MARVELL

White Paper

Marvell OCTEON TX2 DPDK – Getting Started

November 2020

Executive Summary

The OCTEON TX2™ SoC family is the sixth generation of Marvell's OCTEON® multi-core infrastructure processors. OCTEON multi-core SoC processors are the industry's most scalable, high-performance, and power-efficient processing solutions for intelligent networking and security applications.

The OCTEON TX2 SoC family scales from multi-10Gbps to multi-100Gbps packet and security processing. It integrates Armv8.2-based CPU cores which run up to 2.4GHz and offers highly-comprehensive and flexible hardware accelerations for packet and security processing. The integrated hardware accelerators offer throughput, latency, deterministic performance and efficiency advantages for various wired and wireless networking and security applications optimized for processing both virtual and physical network functions (see Figure 1).

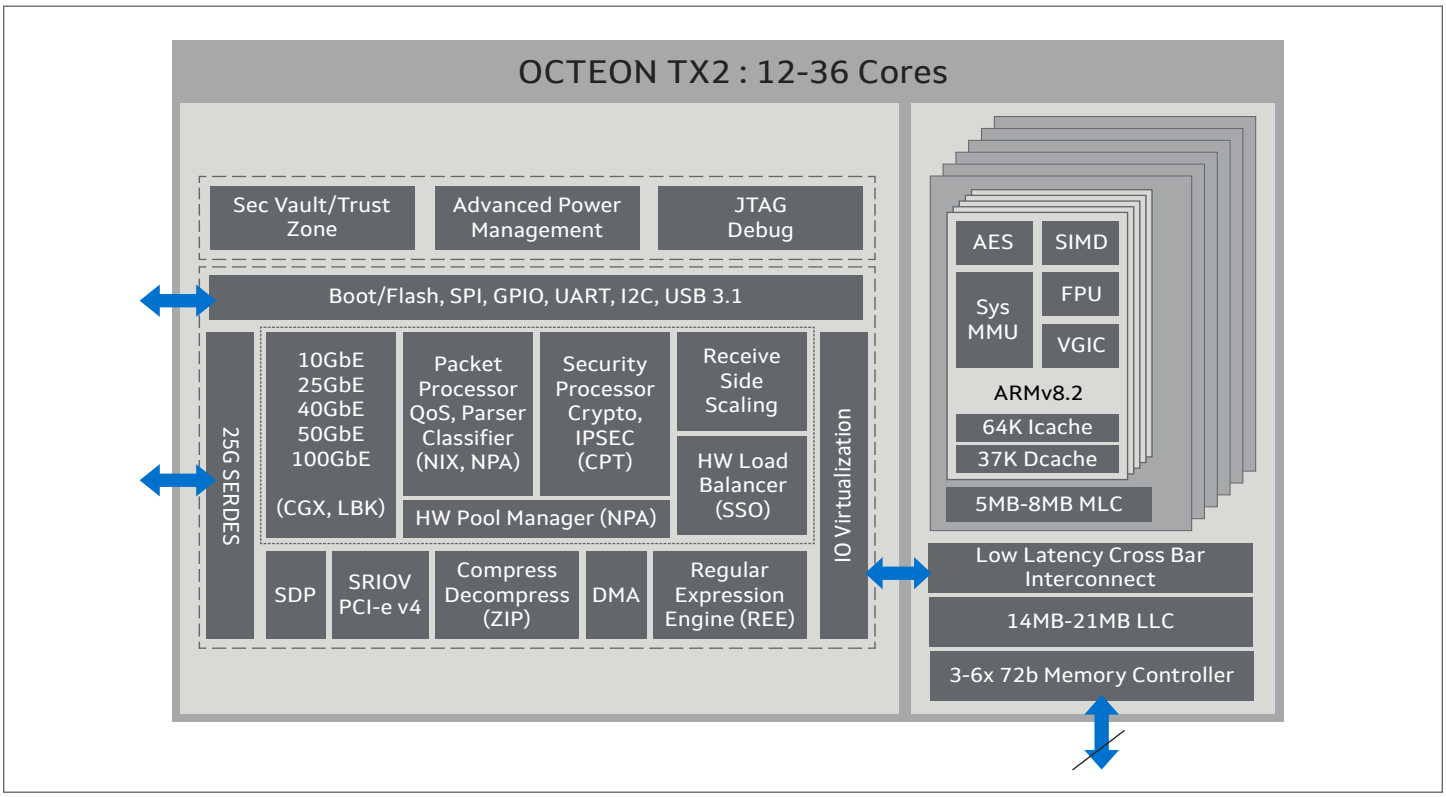

Figure 1: OCTEON TX2 Highlevel Block Diagram

To ensure that developers can successfully implement the OCTEON TX2 advanced packet-processing and hardware accelerators, the OCTEON TX2 Software Development Kit (SDK) includes a Data Plane Development Kit (DPDK) software package. The DPDK abstracts the OCTEON TX2 networking and security capabilities, including advanced hardware accelerators available for optimizing networking and security applications.

This white paper provides an overview of how to install and configure the DPDK software on the OCTEON TX2.

- This document assumes that the OCTEON TX2 board is updated with the most recent OCTEON TX2 SDK release. The intent of this document is to get the user familiar with OCTEON TX2 DPDK. It contains instructions for installing and configuring DPDK on the OCTEON TX2, as well as running a sample DPDK application.
- This document assumes that the OCTEON TX2 board is updated with the most recent SDK release. Please contact your sales representative for information about accessing the most recent SDK release.

Getting Started with DPDK

Users can build a DPDK from sources or use the SDK pre-built DPDK binaries in the Buildroot file system to develop and run DPDK applications. The OCTEON TX2 SDK is shipped with both DPDK sources and with a pre-built DPDK environment.

OCTEON TX2 SDK include all files and utilities needed to build and run the DPDK. You can build the DPDK with a ready script included in the SDK, or manually build the DPDK from sources. This section explains both methods.

Building DPDK from Sources – Prerequisite

The following archives are required to manually build the DPDK sources and to build the DPDK sources with the ready script included in the SDK:

- base-sources-<release-id>.tar.bz2
- toolchain.tar.bz2

The following archived files are required only for building the DPDK sources with the ready script included in the SDK:

- sources-sdk-base -<release-id>.tar.bz2
- sources-buildroot-<release-id>.tar.bz2

Building DPDK from Sources using SDK Ready Script

To build the DPDK with the SDK ready script, extract each of the following archives into the working directory of the x86 host. Use the following commands to extract the archives:

```
tar xvf base-sources-<release-id>.tar.bz2
tar xvf sources-sdk-base-<release-id>.tar.bz2
tar xvf sources-buildroot-<release-id>.tar.bz2
tar xvf toolchain.tar.bz2
```

After all archives are extracted, use the following commands to build a DPDK using the SDK included script:

```
cd sdk-base
./scripts/ci/compile.sh cn96xx -r <release-id> -p dpdk
```

After the build script is completed the cn96xx-release-output directory is created and contains the following files needed to develop and run DPDK applications:

```
cn96xx-release-output
└── target
    └── usr
        ├── bin
        │   ├── dpdk-eventdev_pipeline
        │   ├── dpdk-ipsec-secgw
        │   ├── dpdk-l2fwd
        │   ├── dpdk-l3fwd
        │   ├── dpdk-pdump
        │   ├── dpdk-pmdinfo
        │   ├── dpdk-procinfo
        │   ├── dpdk-test
        │   ├── dpdk-test-compress-perf
        │   ├── dpdk-test-crypto-perf
        │   └── dpdk-test-eventdev
        └── sbin
            └── dpdk-devbind
```

```
cn96xx-release-output
└── target
    └── usr
        ├── lib
        │   └── modules
        │       └── 4.14.76-10.0.0
        │           └── extra
        │               └── dpdk
        │                   └── rte_kni.ko
        └── share
            └── dpdk
                └── usertools
                    ├── dpdk-devbind.py
                    ├── dpdk-telemetry-client.py
                    ├── dpdk-setup.sh
                    └── dpdk-pmdinfo.py
```

Building DPDK Manually from Sources

To build the DPDK manually from sources, follow the steps described below:

1. Extract the Archives

   Use the following commands to extract each of the archives below into the working directory of the x86 host:

   ```
   tar xvf base-sources-<release-id>.tar.bz2
   tar xvf toolchain.tar.bz2
   ```

2. Prepare the toolchain to cross-compile the DPDK sources.In this step you extract the toolchain itself and set the CROSS and the CROSS_COMPILE environment variables:

```
cd toolchain
tar xvf marvell-tools-233.0.tar.bz2
export CROSS=$(pwd)/marvell-tools-233.0/bin/aarch64-marvell-linux-gnu-
export CROSS_COMPILE=$(pwd)/marvell-tools-233.0/bin/aarch64-marvell-linux-gnu-
```

3. Build the Linux kernel

   Some DPDK components require Linux® kernel headers. In the extracted base-sources-<release-id> directory, navigate to the Linux directory and build the Linux kernel:

   ```
   cd base-sources-<release-id>/linux
   export ARCH=arm64
   export RTE_KERNELDIR=$(pwd)
   make marvell_v8_sdk_defconfig
   make
   ```

4. Extract and Build DPDK sources

In the extracted sdk-sources-<release-id> directory, navigate to the dpdk folder and extract the dpdk source as shown below:

```
cd ../dpdk
tar xvf sources-dpdk-19.08-<release-id>.tar.bz2
cd dpdk-19.08-<release-id>
```

Now you must install the DPDK target environment. Set the following environment variables:

```
export RTE_SDK=$(pwd)
export RTE_TARGET=arm64-octeontx2-linuxapp-gcc
export DESTDIR=output
```

Now the environment is ready to be installed. Run the following command to install the target environment:

```
make install T=arm64-octeontx2-linuxapp-gcc
```

Once completed, you can find the desired files in the output directory:

```
output
 ├── bin
 │   ├── dpdk-pdump
 │   ├── dpdk-pmdinfo
 │   ├── dpdk-procinfo
 │   ├── dpdk-test-compress-perf
 │   ├── dpdk-test-crypto-perf
 │   └── dpdk-test-eventdev
 └── sbin
     └── dpdk-devbind
```

```
output
 ├── share
 │    └── dpdk
 │        └── usertools
 │            ├── dpdk-devbind.py
 │            ├── dpdk-pmdinfo.py
 │            ├── dpdk-setup.sh
 │            └── telemetry-client.py
 └── $RTE_KERNELDIR
      └── rte_kni.ko
```

5. Building sample applications

The output directory, shown above, does not include many of the sample applications generated as a result of building the sources using the ready script included in the SDK. The commands below are an example of how to build a sample l2fwd DPDK application:

```
cd examples
cd l2fwd
cd make
       CC main.o
       LD l2fwd
       INSTALL-APP l2fwd
    INSTALL-MAP
2fwd.map
    ls build/app
       l2fwd  l2fwd.map
```

Using DPDK Pre-built Environment

Users can use the pre-built DPDK environment instead of building the DPDK from sources. The SDK includes a ready DPDK environment in the prebuilt root filesystem. When installing the SDK and running the OCTEON TX2 board, the user can find all the DPDK-ready environment-related files as seen below:

```
/usr
 ├── bin
 │   ├── dpdk-eventdev_pipeline
 │   ├── dpdk-ipsec-secgw
 │   ├── dpdk-l2fwd
 │   ├── dpdk-l3fwd
 │   ├── dpdk-pdump
 │   ├── dpdk-pmdinfo
 │   ├── dpdk-procinfo
 │   ├── dpdk-test
 │   ├── dpdk-test-compress-perf
 │   ├── dpdk-test-crypto-perf
 │   ├── dpdk-test-eventdev
 │   └── dpdk-pktgen
 └── sbin
     └── dpdk-devbind
```

```
/usr
 ├── lib
 │   └── modules
 │       └── 4.14.76-8.0.0
 │           └── extra
 │               └── dpdk
 │                   └── rte_kni.ko
 └── share
     └── dpdk
         └── usertools
             ├── dpdk-devbind.py
             ├── dpdk-telemetry-client.py
             ├── dpdk-setup.sh
             └── dpdk-pmdinfo.py
```

Running DPDK l2fwd Sample Application

This section describes what you need to run a DPDK l2fwd sample application and includes instructions to set up the DPDK environment for a sample application to obtain access to the OCTEON TX2 network interfaces. The sample applications should run effectively in an optimized run environment by enabling Linux cores isolation and Hugepages support.

Binding Ethernet Devices to DPDK

By default, OCTEON TX2 ethernet devices are bound to the Linux kernel. For example eth0, by default, is bound to the octeontx2-nicpf kernel driver. If the DPDK application uses eth0 then it should obtain access to that interface by binding eth0 to vfio-pci or uio_pci_generic drivers that expose direct device access to userspace. To do this, run dpdk-devbind -s. You can view the status of all network ports and the drivers, as shown below:

```
# dpdk-devbind -s
Network devices using kernel driver
===================================
0002:02:00.0 'Device a063' if=eth0 drv=octeontx2-nicpf unused=vfio-pci,uio_pci_generic
0002:03:00.0 'Device a063' if=eth1 drv=octeontx2-nicpf unused=vfio-pci,uio_pci_generic
0002:04:00.0 'Device a063' if=eth2 drv=octeontx2-nicpf unused=vfio-pci,uio_pci_generic
0002:05:00.0 'Device a063' if=eth3 drv=octeontx2-nicpf unused=vfio-pci,uio_pci_generic
0002:06:00.0 'Device a063' if=eth4 drv=octeontx2-nicpf unused=vfio-pci,uio_pci_generic
0002:07:00.0 'Device a063' if=eth5 drv=octeontx2-nicpf unused=vfio-pci,uio_pci_generic
0002:08:00.0 'Device a063' if=eth6 drv=octeontx2-nicpf unused=vfio-pci,uio_pci_generic
0002:09:00.0 'Device a063' if=eth7 drv=octeontx2-nicpf unused=vfio-pci,uio_pci_generic
```

UIO is a simple way to set up the device, map device memory to a user space, and register interrupts using LINUX. VFIO is a more robust and secure framework compared to UIO. VFIO exposes direct device access to a userspace in a secure, IOMMU-protected environment which allows safe, non-privileged, userspace drivers (e.g. DPDK drivers). OCTEON TX2 DPDK uses VFIO.

For proper operation of VFIO when running DPDK applications as a non-privileged user, set up appropriate permissions with the DPDK setup script: dpdk-setup.sh at the following location:

```
/usr/share/dpdk/usertools).
```

Proper operation of DPDK l2fwd sample application requires that the relevant ethernet interfaces be unbound from any driver they are using, and then bound to DPDK-compatible drivers as shown in the example commands below:

```
dpdk-devbind -u 0002:02:00.0 0002:06:00.0
dpdk-devbind -b vfio-pci 0002:02:00.0 0002:06:00.0
```

After binding 0002:02:00.0 and 0002:06:00.0 to DPDK, you can view the status of all network ports and the drivers being used by running dpdk-devbind -s , as shown below:

```
# dpdk-devbind -s
Network devices using kernel driver
===================================
0002:02:00.0 'Device a063' if=eth0 drv=octeontx2-nicpf unused=vfio-pci,uio_pci_generic
0002:03:00.0 'Device a063' if=eth1 drv=octeontx2-nicpf unused=vfio-pci,uio_pci_generic
0002:04:00.0 'Device a063' if=eth2 drv=octeontx2-nicpf unused=vfio-pci,uio_pci_generic
0002:05:00.0 'Device a063' if=eth3 drv=octeontx2-nicpf unused=vfio-pci,uio_pci_generic
0002:06:00.0 'Device a063' if=eth4 drv=octeontx2-nicpf unused=vfio-pci,uio_pci_generic
0002:07:00.0 'Device a063' if=eth5 drv=octeontx2-nicpf unused=vfio-pci,uio_pci_generic
0002:08:00.0 'Device a063' if=eth6 drv=octeontx2-nicpf unused=vfio-pci,uio_pci_generic
0002:09:00.0 'Device a063' if=eth7 drv=octeontx2-nicpf unused=vfio-pci,uio_pci_generic
```

Enabling Huge Pages

To efficiently run DPDK applications, you must initialize hugepages. You can see the possible sizes of hugepages in the directory /sys/kernel/mm/hugepages. To initialize hugepages you must create a mountpoint and then mount hugetlbfs.

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

Next use echo to specify the number of pages you want to allocate, in this case 12 hugepages of size 524288kB.

```
echo 12 > /sys/kernel/mm/hugepages/hugepages-524288kB/nr_hugepages
```

Enabling Linux Cores Isolation

You can isolate several CPU cores from use in Linux and reserve them for DPDK applicationsby adding the `isolcpus` arguments to `bootargs` in U-Boot. To do so, while the board is booting up, interrupt it by pressing any key when counting down at the point shown below:

```
U-Boot 2018.03-8.0.0 (Sep 07 2019 - 18:14:29 +0000), Build: SDK10.0-PR2006

OcteonTX2 CN96XX Arm V8 Core
Board: cn96xx-crb
DRAM:  47.9 GiB
MMC:   octeontx-mmc0: 0, octeontx-mmc1: 1
Loading Environment from SPI Flash... SF: Detected mx25l25635f with page size 256 Bytes, erase size
64 KiB, total 32 MiB
OK
In:    serial
Out:   serial
Err:   serial
CGX0 LMAC0 [40G_R]
CGX1 LMAC0 [10G_R]
CGX1 LMAC1 [10G_R]
CGX1 LMAC2 [10G_R]
CGX2 LMAC0 [10G_R]
CGX2 LMAC1 [10G_R]
CGX2 LMAC2 [10G_R]
CGX2 LMAC3 [10G_R]
Net:   eth0: rvu_pf#0, eth1: rvu_pf#1, eth2: rvu_pf#2, eth3: rvu_pf#3, eth4: rvu_pf#4, eth5: rvu_
pf#5, eth6: rvu_pf#6, eth7: rvu_pf#7
Hit any key to stop autoboot:  0
```

Once in U-Boot, you can add isolcpus to bootargs as follows (this example isolates CPU cores 1-23):

```
print bootargs
bootargs=ttyAMA0,115200n8 earlycon=pl011,0x87e028000000 debug maxcpus=24 rootwait rw root=/dev/
mmcblk0p2 coherent_pool=16M net.ifnames=0
edit bootargs
edit: bootargs=ttyAMA0,115200n8 earlycon=pl011,0x87e028000000 debug maxcpus=24 rootwait rw root=/
dev/sda1 coherent_pool=16M net.ifnames=0  nohz_full=1-23 cpuidle.off=1  isolcpus=1-23 hugepages=12
hugepagesz=512M
saveenv
```

Running DPDK l2fw Application

In this example, IXIA is a traffic generator and an OCTEON TX2 CRB is the DUT with two 40G connections to the traffic generator. The CN96 CRB uses ports `0002:02:00.0` and `0002:06:00.0` as shown below:

```
 ------------------------------
| CN96XX CRB                   |
|                              |
| 0002:02:00.0     0002:06:00.0 |
 ------------------------------
       /\                  /\
       ||                  ||
       \/                  \/
 ------------------------------
|     (40G)               (40G) |
|                               |
| XENA                          |
 ------------------------------
```

After Binding eth2 and eth3 to the DPDK, enabling hugepages and Linux Cores isolation as described above, run the Layer 2 forward application using a coremask of 3 (cores 0 & 1) and a portmask of 3 (ports 0 & 1) with the following commands:

dpdk-devbind.py -u 0002:02:00.0 0002:06:00.0

```
testpmd -c 0xc00000 -n 4 -- --nb-cores=1 --port-topology=loop --rxq=1 --txq=1 --portmask=0x3 --rss-ip --no-flush-rx
```

Output:

```
# /usr/bin/testpmd -c 0xc00000 -n 4 -- --nb-cores=1 --port-topology=loop --rxq=1 --txq=1 --portmas-k=0x3 --rss-ip --no-flush-rx
EAL: Detected 24 lcore(s)
EAL: Detected 1 NUMA nodes
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket
EAL: Selected IOVA mode 'VA'
EAL: No available hugepages reported in hugepages-2048kB
EAL: Probing VFIO support...
EAL: PCI device 0002:01:00.1 on NUMA socket 0
EAL:   probe driver: 177d:a0f8 net_octeontx2
EAL: PCI device 0002:01:00.2 on NUMA socket 0
EAL:   probe driver: 177d:a0f8 net_octeontx2
EAL: PCI device 0002:01:00.3 on NUMA socket 0
EAL:   probe driver: 177d:a0f8 net_octeontx2
EAL: PCI device 0002:01:00.4 on NUMA socket 0
EAL:   probe driver: 177d:a0f8 net_octeontx2
EAL: PCI device 0002:01:00.5 on NUMA socket 0
EAL:   probe driver: 177d:a0f8 net_octeontx2
EAL: PCI device 0002:01:00.6 on NUMA socket 0
EAL:   probe driver: 177d:a0f8 net_octeontx2
EAL: PCI device 0002:01:00.7 on NUMA socket 0
EAL:   probe driver: 177d:a0f8 net_octeontx2
EAL: PCI device 0002:01:01.0 on NUMA socket 0
EAL:   probe driver: 177d:a0f8 net_octeontx2
```

```
EAL: PCI device 0002:01:01.1 on NUMA socket 0
EAL:   probe driver: 177d:a0f8 net_octeontx2
EAL: PCI device 0002:01:01.2 on NUMA socket 0
EAL:   probe driver: 177d:a0f8 net_octeontx2
EAL: PCI device 0002:01:01.3 on NUMA socket 0
EAL:   probe driver: 177d:a0f8 net_octeontx2
EAL: PCI device 0002:01:01.4 on NUMA socket 0
EAL:   probe driver: 177d:a0f8 net_octeontx2
EAL: PCI device 0002:01:01.5 on NUMA socket 0
EAL:   probe driver: 177d:a0f8 net_octeontx2
EAL: PCI device 0002:01:01.6 on NUMA socket 0
EAL:   probe driver: 177d:a0f8 net_octeontx2
EAL: PCI device 0002:01:01.7 on NUMA socket 0
EAL:   probe driver: 177d:a0f8 net_octeontx2
EAL: PCI device 0002:01:02.0 on NUMA socket 0
EAL:   probe driver: 177d:a0f8 net_octeontx2
EAL: PCI device 0002:02:00.0 on NUMA socket 0
EAL:   probe driver: 177d:a063 net_octeontx2
EAL:   using IOMMU type 1 (Type 1)
EAL: PCI device 0002:03:00.0 on NUMA socket 0
EAL:   probe driver: 177d:a063 net_octeontx2
EAL: PCI device 0002:04:00.0 on NUMA socket 0
EAL:   probe driver: 177d:a063 net_octeontx2
EAL: PCI device 0002:05:00.0 on NUMA socket 0
EAL:   probe driver: 177d:a063 net_octeontx2
EAL: PCI device 0002:06:00.0 on NUMA socket 0
EAL:   probe driver: 177d:a063 net_octeontx2
EAL: PCI device 0002:07:00.0 on NUMA socket 0
EAL:   probe driver: 177d:a0f9 event_octeontx2
EAL: PCI device 0002:08:00.0 on NUMA socket 0
EAL:   probe driver: 177d:a0f9 event_octeontx2
EAL: PCI device 0002:09:00.0 on NUMA socket 0
EAL:   probe driver: 177d:a0f9 event_octeontx2
EAL: PCI device 0002:0a:00.0 on NUMA socket 0
EAL:   probe driver: 177d:a0f9 event_octeontx2
EAL: PCI device 0002:0b:00.0 on NUMA socket 0
EAL:   probe driver: 177d:a0f9 event_octeontx2
EAL: PCI device 0002:0c:00.0 on NUMA socket 0
EAL:   probe driver: 177d:a0fb mempool_octeontx2
EAL: PCI device 0002:0d:00.0 on NUMA socket 0
EAL:   probe driver: 177d:a0fb mempool_octeontx2
EAL: PCI device 0002:0e:00.0 on NUMA socket 0
EAL:   probe driver: 177d:a0f9 event_octeontx2
EAL: PCI device 0002:0f:00.0 on NUMA socket 0
EAL:   probe driver: 177d:a0fb mempool_octeontx2
testpmd: create a new mbuf pool <mbuf_pool_socket_0>: n=155446, size=2176, socket=0
testpmd: preferred mempool ops selected: octeontx2_npa
Configuring Port 0 (socket 0)
PMD: Port 0: Link Up - speed 40000 Mpbps - full-duplex
```

```
Port 0: link state change event
Port 0: 00:0F:B7:06:0E:A2
Configuring Port 1 (socket 0)
PMD: Port 1: Link Up - speed 40000 Mbps - full-duplex

Port 1: link state change event
Port 1: 00:0F:B7:06:0E:A6
Checking link statuses...
Done
No commandline core given, start packet forwarding
io packet forwarding - ports=2 - cores=1 - streams=2 - NUMA support enabled, MP allocation mode:
native
Logical Core 23 (socket 0) forwards packets on 2 streams:
  RX P=0/Q=0 (socket 0) -> TX P=0/Q=0 (socket 0) peer=02:00:00:00:00:00
  RX P=1/Q=0 (socket 0) -> TX P=1/Q=0 (socket 0) peer=02:00:00:00:00:01

  io packet forwarding packets/burts=32
  nb forwarding cores=1 - nb forwarding ports=2
  port 0: RX queue number: 1 Tx queue number: 1
    Rx offloads=0x0 Tx offloads=0x10000
    RX queue: 0
      RX desc=4096 - RX free threshold=0
      RX threshold registers: pthres=0 hthresh=0  wthresh=0
      RX Offloads=0x0
    TX queue: 0
      TX desc=512 - TX free threshold=0
      TX threshold registers: pthresh=0 hthresh=0  wthresh=0
      TX offloads=0x10000 - TX RS bit threshold=0
  port 1: RX queue number: 1 Tx queue number: 1
    Rx offloads=0x0 Tx offloads=0x10000
    RX queue: 0
      RX desc=4096 - RX free threshold=0
      RX threshold registers: pthres=0 hthresh=0  wthresh=0
      RX Offloads=0x0
    TX queue: 0
      TX desc=512 - TX free threshold=0
      TX threshold registers: pthresh=0 hthresh=0  wthresh=0
      TX offloads=0x10000 - TX RS bit threshold=0
Press enter to exit
```

These commands configure the traffic generator such that:

- the first stream has a destination address of 00:0F:B7:06:0E:A2 and a source address of 02:00:00:00:00:00
- the second stream has a destination address of 00:0f:b7:06:0E:A6 and a source address of 02:00:00:00:00:01.
- the traffic sent has a frame size fixed to 64 bytes.
- The board used for this example has a DDR speed of 3200 MTs, a CPU CLK of 2400 MHz and RCLK of 1000 MHz.