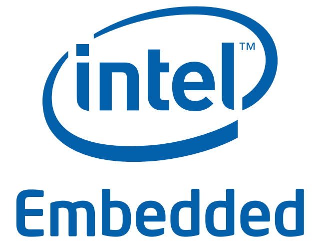


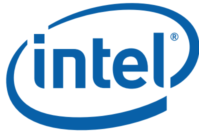
Intel® Data Plane Development Kit - L2 Forwarding Sample Application

User Guide

April 2012

Intel Confidential

Reference Number: 482250-1.1



INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

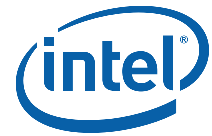
Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm.

Any software source code reprinted in this document is furnished for informational purposes only and may only be used or copied and no license, express or implied, by estoppel or otherwise, to any of the reprinted source code is granted by this document.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2012, Intel Corporation. All rights reserved.

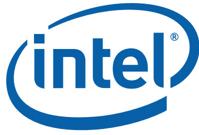


Contents

1.0 Introduction 4
1.1 Documentation Roadmap 4
2.0 Overview 4
3.0 Compiling the Application 5
4.0 Running the Application 5
5.0 Explanation 6
5.1 Command Line Arguments 6
5.2 Mbuf Pool Initialization 6
5.3 Driver Initialization 7
5.4 RX Queue Initialization 8
5.5 TX Queue Initialization 9
5.6 Receive, Process and Transmit Packets 9

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 L2 Forwarding sample application is a simple example of packet processing using the Intel® Data Plane Development Kit (Intel® DPDK).

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.

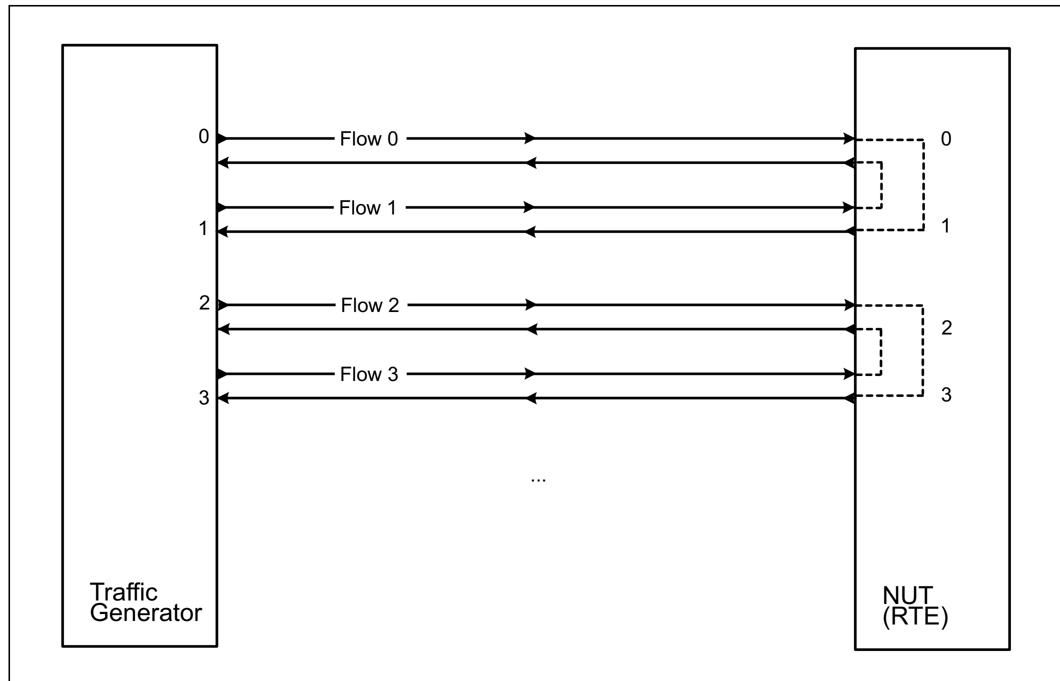
2.0 Overview

The L2 Forwarding 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.

3.0 Compiling the Application

1. Go to the example directory:

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

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

The application requires a number of command line options:

```
./build/l2fwd [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, 16 ports and 8 RX queues per lcore, issue the command:

```
$ ./build/l2fwd -c f -n 4 -- -q 8 -p ffff
```

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 Section 4.0). The preferred way to parse parameters is to use the `getopt()` function, since it is part of a well-defined and portable library.

The parsing of arguments is done in the `l2fwd_parse_args()` function. The method of argument parsing is not described here. Refer to the *glibc getopt (3)* man page for details.

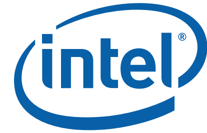
EAL arguments are parsed first, then application-specific arguments. This is done at the beginning of the `main()` function:

```
/* init EAL */
ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid EAL arguments\n");
argc -= ret;
argv += ret;

/* parse application arguments (after the EAL ones) */
ret = l2fwd_parse_args(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid L2FWD arguments\n");
```

5.2 Mbuf Pool Initialization

Once the arguments are parsed, the mbuf pool is created. The mbuf pool contains a set of mbuf objects that will be used by the driver and the application to store network packet data:

```
/* create the mbuf pool */
l2fwd_pktmbuf_pool =
    rte_mempool_create("mbuf_pool", NB_MBUF,
                       MBUF_SIZE, 32,
                       sizeof(struct rte_pktmbuf_pool_private),
                       l2fwd_pktmbuf_pool_init, NULL,
```



```
                            rte_pktmbuf_init, NULL,
                            SOCKET0, 0);
    if (l2fwd_pktmbuf_pool == NULL)
        rte_panic("Cannot init mbuf pool\n");
```

The `rte_mempool` is a generic structure used to handle pools of objects. In this case, it is necessary to create a pool that will be used by the driver, which expects to have some reserved space in the mempool structure, `sizeof(struct rte_pktmbuf_pool_private)` bytes. The number of allocated `pktmbufs` is `NB_MBUF`, with a size of `MBUF_SIZE` each. A per-lcore cache of 32 mbufs is kept. The memory is allocated in NUMA socket 0, but it is possible to extend this code to allocate one mbuf pool per socket.

Two callback pointers are also given to the `rte_mempool_create()` function:

- The first callback pointer is to `rte_pktmbuf_pool_init()` and is used to initialize the private data of the mempool, which is needed by the driver. This function is provided by the mbuf API, but can be copied and extended by the developer.
- The second callback pointer given to `rte_mempool_create()` is the mbuf initializer. The default is used, that is, `rte_pktmbuf_init()`, which is provided in the `rte_mbuf` library. If a more complex application wants to extend the `rte_pktmbuf` structure for its own needs, a new function derived from `rte_pktmbuf_init()` can be created.

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();

    /* initialize all ports */
    for (portid = 0; portid < nb_ports; portid++) {
        /* ... */
        /* rx and tx queue init, refer to the source code for details */
        /* ... */
    }
```

Observe that:

- `rte_igb_pmd_init()` simultaneously registers the driver as a PCI driver and as an Ethernet* Poll Mode 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 (only one lcore is able to poll a given port). The number of TX queues depends on the number of available lcores. 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 */
    },
    .txmode = {
        .hwb_enable = 0, /**< Head Write Back disabled by default */
    },
};
```

5.4 RX Queue Initialization

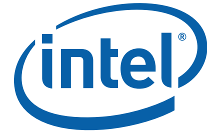
The application uses one lcore to poll one or several ports, depending on the `-q` option, which specifies the number of queues per lcore.

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.

```
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 Receive, Process and Transmit Packets).

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. That is why there are as many TX queues as lcores. A limitation of this approach is that it is not possible to have more lcores than the maximum available TX queues for a given port.

```
        /* init one TX queue per couple (lcore,port) */
        queueid = 0;
        for (lcore_id = 0; lcore_id < RTE_MAX_LCORE; lcore_id++) {
            if (rte_lcore_is_enabled(lcore_id) == 0)
                continue;
            printf("txq=%u,%u ", lcore_id, queueid);
            fflush(stdout);
            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);

           qconf = &lcore_queue_conf[lcore_id];
           qconf->tx_queue_id[portid] = queueid;
           queueid++;
        }
```

For each lcore, the identifier of the queue to use to transmit on a given port is stored in `struct lcore_queue_conf`. This value is used when transmitting a packet on a port (see Receive, Process and Transmit Packets).

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

In the `l2fwd_main_loop()` function, the main task is to read ingress packets from the RX queues. This is done using the following code:

```
/*
 * Read packet from RX queues
 */
for (i = 0; i < qconf->n_rx_queue; i++) {

    portid = qconf->rx_queue_list[i];
    nb_rx = rte_eth_rx_burst(portid, 0, pkts_burst,
                             MAX_PKT_BURST);

    for (j = 0; j < nb_rx; j++) {
        m = pkts_burst[j];
        l2fwd_simple_forward(m, portid);
    }
}
```

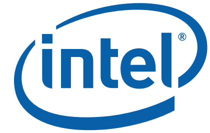
Packets are read in a burst of size `MAX_PKT_BURST`. The `rte_eth_rx_burst()` function writes the mbuf pointers in a local table and returns the number of available mbufs in the table.

Then, each mbuf in the table is processed by the `l2fwd_simple_forward()` function. The processing is very simple: process the TX port from the RX port, then replace the source and destination MAC addresses.

Note: In the following code, one line for getting the output port requires some explanation. During the initialization process, a static array of destination ports (`l2fwd_dst_ports[]`) is filled such that for each source port, a destination port is assigned that is either the next or previous enabled port from the portmask. Naturally, the number of ports in the portmask must be even, otherwise, the application exits.

```
static void
l2fwd_simple_forward(struct rte_mbuf *m, unsigned portid)
{
    struct ether_hdr *eth;
    void *tmp;
    unsigned dst_port;

    dst_port = l2fwd_dst_ports[portid];
    eth = rte_pktmbuf_mtod(m, struct ether_hdr *);

    /* 00:09:c0:00:00:xx */
    tmp = &eth->d_addr.addr_bytes[0];
    *((uint64_t *)tmp) = 0x000000c00900 + (dst_port << 24);

    /* src addr */
    ether_addr_copy(&l2fwd_ports_eth_addr[dst_port], &eth->s_addr);

    l2fwd_send_packet(m, (uint8_t) dst_port);
}
```

Then, the packet is sent using the `l2fwd_send_packet(m, dst_port)` function. For this test application, the processing is exactly the same for all packets arriving on the same RX port. Therefore, it would have been possible to call the `l2fwd_send_burst()` function directly from the main loop to send all the received packets on the same TX port, using the burst-oriented send function, which is more efficient.



However, in real-life applications (such as, L3 routing), packet N is not necessarily forwarded on the same port as packet N-1. The application is implemented to illustrate that, so the same approach can be reused in a more complex application.

The `l2fwd_send_packet()` function stores the packet in a per-lcore and per-txport table. If the table is full, the whole packets table is transmitted using the `l2fwd_send_burst()` function:

```
/* Send the packet on an output interface */
static int
l2fwd_send_packet(struct rte_mbuf *m, uint8_t port)
{
    unsigned lcore_id, len;
    struct lcore_queue_conf *qconf;

    lcore_id = rte_lcore_id();

    qconf = &lcore_queue_conf[lcore_id];
    len = qconf->tx_mbufs[port].len;
    qconf->tx_mbufs[port].m_table[len] = m;
    len++;

    /* enough pkts to be sent */
    if (unlikely(len == MAX_PKT_BURST)) {
        l2fwd_send_burst(qconf, MAX_PKT_BURST, port);
        len = 0;
    }

    qconf->tx_mbufs[port].len = len;
    return 0;
}
```

To ensure that no packets remain in the tables, each lcore does a draining of TX queue in its main loop. This technique introduces some latency when there are not many packets to send, however it improves performance:

```
        cur_tsc = rte_rdtsc();

        /*
         * TX burst queue drain
         */
        diff_tsc = cur_tsc - prev_tsc;
        if (unlikely(diff_tsc > BURST_TX_DRAIN)) {

            /* this could be optimized (use queueid instead of
             * portid), but it is not called so often */
            for (portid = 0; portid < L2FWD_MAX_PORTS; portid++) {
                if (qconf->tx_mbufs[portid].len == 0)
                    continue;
                l2fwd_send_burst(&lcore_queue_conf[lcore_id],
                                 qconf->tx_mbufs[portid].len,
                                 portid);
                qconf->tx_mbufs[portid].len = 0;
            }

            prev_tsc = cur_tsc;
        }
```

§ §