Intel® Data Plane Development Kit - VMDQ and DCB L2 Forwarding Sample Application

User Guide

April 2012

Intel Confidential

Reference Number: 482255-1.1

Contents

Revision History

Date	Revision	Description
April 2012	1.1	Updates for software release 1.2
September 2011	1.0	Initial release

1.0 Introduction

The VMDQ and DCB Forwarding sample application is a simple example of packet processing using the Intel® Data Plane Development Kit (Intel® DPDK). The application performs L2 forwarding using VMDQ and DCB to divide the incoming traffic into 128 queues. The traffic splitting is performed in hardware by the VMDQ and DCB features of the Intel® 82599 10 Gigabit Ethernet Controller.

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

This sample application can be used as a starting point for developing a new application that is based on the Intel® DPK and uses VMDQ and DCB for traffic partitioning.

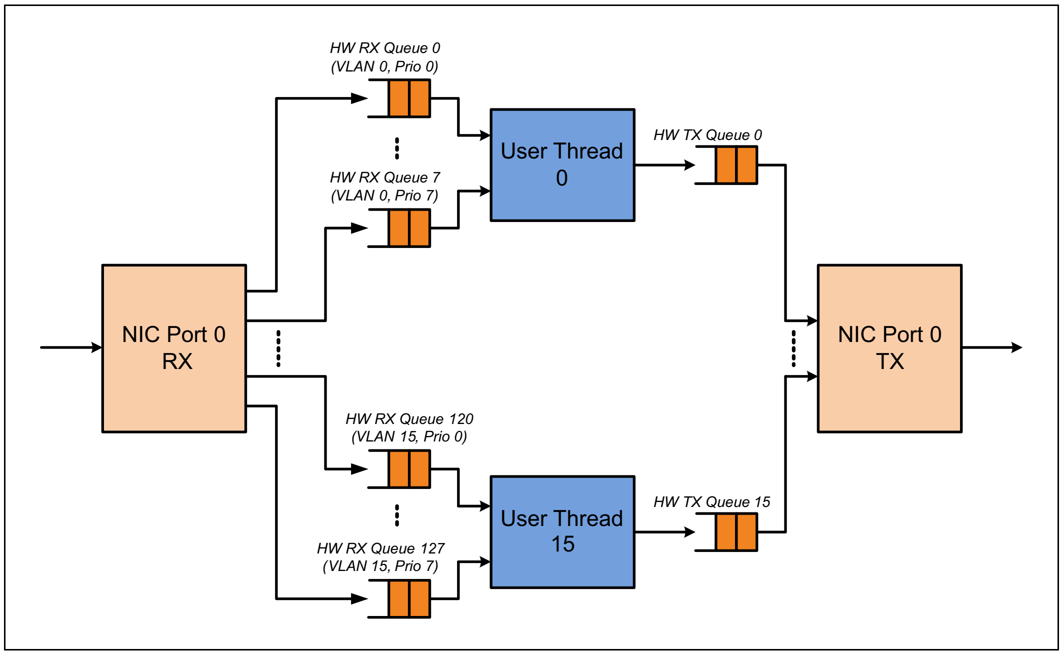
The VMDQ and DCB filters work on VLAN traffic to divide the traffic into 128 input queues on the basis of the VLAN ID field and VLAN user priority field. VMDQ filters split the traffic into 16 or 32 groups based on the VLAN ID. Then, DCB places each packet into one of either 4 or 8 queues within that group, based upon the VLAN user priority field.

In either case, 16 groups of 8 queues, or 32 groups of 4 queues, the traffic can be split into 128 hardware queues on the NIC, each of which can be polled individually by an Intel® DPDK application.

All traffic is read from a single incoming port (port 0) and output on port 1, without any processing being performed. The traffic is split into 128 queues on input, where each thread of the application reads from multiple queues. For example, when run with 8 threads, that is, with the `-c FF` option, each thread receives and forwards packets from 16 queues.

As supplied, the sample application configures the VMDQ feature to have 16 pools with 8 queues each as indicated in Figure 1. The Intel® 82599 10 Gigabit Ethernet Controller NIC also supports the splitting of traffic into 32 pools of 4 queues each and this can be used by changing the `NUM_POOLS` parameter in the supplied code.

Figure 1. Packet Flow Through the VMDQ and DCB Sample Application



In Linux* user space, the application can display statistics with the number of packets received on each queue. To have the application display the statistics, send a `SIGHUP` signal to the running application process, as follows:

```
kill -HUP <pid>
```

where, `<pid>` is the process id of the application process.

The VMDQ and DCB Forwarding sample application is in many ways simpler than the L2 Forwarding application (see the *Intel® DPDK L2 Forwarding Sample Application User Guide*) as it performs unidirectional L2 forwarding of packets from one port to a second port. No command-line options are taken by this application apart from the standard EAL command-line options.

Note: Since VMD queues are being used for VMM, this application works correctly when VTd is disabled in the BIOS or Linux* kernel (`intel_iommu=off`).

3.0 Compiling the Application

1. Go to the examples directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/vmdq_dcb
```

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
   ```

4.0 Running the Application

To run the example in a linuxapp environment:

```
user@target:~$ ./build/vmdq_dcb -c f -n 4
```

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 Initialization

The EAL, driver and PCI configuration is performed largely as in the L2 Forwarding sample application, as is the creation of the mbuf pool. See the *Intel® DPDK L2 Forwarding Sample Application User Guide*. Where this example application differs is in the configuration of the NIC port for RX.

The VMDQ and DCB hardware feature is configured at port initialization time by setting the appropriate values in the `rte_eth_conf` structure passed to the `rte_eth_dev_configure()` API. Initially in the application, a default structure is provided for VMDQ and DCB configuration to be filled in later by the application.

```
/* empty vmdq+dcb configuration structure. Filled in programatically */
static const struct rte_eth_conf vmdq_dcb_conf_default = {
    .rxmode = {
        .mq_mode        = ETH_VMDQ_DCB,
        .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 */
    },
    .txmode = {
        },
    .rx_adv_conf = {
        /*
         * should be overridden separately in code with
         * appropriate values
         */
    .vmdq_dcb_conf = {
        .nb_queue_pools = NUM_POOLS,
        .enable_default_pool = 0,
        .default_pool = 0,
        .nb_pool_maps = 0,
        .pool_map = {{0, 0},},
```

```
        .dcb_queue = {0},
        },
    },
};
```

The `get_eth_conf()` function fills in an `rte_eth_conf` structure with the appropriate values, based on the global `vlan_tags` array, and dividing up the possible user priority values equally among the individual queues (also referred to as *traffic classes*) within each pool, that is, if the number of pools is 32, then the user priority fields are allocated two to a queue. If 16 pools are used, then each of the 8 user priority fields is allocated to its own queue within the pool. For the VLAN IDs, each one can be allocated to possibly multiple pools of queues, so the `pools` parameter in the `rte_eth_vmdq_dcb_conf` structure is specified as a bitmask value.

```
const uint16_t vlan_tags[] = {
    0,  1,  2,  3,  4,  5,  6,  7,
    8,  9, 10, 11,  12, 13, 14, 15,
    16, 17, 18, 19, 20, 21, 22, 23,
    24, 25, 26, 27, 28, 29, 30, 31
};

    /* Builds up the correct configuration for vmdq+dcb based on the vlan tags array
     * given above, and the number of traffic classes available for use. */

static inline int
get_eth_conf(struct rte_eth_conf *eth_conf, enum rte_eth_nb_pools num_pools)
{
    struct rte_eth_vmdq_dcb_conf conf;
    unsigned i;

    if (num_pools != ETH_16_POOLS && num_pools != ETH_32_POOLS ) return -1;

        conf.nb_queue_pools = num_pools;
        conf.enable_default_pool = 0;
        conf.nb_pool_maps = sizeof( vlan_tags )/sizeof( vlan_tags[ 0 ]);
        for (i = 0; i < conf.nb_pool_maps; i++){
            conf.pool_map[i].vlan_id = vlan_tags[ i ];
            conf.pool_map[i].pools = 1 << (i % num_pools);
        }
        for (i = 0; i < ETH_DCB_NUM_USER_PRIORITIES; i++){
             conf.dcb_queue[i] = (uint8_t)(i % (NUM_QUEUES/num_pools));
        }
        rte_memcpy(eth_conf, &vmdq_dcb_conf_default, sizeof(*eth_conf));
        rte_memcpy(&eth_conf->rx_adv_conf.vmdq_dcb_conf, &conf,
                   sizeof(eth_conf->rx_adv_conf.vmdq_dcb_conf));
    return 0;
}
```

Once the network port has been initialized using the correct VMDQ and DCB values, the initialization of the port's RX and TX hardware rings is performed similarly to that in the L2 Forwarding sample application. See the *Intel® DPDK L2 Forwarding Sample Application User Guide* for more information.

5.2 Statistics Display

When run in a linuxapp environment, the VMDQ and DCB Forwarding sample application can display statistics showing the number of packets read from each RX queue. This is provided by way of a signal handler for the `SIGHUP` signal, which simply prints to standard output the packet counts in grid form. Each row of the output is a single pool with the columns being the queue number within that pool.

To generate the statistics output, use the following command:

```
user@host$ sudo killall -HUP vmdq_dcb_app
```

Please note that the statistics output will appear on the terminal where the `vmdq_dcb_app` is running, rather than the terminal from which the `HUP` signal was sent.

§ §