# Intel® Data Plane Development Kit - L2 Forwarding in a Virtualization Environment Sample Application

## User Guide

*April 2012*

**Intel Confidential**

Reference Number: 496039-1.0

# Contents



# Revision History

| Date | Revision | Description |
|---|---|---|
| April 2011 | 1.0 | Initial release |

# 1.0 Introduction

The L2 Forwarding in a Virtualization Environment sample application is a simple example of packet processing using the Intel® Data Plane Development Kit (Intel® DPDK) that takes advantage of Single Root I/O Virtualization (SR-IOV) features in a virtualized environment.

## 1.1 Documentation Roadmap

The following is a list of Intel® DPDK documents in suggested reading order:

- **Release Notes**: Provides release-specific information, including supported features, limitations, fixed issues, known issues and so on. Also, provides the answers to frequently asked questions in FAQ format.
- **Getting Started Guide**: Describes how to install and configure the Intel® DPDK software; designed to get users up and running quickly with the software.
- **Programmer's Guide**: Describes:
  - The software architecture and how to use it (through examples), specifically in a Linux* application (linuxapp) environment
  - The content of the Intel® DPDK, the build system (including the commands that can be used in the root Intel® DPDK Makefile to build the development kit and an application) and guidelines for porting an application
  - Optimizations used in the software and those that should be considered for new development

  A glossary of terms is also provided.
- **API Reference**: Provides detailed information about Intel® DPDK functions, data structures and other programming constructs.
- **Sample Application User Guides**: A set of guides, each describing a sample application that showcases specific functionality, together with instructions on how to compile, run and use the sample application.

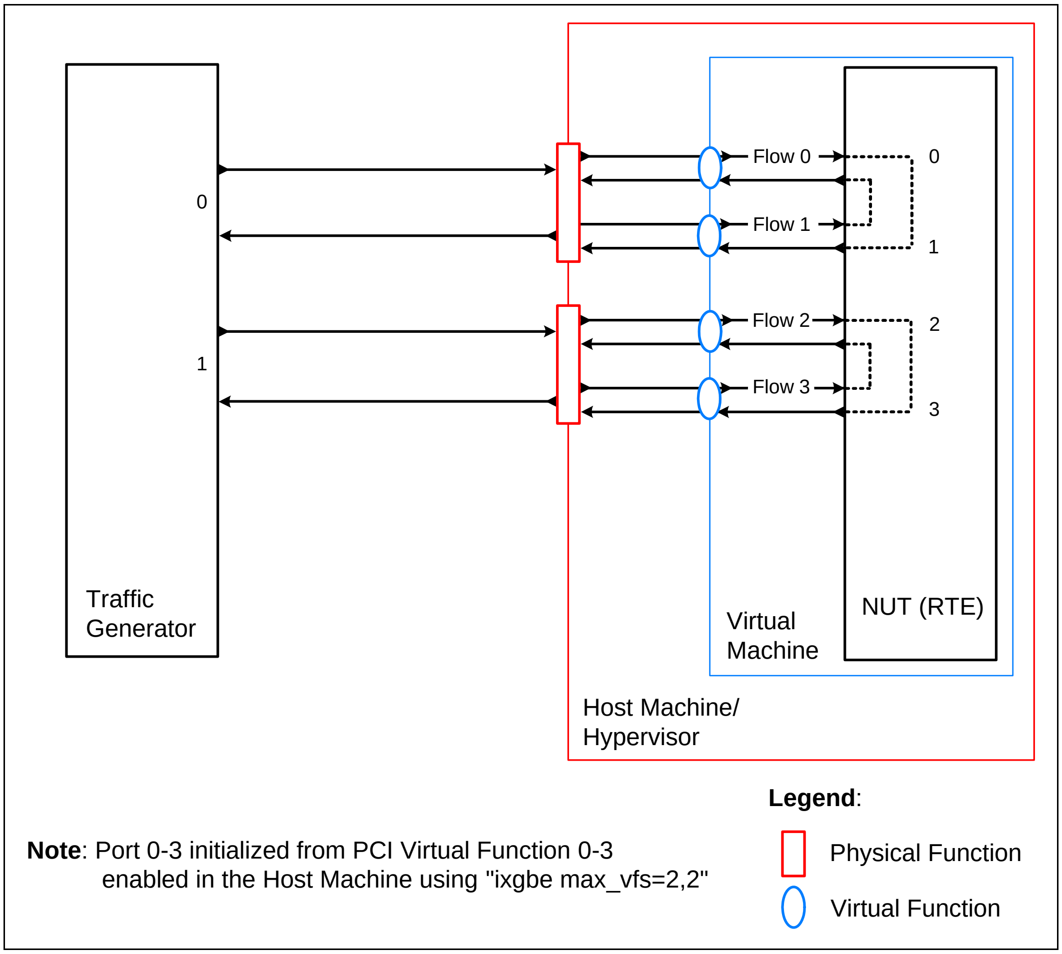
# 2.0 Overview

The L2 Forwarding in a Virtualization Environment sample application performs L2 forwarding for each packet that is received on an RX_PORT. The destination port is the adjacent port from the enabled portmask, that is, if the first four ports are enabled (portmask 0xf), ports 1 and 2 forward into each other, and ports 3 and 4 forward into each other. Also, the MAC addresses are affected as follows:

- The source MAC address is replaced by the TX_PORT MAC address
- The destination MAC address is replaced by `00:09:c0:00:00:TX_PORT_ID`

This application can be used to benchmark performance using a traffic-generator, as shown in the following figure.

**Figure 1. Performance Benchmark Setup**




The L2 Forwarding application can also be used as a starting point for developing a new application based on Intel® DPDK.

## 2.1 Virtual Function Setup Instructions

This application specifically uses the virtual function available in the system and therefore can be used in a virtual machine without passing through the whole Network Device into a guest machine in a virtualized scenario. The virtual functions can be enabled in the host machine or the hypervisor with the respective physical function driver.

For example, in a Linux* host machine, it is possible to enable a virtual function using the following command:

```
modprobe ixgbe max_vfs=2,2
```

This command enables two Virtual Functions on each of Physical Function of the NIC, with two physical ports in the PCI configuration space. It is important to note that enabled Virtual Function 0 and 2 would belong to Physical Function 0 and Virtual Function 1 and 3 would belong to Physical Function 1, in this case enabling a total of four Virtual Functions.

## 3.0 Compiling the Application

1. Go to the example directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/l2fwd-vf
```

2. Set the target (a default target is used if not specified). For example:

```
export RTE_TARGET=x86_64-default-linuxapp-gcc
```

   See the *Intel® DPDK Getting Started Guide* for possible `RTE_TARGET` values.

3. Build the application:

```
make
```

*Note:* The compiled application is written to the `build` subdirectory. To have the application written to a different location, the `O=/path/to/build/directory` option may be specified in the `make` command.

## 4.0 Running the Application

The application requires a number of command line options:

```
./build/l2fwd-vf [EAL options] -- -p PORTMASK [-q NQ]
```

where,

- `-p PORTMASK`: A hexadecimal bitmask of the ports to configure
- `-q NQ`: A number of queues (=ports) per lcore (default is 1)

To run the application in linuxapp environment with 4 lcores, 8 ports and 2 RX queues per lcore, issue the command:

```
$ ./build/l2fwd-vf -c f -n 4 -- -q 2 -p ff
```

Refer to the *Intel® DPDK Getting Started Guide* for general information on running applications and the Environment Abstraction Layer (EAL) options.

## 5.0 Explanation

The following sections provide some explanation of the code.

## 5.1 Command Line Arguments

The L2 Forwarding sample application takes specific parameters, in addition to Environment Abstraction Layer (EAL) arguments (see Chapter 4.0).

Command line parsing is done in the same way as it is done in the basic L2 Forwarding Sample Application. See Section 5.1 in the *Intel® Data Plane Development Kit - L2 Forwarding Sample Application User Guide* for more information.

## 5.2 Mbuf Pool Initialization

Mbuf pool initialization is done in the same way as it is done in the basic L2 Forwarding Sample Application. See Section 5.2 in the *Intel® Data Plane Development Kit - L2 Forwarding Sample Application User Guide* for more information.

## 5.3 Driver Initialization

The main part of the code in the `main()` function relates to the initialization of the driver. To fully understand this code, it is recommended to study the chapters that related to the *Poll Mode Driver* in the *Intel® DPDK Programmer's Guide* and the *Intel® DPDK API Reference*.

```
/* init driver(s) */

#ifdef RTE_LIBRTE_IGB_PMD
    if (rte_igb_pmd_init() < 0)
        rte_exit(EXIT_FAILURE, "Cannot init igb pmd\n");
#endif

#ifdef RTE_LIBRTE_IXGBE_PMD
    if (rte_ixgbe_pmd_init() < 0)
        rte_exit(EXIT_FAILURE, "Cannot init ixgbe pmd\n");
#endif

    if (rte_eal_pci_probe() < 0)
        rte_exit(EXIT_FAILURE, "Cannot probe PCI\n");

    nb_ports = rte_eth_dev_count();

    if (nb_ports == 0)
        rte_exit(EXIT_FAILURE, "No Ethernet ports - bye\n");

    if (nb_ports > L2FWD_MAX_PORTS)
        nb_ports = L2FWD_MAX_PORTS;

    nb_lcores = rte_lcore_count();
```

Observe that:

- `rte_ixgbevf_pmd_init()` simultaneously registers the driver as a PCI driver and as an Ethernet* Poll Mode Virtual Function driver.
- `rte_eal_pci_probe()` parses the devices on the PCI bus and initializes recognized devices.

The next step is to configure the RX and TX queues. For each port, there is only one RX queue and one TX queue (only one lcore is able to poll a given port's RX queue). The `rte_eth_dev_configure()` function is used to configure the number of queues for a port:

```
ret = rte_eth_dev_configure((uint8_t) portid, 1, (uint16_t) n_tx_queue, &port_conf);
 if (ret < 0)
     rte_exit(EXIT_FAILURE, "Cannot configure device: "
              "err=%d, port=%u\n",
              ret, portid);
```

The global configuration is stored in a static structure:

```
static const struct rte_eth_conf port_conf = {
    .rxmode = {
        .split_hdr_size = 0,
        .header_split   = 0, /**< Header Split disabled */
        .hw_ip_checksum = 0, /**< IP checksum offload disabled */
        .hw_vlan_filter = 0, /**< VLAN filtering disabled */
        .jumbo_frame    = 0, /**< Jumbo Frame Support disabled */
        .hw_strip_crc   = 0, /**< CRC stripped by hardware */
    },
    .txmode = {
    },
};
```

## 5.4 RX Queue Initialization

The application uses one lcore to poll one or several ports, depending on the `-q` option, which specifies the number of queues per lcore.

For example, if the user specifies `-q 4`, the application is able to poll four ports with one lcore. If there are 16 ports on the target (and if the portmask argument is `-p ffff`), the application will need four lcores to poll all the ports.

*Note:* A single lcore can poll multiple RX queues, but multiple lcores cannot poll from a single RX queue.) The implied limitation of this is that we cannot have more lcores than the number of ports in our system because each port has only one RX queue implemented in a virtualized environment.

```
ret = rte_eth_rx_queue_setup((uint8_t) portid, 0, nb_rxd,
                             SOCKET0, &rx_conf,
                             l2fwd_pktmbuf_pool);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "rte_eth_tx_queue_setup: "
             "err=%d, port=%u\n",
             ret, portid);
```

The list of queues that must be polled for a given lcore is stored in a private structure called `struct lcore_queue_conf`.

```
struct lcore_queue_conf {
    unsigned n_rx_queue;
    unsigned rx_queue_list[MAX_RX_QUEUE_PER_LCORE];
    unsigned tx_queue_id[L2FWD_MAX_PORTS];
    struct mbuf_table tx_mbufs[L2FWD_MAX_PORTS];

    } __rte_cache_aligned;

struct lcore_queue_conf lcore_queue_conf[RTE_MAX_LCORE];
```

The values `n_rx_queue` and `rx_queue_list[]` are used in the main packet processing loop (see Section 5.6, “Receive, Process and Transmit Packets” on page 9).

The global configuration for the RX queues is stored in a static structure:

```
static const struct rte_eth_rxconf rx_conf = {
    .rx_thresh = {
        .pthresh = RX_PTHRESH,
        .hthresh = RX_HTHRESH,
        .wthresh = RX_WTHRESH,
    },
};
```

## 5.5 TX Queue Initialization

Each lcore should be able to transmit on any port where each port has only one TX queue implemented in virtualized environment.

```
        /* init one TX queue */
        ret = rte_eth_tx_queue_setup((uint8_t) portid,
                                     (uint16_t) queueid, nb_txd,
                                     SOCKET0, &tx_conf);
        if (ret < 0)
            rte_exit(EXIT_FAILURE, "rte_eth_tx_queue_setup: "
                     "err=%d, port=%u queue=%u\n",
                     ret, portid, queueid);
```

The global configuration for RX queues is stored in a static structure:

```
static const struct rte_eth_txconf tx_conf = {
    .tx_thresh = {
        .pthresh = TX_PTHRESH,
        .hthresh = TX_HTHRESH,
        .wthresh = TX_WTHRESH,
    },
    .tx_free_thresh = RTE_TEST_TX_DESC_DEFAULT + 1, /* disable feature */
};
```

## 5.6 Receive, Process and Transmit Packets

The receive, process and transmit packets operations are the same as those for the basic L2 Forwarding Sample Application. See Section 5.6 in the *Intel®* Data Plane Development Kit - *L2 Forwarding Sample Application User Guide* for more information.

§ §