DPDK

Overview

What does DPDK want?

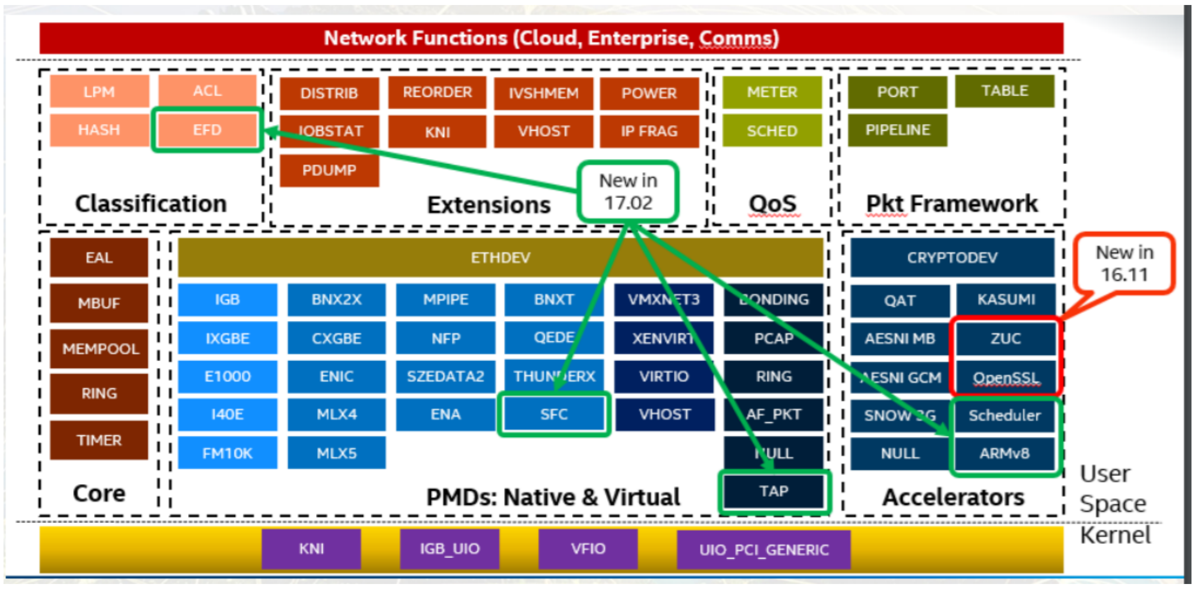
High performance network: low latency & high throughput

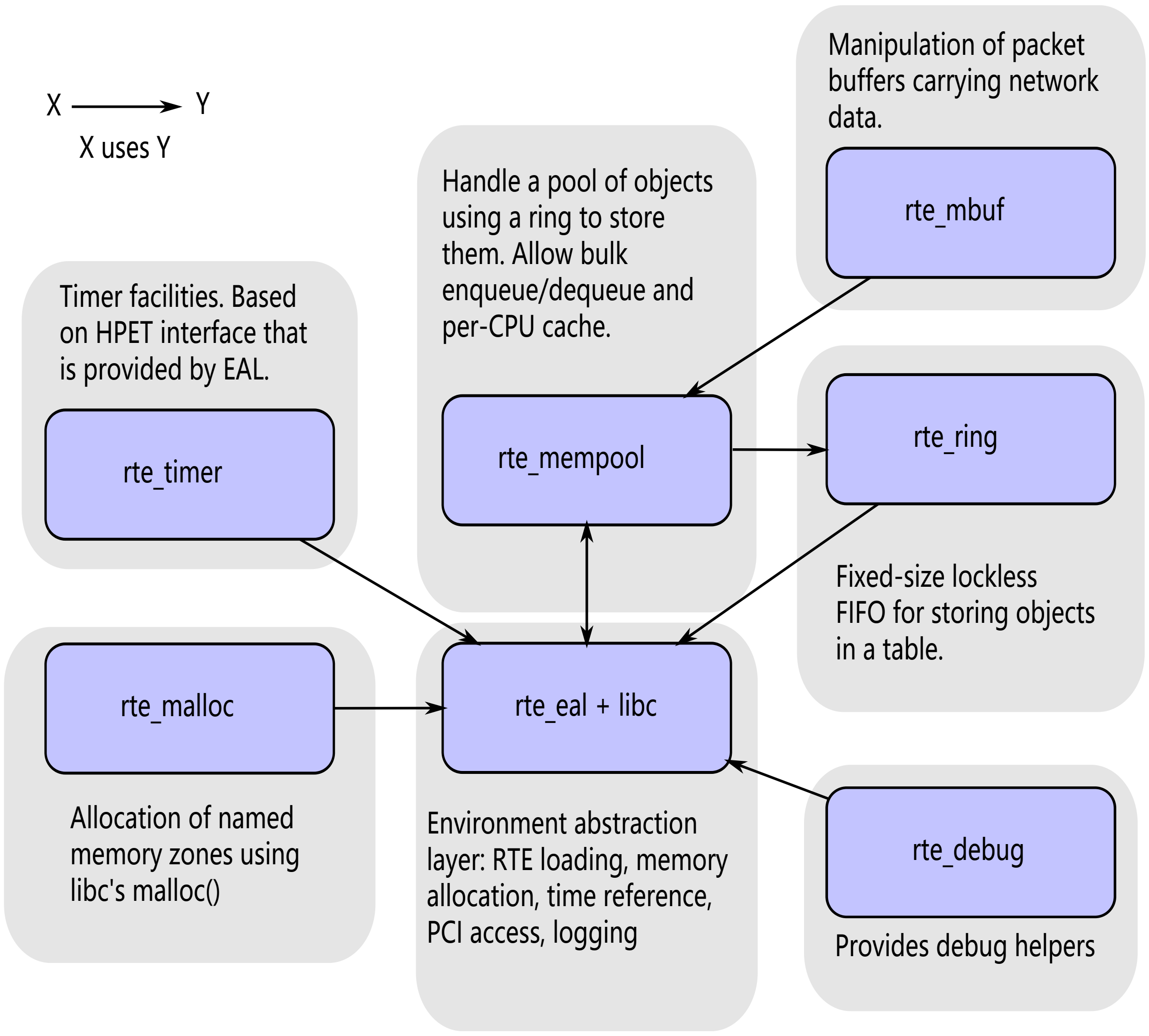
How?

kernel-bypass, zero-copy, optimizations(NUMA/cache/AVX/etc..)

Architecture

> Figure from **https://www.dpdk.org/wp-content/uploads/sites/35/2017/04/DPDK-India2017-RamiaJain-ArchitectureRoadmap.pdf**



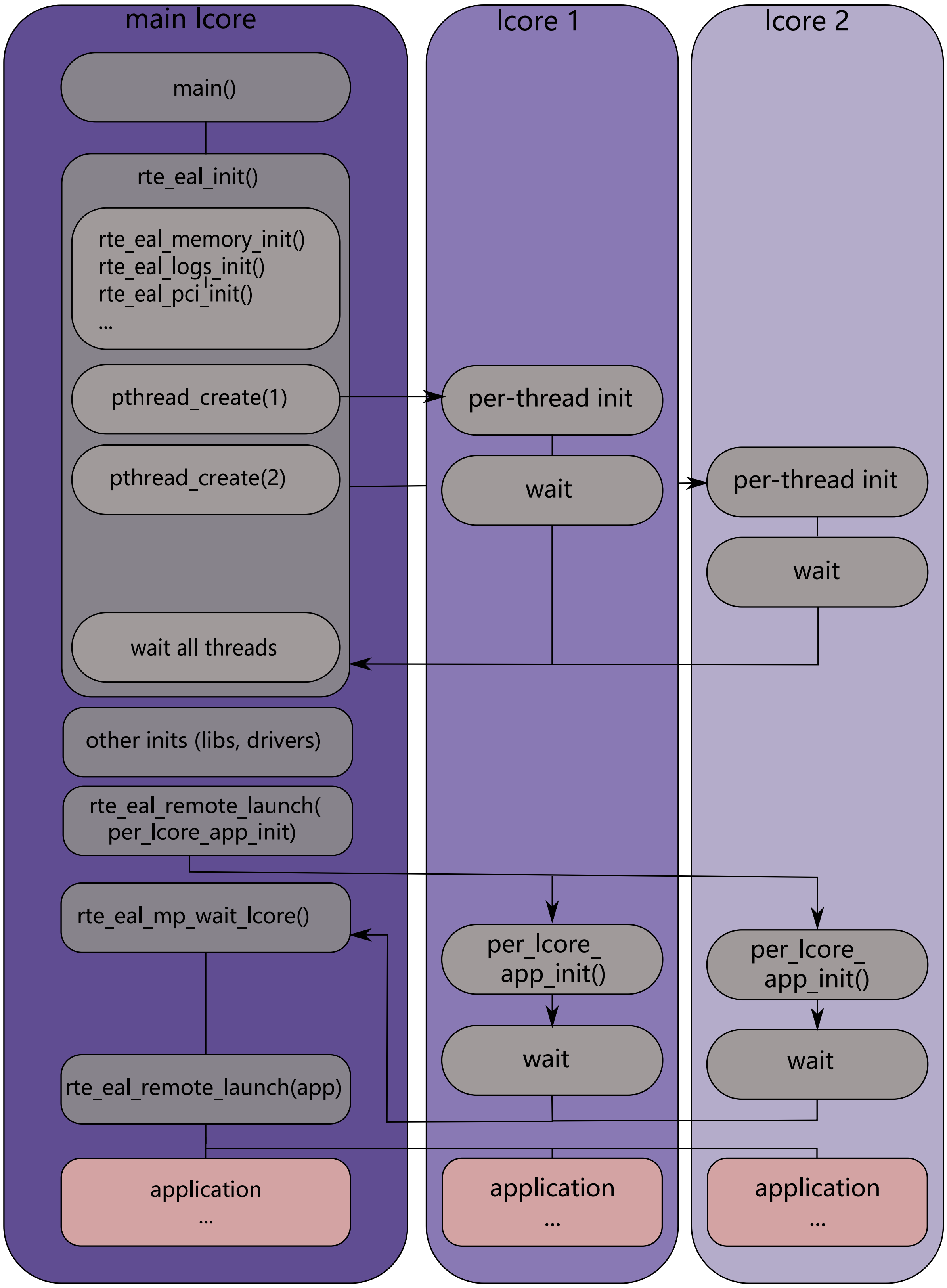


X → Y
X uses Y
Manipulation of packet buffers carrying network data.
rte_mbuf
Handle a pool of objects using a ring to store them. Allow bulk enqueue/dequeue and per-CPU cache.
rte_mempool
Timer facilities. Based on HPET interface that is provided by EAL.
rte_timer
rte_ring
Fixed-size lockless FIFO for storing objects in a table.
rte_malloc
Allocation of named memory zones using libc's malloc()
rte_eal + libc
Environment abstraction layer: RTE loading, memory allocation, time reference, PCI access, logging
rte_debug
Provides debug helpers

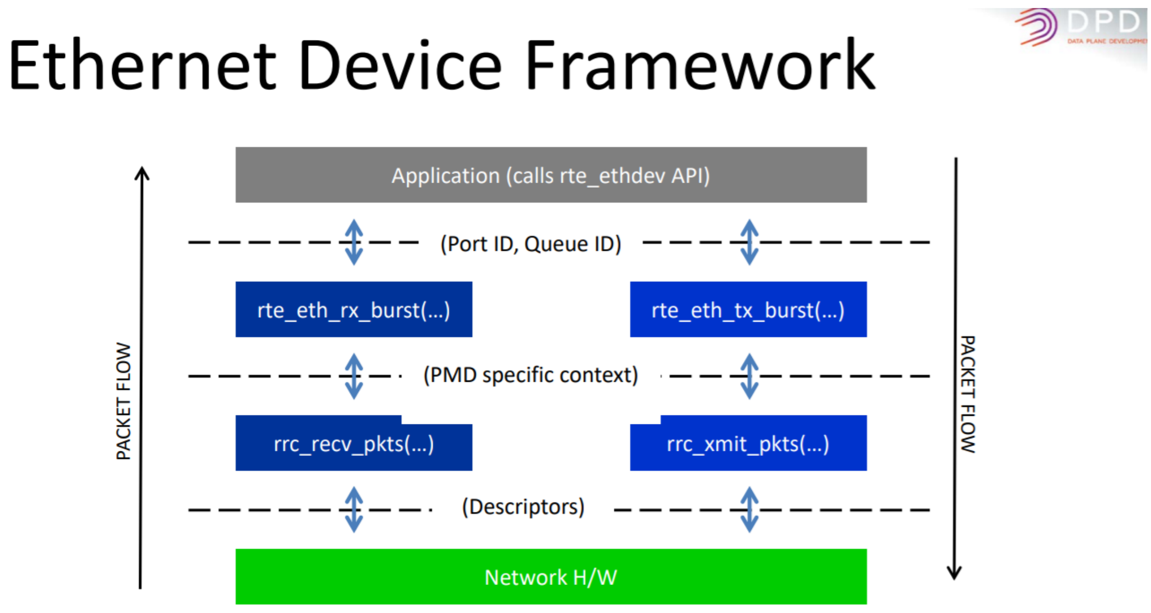
Several Concepts

EAL = Environment Abstraction Layer

need to be set up by call `rte_eal_init(,)` before using any DPDK components.



ETHDEV Framework

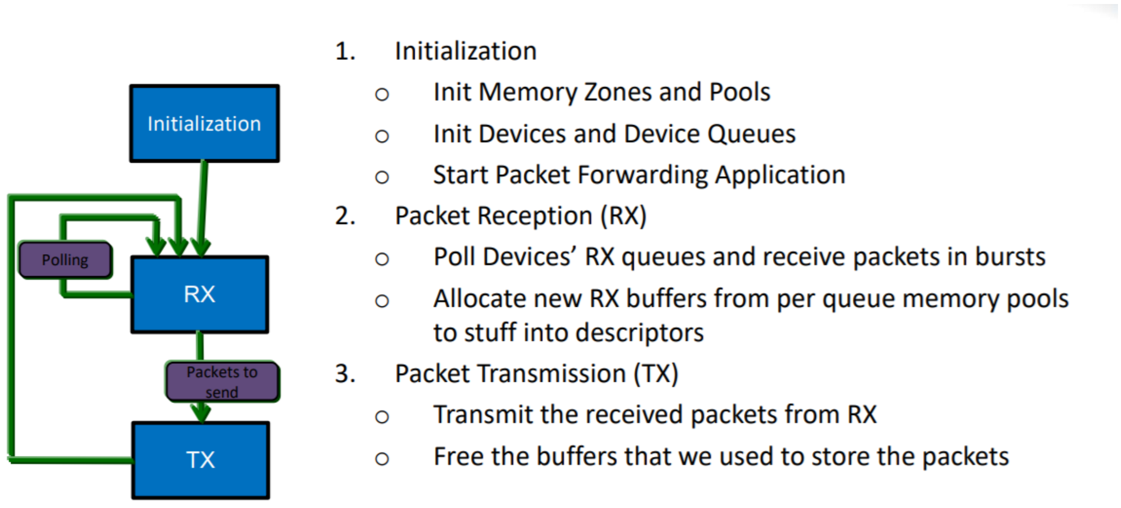


Ringbuffer: descriptor <-> mbuf

When packet arrives at NIC, NIC takes a descriptor out the ringbuffer, and do DMA.

When call **rte_eth_rx_burst**, DPDK takes up to MAX_BURST descriptors from the ringbuffer, fill necessary metainfo, and allocate new descriptors from the mempool to replace the taken descriptors.

Typical packet flow l2fwd



1. Initialization
   - Init Memory Zones and Pools
   - Init Devices and Device Queues
   - Start Packet Forwarding Application
2. Packet Reception (RX)
   - Poll Devices’ RX queues and receive packets in bursts
   - Allocate new RX buffers from per queue memory pools to stuff into descriptors
3. Packet Transmission (TX)
   - Transmit the received packets from RX
   - Free the buffers that we used to store the packets

Setting up environment

compiling & installing DPDK

> Full guide for Linux:
>
> http://doc.dpdk.org/guides/linux_gsg/index.html

```
tar xf dpdk.tar.gz
cd dpdk
meson build

## to include examples, replace meson build with command below
# meson -Dexamples=all build

ninja -C build
ninja -C build install
```

setting up Linux drivers

> **http://doc.dpdk.org/guides/linux_gsg/linux_drivers.html**
>
> available options are: vfio, vfio without MMU, uio

take **UIO** for example

```
sudo modprobe uio_pci_generic
```

or build `igb_uio` from source

```
git clone http://dpdk.org/git/dpdk-kmods
cd dpdk-kmods/linux/igb_uio
make
```

and load the module

```
sudo modprobe uio
sudo insmod igb_uio.ko
```

setting up huge pages

```
mkdir -p /dev/hugepages
mountpoint -q /dev/hugepages || mount -t hugetlbfs nodev /dev/hugepages
# 64 -> nr_hugepages, node0 -> NUMA node
echo 64 > /sys/devices/system/node/node0/hugepages/hugepages-2048kB/nr_hugepages
```

identify and bind NIC(ports) to kernel modules

use `dpdk-devbind.py`

```
./usertools/dpdk-devbind.py --status

Network devices using DPDK-compatible driver
============================================
0000:82:00.0 '82599EB 10-GbE NIC' drv=uio_pci_generic unused=ixgbe
0000:82:00.1 '82599EB 10-GbE NIC' drv=uio_pci_generic unused=ixgbe

Network devices using kernel driver
===================================
0000:04:00.0 'I350 1-GbE NIC' if=em0  drv=igb unused=uio_pci_generic *Active*
0000:04:00.1 'I350 1-GbE NIC' if=eth1 drv=igb unused=uio_pci_generic
0000:04:00.2 'I350 1-GbE NIC' if=eth2 drv=igb unused=uio_pci_generic
0000:04:00.3 'I350 1-GbE NIC' if=eth3 drv=igb unused=uio_pci_generic
```

```
Other network devices
=====================
<none>
```

to bind to `uio` driver

```
./usertools/dpdk-devbind.py --bind=uio_pci_generic 04:00.1
# or
./usertools/dpdk-devbind.py --bind=uio_pci_generic eth1
```

to unbind

```
./usertools/dpdk-devbind.py --bind=igb 04:00.1
```

Simple Application

> with code examples/l2fwd-cat/l2fwd-cat.c

Initializing EAL

```
    /* Initialize the Environment Abstraction Layer (EAL). */
    int ret = rte_eal_init(argc, argv);
    if (ret < 0)
        rte_exit(EXIT_FAILURE, "Error with EAL initialization\n");
```

Create mbuf pool

```
    /* Creates a new mempool in memory to hold the mbufs. */
    mbuf_pool = rte_pktmbuf_pool_create("MBUF_POOL", NUM_MBUFS * nb_ports,
        MBUF_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE, rte_socket_id());
```

Initial port (RX & TX)

```
/*
 * Initializes a given port using global settings and with the RX buffers
 * coming from the mbuf_pool passed as a parameter.
 */
static inline int
port_init(uint16_t port, struct rte_mempool *mbuf_pool)
{
    struct rte_eth_conf port_conf = port_conf_default;
    const uint16_t rx_rings = 1, tx_rings = 1;
    int retval;
    uint16_t q;
    uint16_t nb_rxd = RX_RING_SIZE;
    uint16_t nb_txd = TX_RING_SIZE;

    if (!rte_eth_dev_is_valid_port(port))
        return -1;

    /* Configure the Ethernet device. */
    retval = rte_eth_dev_configure(port, rx_rings, tx_rings, &port_conf);
    if (retval != 0)
```

```
        return retval;

    retval = rte_eth_dev_adjust_nb_rx_tx_desc(port, &nb_rxd, &nb_txd);
    if (retval != 0)
        return retval;

    /* Allocate and set up 1 RX queue per Ethernet port. */
    for (q = 0; q < rx_rings; q++) {
        retval = rte_eth_rx_queue_setup(port, q, nb_rxd,
                rte_eth_dev_socket_id(port), NULL, mbuf_pool); // NULL for rx_conf
        if (retval < 0)
            return retval;
    }

    /* Allocate and set up 1 TX queue per Ethernet port. */
    for (q = 0; q < tx_rings; q++) {
        retval = rte_eth_tx_queue_setup(port, q, nb_txd,
                rte_eth_dev_socket_id(port), NULL);
        if (retval < 0)
            return retval;
    }

    /* Start the Ethernet port. */
    retval = rte_eth_dev_start(port);
    if (retval < 0)
        return retval;

    /* Display the port MAC address. */
    struct rte_ether_addr addr;
    retval = rte_eth_macaddr_get(port, &addr);
    if (retval < 0)
        return retval;

    printf("Port %u MAC: %02" PRIx8 " %02" PRIx8 " %02" PRIx8
               " %02" PRIx8 " %02" PRIx8 " %02" PRIx8 "\n",
            port,
            addr.addr_bytes[0], addr.addr_bytes[1],
            addr.addr_bytes[2], addr.addr_bytes[3],
            addr.addr_bytes[4], addr.addr_bytes[5]);

    /* Enable RX in promiscuous mode for the Ethernet device. */
    retval = rte_eth_promiscuous_enable(port);
    if (retval != 0)
        return retval;

    return 0;
}
```

Receiving and forwarding packets

```
        RTE_ETH_FOREACH_DEV(port) {
            /* Get burst of RX packets, from first port of pair. */
            struct rte_mbuf *bufs[BURST_SIZE];
            const uint16_t nb_rx = rte_eth_rx_burst(port, 0,
                    bufs, BURST_SIZE);

            if (unlikely(nb_rx == 0))
```

```
                continue;

            /* Send burst of TX packets, to second port of pair. */
            const uint16_t nb_tx = rte_eth_tx_burst(port ^ 1, 0,
                    bufs, nb_rx);

            /* Free any unsent packets. */
            if (unlikely(nb_tx < nb_rx)) {
                uint16_t buf;
                for (buf = nb_tx; buf < nb_rx; buf++)
                    rte_pktmbuf_free(bufs[buf]);
            }
        }
```

Dive deeper

rte_eal_init() args

> https://doc.dpdk.org/guides/prog_guide/env_abstraction_layer.html
>
> https://doc.dpdk.org/guides/linux_gsg/linux_eal_parameters.html

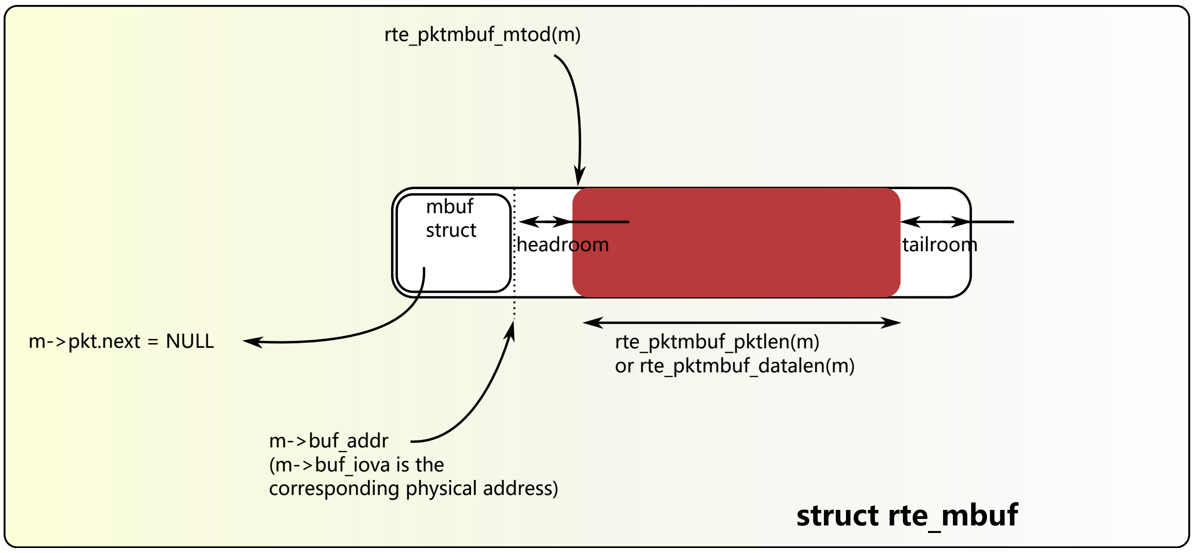
- `-c` : cpumask
- `-l` : lcore list
- `--main-lcore <core ID>`

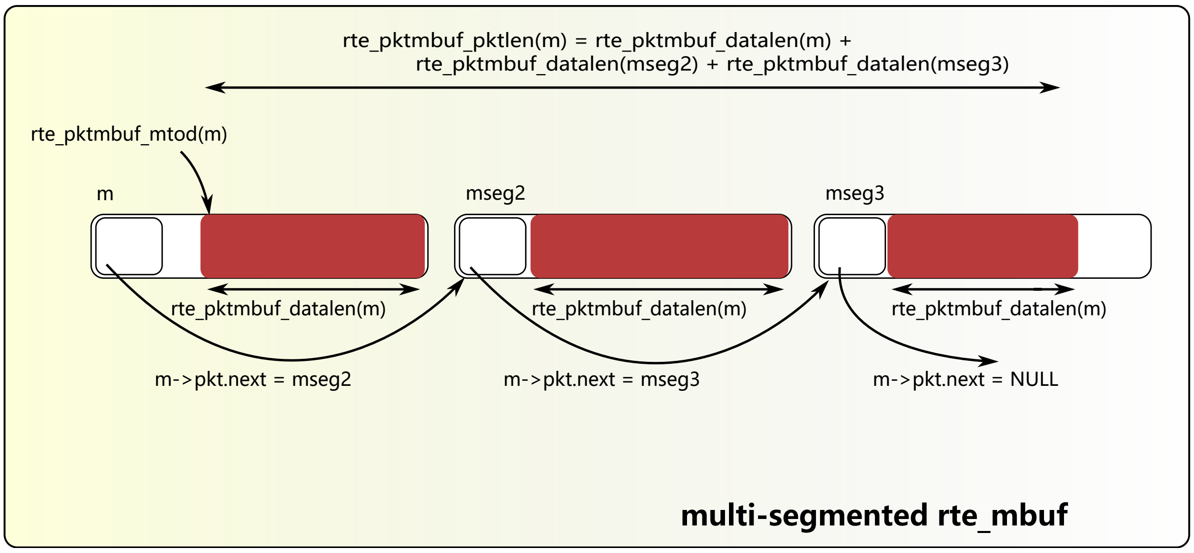
e.g.

```
./dpdk-helloworld -l 0-3
```

rte_mbuf

> http://doc.dpdk.org/guides-20.11/prog_guide/mbuf_lib.html





```
struct rte_mbuf *rte_pktmbuf_alloc(struct rte_mempool *mp);
void rte_pktmbuf_free(struct rte_mbuf *m);
```

mbuf from RX need to be manually freed.

successfully sent TX mbuf will be freed by DPDK.

mempool

```
/**
 * Create a mbuf pool.
 *
 * This function creates and initializes a packet mbuf pool. It is
 * a wrapper to rte_mempool functions.
 *
 * @param name
 *   The name of the mbuf pool.
 * @param n
 *   The number of elements in the mbuf pool. The optimum size (in terms
 *   of memory usage) for a mempool is when n is a power of two minus one:
 *   n = (2^q - 1).
 * @param cache_size
 *   Size of the per-core object cache. See rte_mempool_create() for
 *   details.
 * @param priv_size
 *   Size of application private are between the rte_mbuf structure
 *   and the data buffer. This value must be aligned to RTE_MBUF_PRIV_ALIGN.
 * @param data_room_size
 *   Size of data buffer in each mbuf, including RTE_PKTMBUF_HEADROOM.
 * @param socket_id
 *   The socket identifier where the memory should be allocated. The
 *   value can be *SOCKET_ID_ANY* if there is no NUMA constraint for the
 *   reserved zone.
 * @return
 *   The pointer to the new allocated mempool, on success. NULL on error
 *   with rte_errno set appropriately. Possible rte_errno values include:
 *    - E_RTE_NO_CONFIG - function could not get pointer to rte_config structure
 *    - E_RTE_SECONDARY - function was called from a secondary process instance
 *    - EINVAL - cache size provided is too large, or priv_size is not aligned.
 *    - ENOSPC - the maximum number of memzones has already been allocated
 *    - EEXIST - a memzone with the same name already exists
 *    - ENOMEM - no appropriate memory area found in which to create memzone
 */
struct rte_mempool *
rte_pktmbuf_pool_create(const char *name, unsigned n,
    unsigned cache_size, uint16_t priv_size, uint16_t data_room_size,
    int socket_id);
```

```
    /* Creates a new mempool in memory to hold the mbufs. */
    mbuf_pool = rte_pktmbuf_pool_create("MBUF_POOL", NUM_MBUFS * nb_ports,
        MBUF_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE, rte_socket_id());
    // MBUF_CACHE_SIZE = 250
```

```
unsigned rte_lcore_id();        // get lcore id
unsigned int rte_socket_id();   // get NUMA node id
int rte_thread_set_affinity(rte_cpuset_t *cpusetp)  // bind to lcores
```

dev config

```
int rte_eth_dev_configure(uint16_t port_id, uint16_t nb_rx_queue,
        uint16_t nb_tx_queue, const struct rte_eth_conf *eth_conf);

int rte_eth_rx_queue_setup(uint16_t port_id, uint16_t rx_queue_id,
        uint16_t nb_rx_desc, unsigned int socket_id,
        const struct rte_eth_rxconf *rx_conf,
        struct rte_mempool *mb_pool);

int rte_eth_tx_queue_setup(uint16_t port_id, uint16_t tx_queue_id,
        uint16_t nb_tx_desc, unsigned int socket_id,
        const struct rte_eth_txconf *tx_conf);
```

port_conf

```
// simple
static const struct rte_eth_conf port_conf_default = {
    .rxmode = { .max_rx_pkt_len = RTE_ETHER_MAX_LEN }  // 1518 = max frame length
including CRC
};

// added rss & offload
static uint8_t rss_key[] = {
    0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A, /* 10 */
    0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A, /* 20 */
    0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A, /* 30 */
    0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A, 0x6D, 0x5A /* 40 */
};

const struct rte_eth_conf port_conf_default = {
    .rxmode = {
        .mq_mode = ETH_MQ_RX_RSS,
        .split_hdr_size = 0,
    },
    .rx_adv_conf = {
        .rss_conf = {
            .rss_key = rss_key,
            .rss_key_len = sizeof(rss_key),
            .rss_hf =
                ETH_RSS_FRAG_IPV4          |
                ETH_RSS_NONFRAG_IPV4_UDP
        }
    },
    .txmode = {
        .offloads =
            DEV_TX_OFFLOAD_VLAN_INSERT |
            DEV_TX_OFFLOAD_IPV4_CKSUM  |
            DEV_TX_OFFLOAD_UDP_CKSUM   |
            DEV_TX_OFFLOAD_TCP_CKSUM   |
            DEV_TX_OFFLOAD_SCTP_CKSUM  |
            DEV_TX_OFFLOAD_TCP_TSO,
    }};
```

> to correctly offload checksum, you also need to set `mbuf->ol_flags, mbuf->l2_len, mbuf->l3_len` correctly, see **https://doc.dpdk.org/guides/prog_guide/mbuf_lib.html#meta-information** for more info.

dev_info

```
int rte_eth_dev_info_get(uint16_t port_id, struct rte_eth_dev_info *dev_info);
```

```
dev_info->default_rxportconf
dev_info->default_txportconf
```

Also see

DPDK Getting Started Guide for Linux: **https://doc.dpdk.org/guides/linux_gsg/index.html**

DPDK Programmer's Guide: **https://doc.dpdk.org/guides/prog_guide/index.html**

DPDK API Guide: **https://doc.dpdk.org/api/**

DPDK Efficient Code Guide: **https://doc.dpdk.org/guides/prog_guide/writing_efficient_code.html**

Multi-process support: **https://doc.dpdk.org/guides/prog_guide/multi_proc_support.html#multi-process-support**

rte_hash API: **https://doc.dpdk.org/guides/prog_guide/hash_lib.html**

rte_flow API: **https://doc.dpdk.org/guides/prog_guide/rte_flow.html**