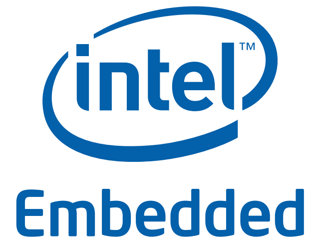


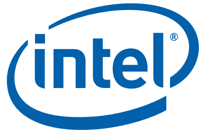
Intel® Data Plane Development Kit - IPv4 Multicast Sample Application

User Guide

April 2012

Intel Confidential

Reference Number: 496632-1.0



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.

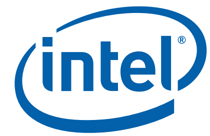
Any software source code reprinted in this document is furnished for informational purposes only and may only be used or copied and no license, express or implied, by estoppel or otherwise, to any of the reprinted source code is granted by this document.

Code Names are only for use by Intel to identify products, platforms, programs, services, etc. ("products") in development by Intel that have not been made commercially available to the public, i.e., announced, launched or shipped. They are never to be used as "commercial" names for products. Also, they are not intended to function as trademarks.

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

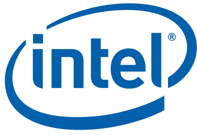
3.0 Building the Application 5

4.0 Running the Application 5

5.0 Explanation 6
- 5.1 Memory Pool Initialization 6
- 5.2 Hash Initialization 6
- 5.3 Forwarding 7
- 5.4 Buffer Cloning 8

Revision History

Date	Revision	Description
April 2012	1.0	Initial version of document



1.0 Introduction

The IPv4 Multicast application is a simple example of packet processing using the Intel® Data Plane Development Kit (Intel® DPDK). The application performs L3 multicasting.

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.

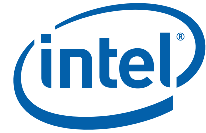
2.0 Overview

The application demonstrates the use of zero-copy buffers for packet forwarding. The initialization and run-time paths are very similar to those of the L2 forwarding application (see the *Intel® DPDK L2 Forwarding Sample Application User Guide* for more information). This guide highlights the differences between the two applications. There are two key differences from the L2 Forwarding sample application:

- The IPv4 Multicast sample application makes use of indirect buffers;
- The forwarding decision is taken based on information read from the input packet's IPv4 header.

The lookup method is the Four-byte Key (FBK) hash-based method. The lookup table is composed of pairs of destination IPv4 address (the FBK) and a port mask associated with that IPv4 address.

For convenience and simplicity, this sample application does not take IANA-assigned multicast addresses into account, but instead equates the last four bytes of the multicast group (that is, the last four bytes of the destination IP address) with the mask of ports to multicast packets to. Also, the application does not consider the Ethernet addresses; it looks only at the IPv4 destination address for any given packet.



3.0 Building the Application

To compile the application:

1. Go to the sample application directory:

```
export RTE_SDK=/path/to/rte_sdk
cd ${RTE_SDK}/examples/ipv4_multicast
```

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 has a number of command line options:

```
./build/ipv4_multicast [EAL options] -- -p PORTMASK [-q NQ]
```

where,

- `-p PORTMASK`: Hexadecimal bitmask of ports to configure
- `-q NQ`: determines the number of queues per lcore

Note: Unlike the basic L2/L3 Forwarding sample applications, NUMA support is not provided in the IPv4 Multicast sample application.

Typically, to run the IPv4 Multicast sample application, issue the following command (as root):

```
./build/ipv4_multicast -c 0x00f -n 3 -- -p 0x3 -q 1
```

In this command:

- The `-c` option enables cores 0, 1, 2 and 3
- The `-n` option specifies 3 memory channels
- The `-p` option enables ports 0 and 1
- The `-q` option assigns 1 queue to each lcore

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. As mentioned in the overview section, the initialization and run-time paths are very similar to those of the L2 Forwarding sample application (see the *Intel® DPDK L2 Forwarding Sample Application User Guide* for more information). The following sections describe aspects that are specific to the IPv4 Multicast sample application.

5.1 Memory Pool Initialization

The IPv4 Multicast sample application uses three memory pools. Two of the pools are for indirect buffers used for packet duplication purposes. Memory pools for indirect buffers are initialized differently from the memory pool for direct buffers:

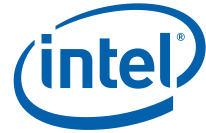
```
packet_pool = rte_mempool_create("packet_pool", NB_PKT_MBUF,
    PKT_MBUF_SIZE, 32, sizeof(struct rte_pktmbuf_pool_private),
    rte_pktmbuf_pool_init, NULL, rte_pktmbuf_init, NULL,
    SOCKET0, 0);

header_pool = rte_mempool_create("header_pool", NB_HDR_MBUF,
    HDR_MBUF_SIZE, 32, 0, NULL, NULL, rte_pktmbuf_init, NULL,
    SOCKET0, 0);

clone_pool = rte_mempool_create("clone_pool", NB_CLONE_MBUF,
    CLONE_MBUF_SIZE, 32, 0, NULL, NULL, rte_pktmbuf_init, NULL,
    SOCKET0, 0);
```

The reason for this is because indirect buffers are not supposed to hold any packet data and therefore can be initialized with lower amount of reserved memory for each buffer.

5.2 Hash Initialization

The hash object is created and loaded with the pre-configured entries read from a global array:

```
static int
init_mcast_hash(void)
{
    uint32_t i;

    mcast_hash = rte_fbk_hash_create(&mcast_hash_params);
    if (mcast_hash == NULL){
        return -1;
    }

    for (i = 0; i < N_MCAST_GROUPS; i ++){
        if (rte_fbk_hash_add_key(mcast_hash,
            mcast_group_table[i].ip,
            mcast_group_table[i].port_mask) < 0) {
            return -1;
        }
    }

    return 0;
}
```



5.3 Forwarding

All forwarding is done inside the `mcast_forward()` function. Firstly, the Ethernet* header is removed from the packet and the IPv4 address is extracted from the IPv4 header:

```
/* Remove the Ethernet header from the input packet */
    iphdr = (struct ipv4_hdr *)rte_pktmbuf_adj(m, sizeof(struct ether_hdr));
    RTE_MBUF_ASSERT(iphdr != NULL);

    dest_addr = rte_be_to_cpu_32(iphdr->dst_addr);
```

Then, the packet is checked to see if it is in fact it has a multicast destination address and if the routing table has any ports assigned to the destination address:

```
if(!IS_IPV4_MCAST(dest_addr) ||
        (hash = rte_fbk_hash_lookup(mcast_hash, dest_addr)) <= 0 ||
        (port_mask = hash & enabled_port_mask) == 0) {
        rte_pktmbuf_free(m);
        return;
    }
```

Then, the number of ports in the destination `portmask` is calculated with the help of the `bitcnt()` function:

```
/* Get number of bits set. */
static inline uint32_t bitcnt(uint32_t v)
{
    uint32_t n;

    for (n = 0; v != 0; v &= v - 1, n++)
        ;

    return (n);
}
```

This is done to determine which forwarding algorithm to use. This is explained in more detail in the next section.

Thereafter, a destination Ethernet address is constructed:

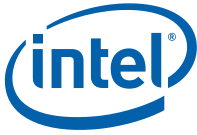
```
    /* construct destination ethernet address */
    dst_eth_addr = ETHER_ADDR_FOR_IPV4_MCAST(dest_addr);
```

Since Ethernet addresses are also part of the multicast process, each outgoing packet carries the same destination Ethernet address. The destination Ethernet address is constructed from the lower 23 bits of the multicast group ORed with the Ethernet address 01:00:5e:00:00:00, as per RFC 1112:

```
# define ETHER_ADDR_FOR_IPV4_MCAST(x) \
    (rte_cpu_to_be_64(0x01005e000000 | ((x) & 0x7fffff)) >> 16)
```

Then, packets are dispatched to the destination ports according to the `portmask` associated with a multicast group:

```
    for (port = 0; use_clone != port_mask; port_mask >>= 1, port++) {

        /* Prepare output packet and send it out. */
        if ((port_mask & 1) != 0) {
            if (likely ((mc = mcast_out_pkt(m, use_clone)) != NULL))
```




```
                mcast_send_pkt(mc,
                    (struct ether_addr *)&dst_eth_addr,
                    qconf, port);
            else if (use_clone == 0)
                rte_pktmbuf_free(m);
        }
    }
```

The actual packet transmission is done in the `mcask_send_pkt()` function:

```
static inline void mcast_send_pkt(struct rte_mbuf *pkt,
    struct ether_addr *dest_addr, struct lcore_queue_conf *qconf, uint8_t port)
{
    struct ether_hdr *ethdr;
    uint16_t len;

    /* Construct Ethernet header. */
    ethdr = (struct ether_hdr *)rte_pktmbuf_prepend(pkt, sizeof(*ethdr));
    RTE_MBUF_ASSERT(ethdr != NULL);

    ether_addr_copy(dest_addr, &ethdr->d_addr);
    ether_addr_copy(&ports_eth_addr[port], &ethdr->s_addr);
    ethdr->ether_type = rte_be_to_cpu_16(ETHER_TYPE_IPv4);

    /* Put new packet into the output queue */
    len = qconf->tx_mbufs[port].len;
    qconf->tx_mbufs[port].m_table[len] = pkt;
    qconf->tx_mbufs[port].len = ++len;

    /* Transmit packets */
    if (unlikely(MAX_PKT_BURST == len))
        send_burst(qconf, port);
}
```

5.4 Buffer Cloning

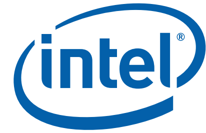
This is the most important part of the application since it demonstrates the use of zero-copy buffer cloning. There are two approaches for creating the outgoing packet and although both are based on the data zero-copy idea, there are some differences in the detail.

The first approach creates a clone of the input packet, for example, walk though all segments of the input packet and for each of segment, create a new buffer and attach that new buffer to the segment (refer to `rte_pktmbuf_clone()` in the *rte_mbuf* library for more details). A new buffer is then allocated for the packet header and is prepended to the cloned buffer.

The secondly approach does not make a clone, it just increments the reference counter for all input packet segment, allocates a new buffer for the packet header and prepends it to the input packet.

Basically, the first approach reuses only the input packet's data, but creates its own copy of packet's metadata. The second approach reuses both input packet's data and metadata.

The advantage of first approach is that each outgoing packet has its own copy of the metadata, so we can safely modify the data pointer of the input packet. That allows us to skip creation if the output packet is for the last destination port and instead modify input packet's header in place. For example, for N destination ports, we need to invoke `mcast_out_pkt()` (N-1) times.



The advantage of the second approach is that there is less work to be done for each outgoing packet, that is, the “clone” operation is skipped completely. However, there is a price to pay. The input packet’s metadata must remain intact, so for N destination ports, we need to invoke `mcast_out_pkt()` (N) times.

Therefore, for a small number of outgoing ports (and segments in the input packet), first approach is faster. As the number of outgoing ports (and/or input segments) grows, the second approach becomes more preferable.

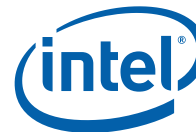
Depending on the number of segments or the number of ports in the outgoing portmask, either the first (with cloning) or the second (without cloning) approach is taken:

```
    use_clone = (port_num <= MCAST_CLONE_PORTS &&
        m->pkt.nb_segs <= MCAST_CLONE_SEGS);
```

It is the `mcast_out_pkt()` function that performs the packet duplication (either with or without actually cloning the buffers):

```
static inline struct rte_mbuf *mcast_out_pkt(struct rte_mbuf *pkt,
    int use_clone)
{
    struct rte_mbuf *hdr;

    /* Create new mbuf for the header. */
    if (unlikely ((hdr = rte_pktmbuf_alloc(header_pool)) == NULL))
        return (NULL);

    /* If requested, then make a new clone packet. */
    if (use_clone != 0 &&
        unlikely ((pkt = rte_pktmbuf_clone(pkt, clone_pool)) == NULL)) {
        rte_pktmbuf_free(hdr);
        return (NULL);
    }

    /* prepend new header */
    hdr->pkt.next = pkt;

    /* copy metadata from source packet*/
    rte_memcpy(&hdr->pkt.ol_flags, &pkt->pkt.ol_flags,
        sizeof (*hdr) - offsetof(struct rte_mbuf, pkt.ol_flags));

    /* update header's fields */
    hdr->pkt.pkt_len = (uint16_t)(hdr->pkt.data_len + pkt->pkt.pkt_len);
    hdr->pkt.nb_segs = (uint8_t)(pkt->pkt.nb_segs + 1);

    __rte_mbuf_sanity_check(hdr, RTE_MBUF_PKT, 1);
    return (hdr);
}
```

§ §



1.0 Introduction 4
1.1 Documentation Roadmap 4
2.0 Overview 4
3.0 Building the Application 5
4.0 Running the Application 5
5.0 Explanation 6
5.1 Memory Pool Initialization 6
5.2 Hash Initialization 6
5.3 Forwarding 7
5.4 Buffer Cloning 8