0000:01:00.0
```

5. Start QEMU as follows:

```
$: qemu-system-aarch64 -smp 8 -m 1024 -mem-path /mnt/hugetlbfs -cpu host -machine type=virt,gic-
version=3 -kernel /boot/Image -enable-kvm -display none -serial tcp::4446,server,telnet -
initrd /boot/fsl-image-networking-ls2088ardb.ext2.gz -device vfio-pci,host=0000:01:00.0 -append
'root=/dev/ram rw console=ttyAMA0 ramdisk_size=1000000 rootwait earlyprintk' -monitor stdio
```

How to use pass-through of USB devices

Use one of the following two approaches for passing through a USB device:

- By specifying USB vendor ID and product ID of the device:
  1. Specify the USB vendor ID and the product ID of the device (identify the information using `lsusb`):

```
-device nec-usb-xhci,id=xhci
-device usb-host,bus=xhci.0,vendorid=0x13fe,productid=0x3600
```

  2. Start QEMU as follows:

```
$: qemu-system-aarch64 -smp 8 -m 1024 -mem-path /mnt/hugetlbfs -cpu host -
machine type=virt,gic-version=3 -kernel /boot/Image -enable-kvm -display none -serial
tcp::4446,server,telnet -initrd /boot/fsl-image-networking-ls2088ardb.ext2.gz -device nec-
usb-xhci,id=xhci -device usb-host,bus=xhci.0,vendorid=0x13fe,productid=0x3600 -append
'root=/dev/ram rw console=ttyAMA0 ramdisk_size=1000000 rootwait earlyprintk' -monitor stdio
```

- By specifying USB bus and port number:
  1. Specify the USB bus and the port number (identify the information using `lsusb -t`):

```
-device nec-usb-xhci,id=xhci
-device usb-host,bus=xhci.0,hostbus=1,hostport=1
```

  2. Start QEMU as follows:

```
$: qemu-system-aarch64 -smp 8 -m 1024 -mem-path /mnt/hugetlbfs -cpu host -
machine type=virt,gic-version=3 -kernel /boot/Image -enable-kvm -display none -serial
tcp::4446,server,telnet -initrd /boot/fsl-image-networking-ls2088ardb.ext2.gz -device nec-
usb-xhci,id=xhci -device usb-host,bus=xhci.0,hostbus=1,hostport=1 -append 'root=/dev/ram rw
console=ttyAMA0 ramdisk_size=1000000 rootwait earlyprintk' -monitor stdio
```

10 Frequently asked questions

Question 1: How can I compile ATF with OPTEE?

Answer: For compiling ATF with OPTEE, add the below line in your conf/local.conf file:

```
DISTRO_FEATURES_append = " optee"
```

Question 2: How can I compile ATF with RCW image that is different from default image?

Answer: Modify RCWNOR or RCWSD and RCWNAND in <machine>.conf file. For example, update ls1043ardb.conf file in meta-freescale layer as follows:

```
RCWNOR ?= "RR_FQPP_1455/rcw_1600"
```

Question 3: How can I add custom additional packages to default rootfs?

Answer: Set IMAGE_INSTALL_append in the conf/local.conf file.

11 Related resources

The table below provides resources available for reference.

Table 14. Related resources

Resource	Resource link
NXP LSDK official website	https://www.nxp.com/lsdk
Layerscape Software Development Kit User Guide	https://www.nxp.com/design/software/embedded-software/linux-software-and-development-tools/layerscape-software-development-kit-v19.09:LAYERSCAPE-SDK?tab=Documentation_Tab
NXP LSDK Github portal	https://lsdk.github.io/
Yocto Open Source User Guide	https://www.yoctoproject.org/docs/

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Limited warranty and liability — Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

Right to make changes - NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Security — Customer understands that all NXP products may be subject to unidentified or documented vulnerabilities. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer’s applications and products. Customer’s responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer’s applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately. Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, µVision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© NXP B.V. 2021. All rights reserved.

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 04 October 2021
Document identifier: LSDKYOCTOUG