Intel® Data Plane Development Kit - Exception Path Sample Application

User Guide

April 2012

Intel Confidential

Reference Number: 482248-1.1

Contents

1.0 Introduction 4
1.1 Documentation Roadmap 4
2.0 Overview 5
3.0 Compiling the Application 5
4.0 Running the Application 6
4.1 Getting Statistics 6
5.0 Explanation 6
5.1 Initialization 6
5.2 Packet Forwarding 8
5.3 Managing TAP Interfaces and Bridges 9

Revision History

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

1.0 Introduction

The Exception Path sample application is a simple example that demonstrates the use of the Intel® DPDK to set up an *exception path* for packets to go through the Linux* kernel. This is done by using virtual TAP network interfaces. These can be read from and written to by the Intel® DPDK application and appear to the kernel as a standard network interface.

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

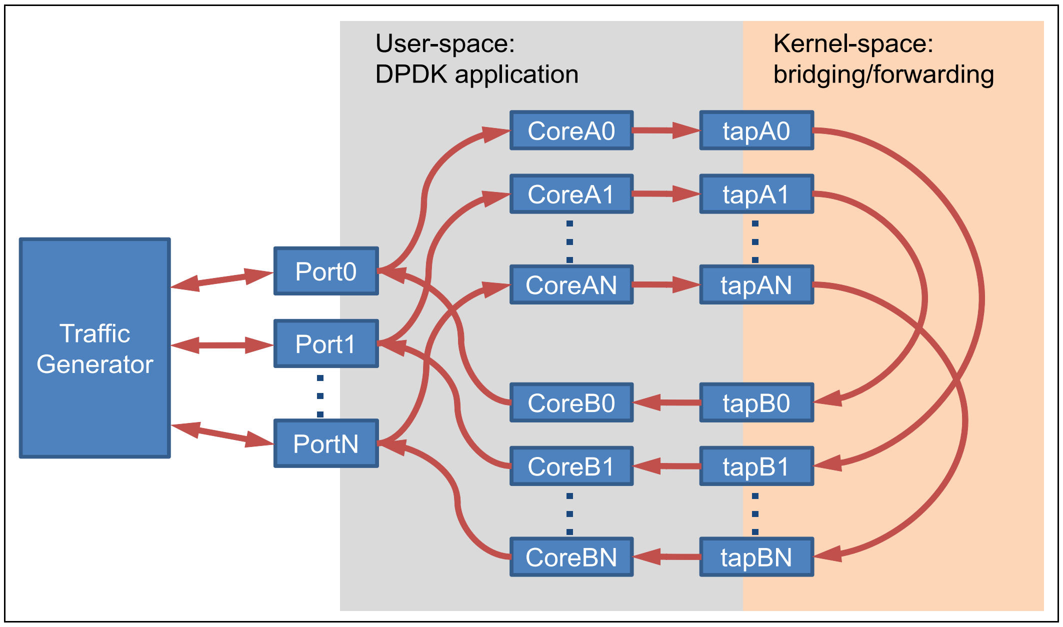
  A glossary of terms is also provided.
- **API Reference**: Provides detailed information about Intel® DPDK functions, data structures and other programming constructs.
- **Sample Application User Guides**: A set of guides, each describing a sample application that showcases specific functionality, together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

2.0 Overview

The application creates two threads for each NIC port being used. One thread reads from the port and writes the data unmodified to a thread-specific TAP interface. The second thread reads from a TAP interface and writes the data unmodified to the NIC port.

The packet flow through the exception path application is as shown in the following figure.

Figure 1. Packet Flow



To make throughput measurements, kernel bridges must be setup to forward data between the bridges appropriately.

3.0 Compiling the Application

1. Go to example directory:

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

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

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

Note: This application is intended as a linuxapp only.

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/exception_path [EAL options] -- -p PORTMASK -i IN_CORES -o OUT_CORES
```

where:

- `-p PORTMASK`: A hex bitmask of ports to use
- `-i IN_CORES`: A hex bitmask of cores which read from NIC
- `-o OUT_CORES`: A hex bitmask of cores which write to NIC

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

The number of bits set in each bitmask must be the same, and half of the number of bits set in the coremask `-c` parameter of the EAL options. The same bit must not be set in `IN_CORES` and `OUT_CORES`. The affinities between ports and cores are set beginning with the least significant bit of each mask, that is, the port represented by the lowest bit in `PORTMASK` is read from by the core represented by the lowest bit in `IN_CORES`, and written to by the core represented by the lowest bit in `OUT_CORES`.

For example to run the application with two ports and four cores:

```
./build/exception_path -c f -n 4 -- -p 3 -i 3 -o c
```

4.1 Getting Statistics

While the application is running, statistics on packets sent and received can be displayed by sending the SIGUSR1 signal to the application from another terminal:

```
killall -USR1 exception_path
```

The statistics can be reset by sending a `SIGUSR2` signal in a similar way.

5.0 Explanation

The following sections provide some explanation of the code.

5.1 Initialization

Setup of the mbuf pool, driver and queues is similar to the setup done in the L2 Forwarding sample application (see the *Intel® DPDK L2 Forwarding Sample Application User Guide* for details). In addition, the TAP interfaces must also be created. A TAP interface is created for each lcore that is being used. The code for creating the TAP interface is as follows:

```
/*
 * Create a tap network interface, or use existing one with same name.
 * If name[0]='\0' then a name is automatically assigned and returned in name.
 */

static int tap_create(char *name)
{
    struct ifreq ifr;
    int fd, ret;

    fd = open("/dev/net/tun", O_RDWR);
    if (fd < 0)
        return fd;

    memset(&ifr, 0, sizeof(ifr));

    /* TAP device without packet information */
    ifr.ifr_flags = IFF_TAP | IFF_NO_PI;
    if (*name)
        strncpy(ifr.ifr_name, name, IFNAMSIZ);

    ret = ioctl(fd, TUNSETIFF, (void *) &ifr);
    if (ret < 0) {
        close(fd);
        return ret;
    }

    strcpy(name, ifr.ifr_name);
    return fd;
}
```

The other step in the initialization process that is unique to this sample application is the association of each port with two cores:

- One core to read from the port and write to a TAP interface
- A second core to read from a TAP interface and write to the port

This is done using an array called `port_ids[]`, which is indexed by the lcore IDs. The population of this array is shown below:

```
tx_port = 0;
rx_port = 0;
RTE_LCORE_FOREACH(i) {
    if (input_cores_mask & (1 << i)) {
        /* Skip ports that are not enabled */
        while ((ports_mask & (1 << rx_port)) == 0) {
            rx_port++;
            if (rx_port > (sizeof(ports_mask) * 8))
                goto fail; /* not enough ports */
        }
        port_ids[i] = rx_port++;
    } else if (output_cores_mask & (1 << i)) {
        /* Skip ports that are not enabled */
        while ((ports_mask & (1 << tx_port)) == 0) {
            tx_port++;
            if (tx_port > (sizeof(ports_mask) * 8))
                goto fail; /* not enough ports */
        }
        port_ids[i] = tx_port++;
    }
}
```

5.2 Packet Forwarding

After the initialization steps are complete, the `main_loop()` function is run on each lcore. This function first checks the `lcore_id` against the user provided `input_cores_mask` and `output_cores_mask` to see if this core is reading from or writing to a TAP interface.

For the case that reads from a NIC port, the packet reception is the same as in the L2 Forwarding sample application (see the "Receive, Process and Transmit Packets" section in the *Intel® DPDK L2 Forwarding Sample Application User Guide*). The packet transmission is done by calling `write()` with the file descriptor of the appropriate TAP interface and then explicitly freeing the mbuf back to the pool.

```
/* Loop forever reading from NIC and writing to tap */
for (;;) {
    struct rte_mbuf *pkts_burst[PKT_BURST_SZ];
    unsigned i;
    const unsigned nb_rx = rte_eth_rx_burst(port_ids[lcore_id], 0,
                                        pkts_burst, PKT_BURST_SZ);
    lcore_stats[lcore_id].rx += nb_rx;
    for (i = 0; likely(i < nb_rx); i++) {
        struct rte_mbuf *m = pkts_burst[i];
        int ret = write(tap_fd, rte_pktmbuf_mtod(m, void*),
                        rte_pktmbuf_data_len(m));
        rte_pktmbuf_free(m);
    }
}
```

For the other case that reads from a TAP interface and writes to a NIC port, packets are retrieved by doing a `read()` from the file descriptor of the appropriate TAP interface. This fills in the data into the mbuf, then other fields are set manually. The packet can then be transmitted as normal.

```
/* Loop forever reading from tap and writing to NIC */
for (;;) {
    int ret;
    struct rte_mbuf *m = rte_pktmbuf_alloc(pktmbuf_pool);

    ret = read(tap_fd, m->pkt.data, MAX_PACKET_SZ);
    lcore_stats[lcore_id].rx++;
    if (unlikely(ret < 0)) {
        FATAL_ERROR("Reading from %s interface failed",
                    tap_name);
    }
    m->pkt.nb_segs = 1;
    m->pkt.next = NULL;
    m->pkt.pkt_len = (uint16_t)ret;
    m->pkt.data_len = (uint16_t)ret;
    ret = rte_eth_tx_burst(port_ids[lcore_id], 0, &m, 1);
    if (unlikely(ret < 1))
        rte_pktmbuf_free(m);
}
```

To set up loops for measuring throughput, TAP interfaces can be connected using bridging. The steps to do this are described in the Managing TAP Interfaces and Bridges section that follows.

5.3 Managing TAP Interfaces and Bridges

The Exception Path sample application creates TAP interfaces with names of the format `tap_dpdk_nn`, where `nn` is the lcore ID. These TAP interfaces need to be configured for use:

```
ifconfig tap_dpdk_00 up
```

To set up a bridge between two interfaces so that packets sent to one interface can be read from another, use the `brctl` tool:

```
brctl addbr "br0"
brctl addif br0 tap_dpdk_00
brctl addif br0 tap_dpdk_03
ifconfig br0 up
```

The TAP interfaces created by this application exist only when the application is running, so the steps above need to be repeated each time the application is run. To avoid this, persistent TAP interfaces can be created using `openvpn`:

```
openvpn --mktun --dev tap_dpdk_00
```

If this method is used, then the steps above have to be done only once and the same TAP interfaces can be reused each time the application is run. To remove bridges and persistent TAP interfaces, the following commands are used:

```
ifconfig br0 down
brctl delbr br0
openvpn --rmtun --dev tap_dpdk_00
```

§ §