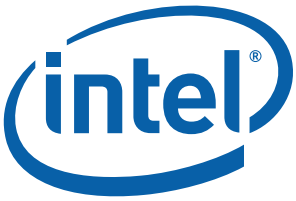

Intel® Data Plane Development Kit (Intel® DPDK)

API Reference

January 2014

Reference Number: 326004-006

Contents

Chapter 1

Deprecated List

Global rte_lpm::mem_location

Global RTE_LPM_HEAP

Possible location to allocate memory. This was for last parameter of rte_lpm_create(), but is now redundant. The LPM table is always allocated in memory using librte_malloc which uses a memzone.

Global RTE_LPM_MEMZONE

Possible location to allocate memory. This was for last parameter of rte_lpm_create(), but is now redundant. The LPM table is always allocated in memory using librte_malloc which uses a memzone.

Chapter 2

Data Structure Documentation

2.1 rte_ring::cons Struct Reference

Data Fields

- uint32_t sc_dequeue
- uint32_t size
- uint32_t mask
- volatile uint32_t head
- volatile uint32_t tail

2.1.1 Detailed Description

Ring consumer status.

2.1.2 Field Documentation

2.1.2.1 uint32_t rte_ring::cons::sc_dequeue

True, if single consumer.

2.1.2.2 uint32_t rte_ring::cons::size

Size of the ring.

2.1.2.3 uint32_t rte_ring::cons::mask

Mask (size-1) of ring.

2.1.2.4 volatile uint32_t rte_ring::cons::head

Consumer head.

2.1.2.5 volatile uint32_t rte_ring::cons::tail

Consumer tail.

2.2 eth_dev_ops Struct Reference

Data Fields

- eth_dev_configure_t dev_configure
- eth_dev_start_t dev_start
- eth_dev_stop_t dev_stop
- eth_dev_close_t dev_close
- eth_promiscuous_enable_t promiscuous_enable
- eth_promiscuous_disable_t promiscuous_disable
- eth_allmulticast_enable_t allmulticast_enable
- eth_allmulticast_disable_t allmulticast_disable
- eth_link_update_t link_update
- eth_stats_get_t stats_get
- eth_stats_reset_t stats_reset
- eth_queue_stats_mapping_set_t queue_stats_mapping_set
- eth_dev_infos_get_t dev_infos_get
- vlan_filter_set_t vlan_filter_set
- vlan_tpid_set_t vlan_tpid_set
- vlan_strip_queue_set_t vlan_strip_queue_set
- vlan_offload_set_t vlan_offload_set
- eth_rx_queue_setup_t rx_queue_setup
- eth_queue_release_t rx_queue_release
- eth_rx_queue_count_t rx_queue_count
- eth_rx_descriptor_done_t rx_descriptor_done
- eth_tx_queue_setup_t tx_queue_setup
- eth_queue_release_t tx_queue_release
- eth_dev_led_on_t dev_led_on
- eth_dev_led_off_t dev_led_off
- flow_ctrl_set_t flow_ctrl_set
- priority_flow_ctrl_set_t priority_flow_ctrl_set
- eth_mac_addr_remove_t mac_addr_remove
- eth_mac_addr_add_t mac_addr_add
- eth_uc_hash_table_set_t uc_hash_table_set
- eth_uc_all_hash_table_set_t uc_all_hash_table_set
- eth_mirror_rule_set_t mirror_rule_set

- eth_mirror_rule_reset_t mirror_rule_reset
- eth_set_vf_rx_mode_t set_vf_rx_mode
- eth_set_vf_rx_t set_vf_rx
- eth_set_vf_tx_t set_vf_tx
- eth_set_vf_vlan_filter_t set_vf_vlan_filter
- fdir_add_signature_filter_t fdir_add_signature_filter
- fdir_update_signature_filter_t fdir_update_signature_filter
- fdir_remove_signature_filter_t fdir_remove_signature_filter
- fdir_infos_get_t fdir_infos_get
- fdir_add_perfect_filter_t fdir_add_perfect_filter
- fdir_update_perfect_filter_t fdir_update_perfect_filter
- fdir_remove_perfect_filter_t fdir_remove_perfect_filter
- fdir_set_masks_t fdir_set_masks
- reta_update_t reta_update
- reta_query_t reta_query

2.2.1 Field Documentation

2.2.1.1 eth_dev_configure_t eth_dev_ops::dev_configure

Configure device.

2.2.1.2 eth_dev_start_t eth_dev_ops::dev_start

Start device.

2.2.1.3 eth_dev_stop_t eth_dev_ops::dev_stop

Stop device.

2.2.1.4 eth_dev_close_t eth_dev_ops::dev_close

Close device.

2.2.1.5 eth_promiscuous_enable_t eth_dev_ops::promiscuous_enable

Promiscuous ON.

2.2.1.6 eth_promiscuous_disable_t eth_dev_ops::promiscuous_disable

Promiscuous OFF.

2.2.1.7 eth_allmulticast_enable_t eth_dev_ops::allmulticast_enable

RX multicast ON.

2.2.1.8 eth_allmulticast_disable_t eth_dev_ops::allmulticast_disable

RX multicast OF.

2.2.1.9 eth_link_update_t eth_dev_ops::link_update

Get device link state.

2.2.1.10 eth_stats_get_t eth_dev_ops::stats_get

Get device statistics.

2.2.1.11 eth_stats_reset_t eth_dev_ops::stats_reset

Reset device statistics.

2.2.1.12 eth_queue_stats_mapping_set_t eth_dev_ops::queue_stats_mapping_set

Configure per queue stat counter mapping.

2.2.1.13 eth_dev_infos_get_t eth_dev_ops::dev_infos_get

Get device info.

2.2.1.14 vlan_filter_set_t eth_dev_ops::vlan_filter_set

Filter VLAN Setup.

2.2.1.15 vlan_tpid_set_t eth_dev_ops::vlan_tpid_set

Outer VLAN TPID Setup.

2.2.1.16 vlan_strip_queue_set_t eth_dev_ops::vlan_strip_queue_set

VLAN Stripping on queue.

2.2.1.17 vlan_offload_set_t eth_dev_ops::vlan_offload_set

Set VLAN Offload.

2.2.1.18 eth_rx_queue_setup_t eth_dev_ops::rx_queue_setup

Set up device RX queue.

2.2.1.19 eth_queue_release_t eth_dev_ops::rx_queue_release

Release RX queue.

2.2.1.20 eth_rx_queue_count_t eth_dev_ops::rx_queue_count

Get Rx queue count.

2.2.1.21 eth_rx_descriptor_done_t eth_dev_ops::rx_descriptor_done

Check rxd DD bit

2.2.1.22 eth_tx_queue_setup_t eth_dev_ops::tx_queue_setup

Set up device TX queue.

2.2.1.23 eth_queue_release_t eth_dev_ops::tx_queue_release

Release TX queue.

2.2.1.24 eth_dev_led_on_t eth_dev_ops::dev_led_on

Turn on LED.

2.2.1.25 eth_dev_led_off_t eth_dev_ops::dev_led_off

Turn off LED.

2.2.1.26 flow_ctrl_set_t eth_dev_ops::flow_ctrl_set

Setup flow control.

2.2.1.27 priority_flow_ctrl_set_t eth_dev_ops::priority_flow_ctrl_set

Setup priority flow control.

2.2.1.28 eth_mac_addr_remove_t eth_dev_ops::mac_addr_remove

Remove MAC address

2.2.1.29 eth_mac_addr_add_t eth_dev_ops::mac_addr_add

Add a MAC address

2.2.1.30 eth_uc_hash_table_set_t eth_dev_ops::uc_hash_table_set

Set Unicast Table Array

2.2.1.31 eth_uc_all_hash_table_set_t eth_dev_ops::uc_all_hash_table_set

Set Unicast hash bitmap

2.2.1.32 eth_mirror_rule_set_t eth_dev_ops::mirror_rule_set

Add a traffic mirror rule.

2.2.1.33 eth_mirror_rule_reset_t eth_dev_ops::mirror_rule_reset

reset a traffic mirror rule.

2.2.1.34 eth_set_vf_rx_mode_t eth_dev_ops::set_vf_rx_mode

Set VF RX mode

2.2.1.35 eth_set_vf_rx_t eth_dev_ops::set_vf_rx

enable/disable a VF receive

2.2.1.36 eth_set_vf_tx_t eth_dev_ops::set_vf_tx

enable/disable a VF transmit

2.2.1.37 eth_set_vf_vlan_filter_t eth_dev_ops::set_vf_vlan_filter

Set VF VLAN filter

2.2.1.38 fdir_add_signature_filter_t eth_dev_ops::fdir_add_signature_filter

Add a signature filter.

2.2.1.39 fdir_update_signature_filter_t eth_dev_ops::fdir_update_signature_filter

Update a signature filter.

2.2.1.40 fdir_remove_signature_filter_t eth_dev_ops::fdir_remove_signature_filter

Remove a signature filter.

2.2.1.41 fdir_infos_get_t eth_dev_ops::fdir_infos_get

Get information about FDIR status.

2.2.1.42 fdir_add_perfect_filter_t eth_dev_ops::fdir_add_perfect_filter

Add a perfect filter.

2.2.1.43 fdir_update_perfect_filter_t eth_dev_ops::fdir_update_perfect_filter

Update a perfect filter.

2.2.1.44 fdir_remove_perfect_filter_t eth_dev_ops::fdir_remove_perfect_filter

Remove a perfect filter.

2.2.1.45 fdir_set_masks_t eth_dev_ops::fdir_set_masks

Setup masks for FDIR filtering.

2.2.1.46 reta_update_t eth_dev_ops::reta_update

Update redirection table.

2.2.1.47 reta_query_t eth_dev_ops::reta_query

Query redirection table.

2.3 eth_driver Struct Reference

Data Fields

- struct rte_pci_driver pci_drv
- eth_dev_init_t eth_dev_init
- unsigned int dev_private_size

2.3.1 Field Documentation

2.3.1.1 struct rte_pci_driver eth_driver::pci_drv

The PMD is also a PCI driver.

2.3.1.2 eth_dev_init_t eth_driver::eth_dev_init

Device init function.

2.3.1.3 unsigned int eth_driver::dev_private_size

Size of device private data.

2.4 ether_addr Struct Reference

Data Fields

- uint8_t addr_bytes [ETHER_ADDR_LEN]

2.4.1 Detailed Description

Ethernet address: A universally administered address is uniquely assigned to a device by its manufacturer. The first three octets (in transmission order) contain the Organizationally Unique Identifier (OUI). The following three (MAC-48 and EUI-48) octets are assigned by that organization with the only constraint of uniqueness. A locally administered address is assigned to a device by a network administrator and does not contain OUIs. See http://standards.ieee.org/regauth/groupmac/tutorial.html

2.4.2 Field Documentation

2.4.2.1 uint8_t ether_addr::addr_bytes[ETHER_ADDR_LEN]

Address bytes in transmission order

2.5 ether_hdr Struct Reference

Data Fields

- struct ether_addr d_addr
- struct ether_addr s_addr
- uint16_t ether_type

2.5.1 Detailed Description

Ethernet header: Contains the destination address, source address and frame type.

2.5.2 Field Documentation

2.5.2.1 struct ether_addr ether_hdr::d_addr

Destination address.

2.5.2.2 struct ether_addr ether_hdr::s_addr

Source address.

2.5.2.3 uint16_t ether_hdr::ether_type

Frame type.

2.6 ipv4_hdr Struct Reference

Data Fields

- uint8_t version_ihl
- uint8_t type_of_service
- uint16_t total_length
- uint16_t packet_id

- uint16_t fragment_offset
- uint8_t time_to_live
- uint8_t next_proto_id
- uint16_t hdr_checksum
- uint32_t src_addr
- uint32_t dst_addr

2.6.1 Detailed Description

IPv4 Header

2.6.2 Field Documentation

2.6.2.1 uint8_t ipv4_hdr::version_ihl

version and header length

2.6.2.2 uint8_t ipv4_hdr::type_of_service

type of service

2.6.2.3 uint16_t ipv4_hdr::total_length

length of packet

2.6.2.4 uint16_t ipv4_hdr::packet_id

packet ID

2.6.2.5 uint16_t ipv4_hdr::fragment_offset

fragmentation offset

2.6.2.6 uint8_t ipv4_hdr::time_to_live

time to live

2.6.2.7 uint8_t ipv4_hdr::next_proto_id

protocol ID

2.6.2.8 uint16_t ipv4_hdr::hdr_checksum

header checksum

2.6.2.9 uint32_t ipv4_hdr::src_addr

source address

2.6.2.10 uint32_t ipv4_hdr::dst_addr

destination address

2.7 ipv6_hdr Struct Reference

Data Fields

- uint32_t vtc_flow
- uint16_t payload_len
- uint8_t proto
- uint8_t hop_limits
- uint8_t src_addr [16]
- uint8_t dst_addr [16]

2.7.1 Detailed Description

IPv6 Header

2.7.2 Field Documentation

2.7.2.1 uint32_t ipv6_hdr::vtc_flow

IP version, traffic class & flow label.

2.7.2.2 uint16_t ipv6_hdr::payload_len

IP packet length - includes sizeof(ip_header).

2.7.2.3 uint8_t ipv6_hdr::proto

Protocol, next header.

2.7.2.4 uint8_t ipv6_hdr::hop_limits

Hop limits.

2.7.2.5 uint8_t ipv6_hdr::src_addr[16]

IP address of source host.

2.7.2.6 uint8_t ipv6_hdr::dst_addr[16]

IP address of destination host(s).

2.8 malloc_heap Struct Reference

2.8.1 Detailed Description

Structure to hold malloc heap

2.9 rte_ring::prod Struct Reference

Data Fields

- uint32_t watermark
- uint32_t sp_enqueue
- uint32_t size
- uint32_t mask
- volatile uint32_t head
- volatile uint32_t tail

2.9.1 Detailed Description

Ring producer status.

2.9.2 Field Documentation

2.9.2.1 uint32_t rte_ring::prod::watermark

Maximum items before EDQUOT.

2.9.2.2 uint32_t rte_ring::prod::sp_enqueue

True, if single producer.

2.9.2.3 uint32_t rte_ring::prod::size

Size of ring.

2.9.2.4 uint32_t rte_ring::prod::mask

Mask (size-1) of ring.

2.9.2.5 volatile uint32_t rte_ring::prod::head

Producer head.

2.9.2.6 volatile uint32_t rte_ring::prod::tail

Producer tail.

2.10 rte_atomic16_t Struct Reference

Data Fields

- volatile int16_t cnt

2.10.1 Detailed Description

The atomic counter structure.

2.10.2 Field Documentation

2.10.2.1 volatile int16_t rte_atomic16_t::cnt

An internal counter value.

2.11 rte_atomic32_t Struct Reference

Data Fields

- volatile int32_t cnt

2.11.1 Detailed Description

The atomic counter structure.

2.11.2 Field Documentation

2.11.2.1 volatile int32_t rte_atomic32_t::cnt

An internal counter value.

2.12 rte_atomic64_t Struct Reference

Data Fields

- volatile int64_t cnt

2.12.1 Detailed Description

The atomic counter structure.

2.12.2 Field Documentation

2.12.2.1 volatile int64_t rte_atomic64_t::cnt

Internal counter value.

2.13 rte_config Struct Reference

Data Fields

- uint32_t version
- uint32_t magic
- uint32_t master_lcore

- uint32_t lcore_count
- enum rte_lcore_role_t lcore_role [32,]
- enum rte_proc_type_t process_type
- unsigned flags
- struct rte_mem_config * mem_config

2.13.1 Detailed Description

The global RTE configuration structure.

2.13.2 Field Documentation

2.13.2.1 uint32_t rte_config::version

Configuration [structure] version.

2.13.2.2 uint32_t rte_config::magic

Magic number - Sanity check.

2.13.2.3 uint32_t rte_config::master_lcore

Id of the master lcore

2.13.2.4 uint32_t rte_config::lcore_count

Number of available logical cores.

2.13.2.5 enum rte_lcore_role_t rte_config::lcore_role[32,]

State of cores.

2.13.2.6 enum rte_proc_type_t rte_config::process_type

Primary or secondary configuration

2.13.2.7 unsigned rte_config::flags

A set of general status flags

2.13.2.8 struct rte_mem_config* rte_config::mem_config

Pointer to memory configuration, which may be shared across multiple Intel DPDK instances

2.14 rte_ctrlmbuf Struct Reference

Data Fields

- void * data
- uint32_t data_len

2.14.1 Detailed Description

A control message buffer.

2.14.2 Field Documentation

2.14.2.1 void* rte_ctrlmbuf::data

Pointer to data.

2.14.2.2 uint32_t rte_ctrlmbuf::data_len

Length of data.

2.15 rte_dummy Struct Reference

Data Fields

- TAILQ_ENTRY next

2.15.1 Detailed Description

dummy structure type used by the rte_tailq APIs

2.15.2 Field Documentation

2.15.2.1 TAILQ_ENTRY rte_dummy::next

Pointer entries for a tailq list

2.16 rte_eth_conf Struct Reference

Data Fields

- uint16_t link_speed
- uint16_t link_duplex
- struct rte_eth_rxmode rxmode
- struct rte_eth_txmode txmode
- uint32_t lpbk_mode
- union {
  struct rte_eth_rss_conf rss_conf
  struct rte_eth_vmdq_dcb_conf vmdq_dcb_conf
  struct rte_eth_dcb_rx_conf dcb_rx_conf
  struct rte_eth_vmdq_rx_conf vmdq_rx_conf
  } rx_adv_conf

- union {
  struct rte_eth_vmdq_dcb_tx_conf vmdq_dcb_tx_conf
  struct rte_eth_dcb_tx_conf dcb_tx_conf
  struct rte_eth_vmdq_tx_conf vmdq_tx_conf
  } tx_adv_conf

- uint32_t dcb_capability_en
- struct rte_fdir_conf fdir_conf
- struct rte_intr_conf intr_conf

2.16.1 Detailed Description

A structure used to configure an Ethernet port. Depending upon the RX multi-queue mode, extra advanced configuration settings may be needed.

2.16.2 Field Documentation

2.16.2.1 uint16_t rte_eth_conf::link_speed

ETH_LINK_SPEED_10[0|00|000], or 0 for autonegotation

2.16.2.2 uint16_t rte_eth_conf::link_duplex

ETH_LINK_[HALF_DUPLEX|FULL_DUPLEX], or 0 for autonegotation

2.16.2.3 struct rte_eth_rxmode rte_eth_conf::rxmode

Port RX configuration.

2.16.2.4 struct rte_eth_txmode rte_eth_conf::txmode

Port TX configuration.

2.16.2.5 uint32_t rte_eth_conf::lpbk_mode

Loopback operation mode. By default the value is 0, meaning the loopback mode is disabled. Read the datasheet of given ethernet controller for details. The possible values of this field are defined in implementation of each driver.

2.16.2.6 struct rte_eth_rss_conf rte_eth_conf::rss_conf

Port RSS configuration

2.16.2.7 struct rte_eth_vmdq_dcb_conf rte_eth_conf::vmdq_dcb_conf

Port vmdq+dcb configuration.

2.16.2.8 struct rte_eth_dcb_rx_conf rte_eth_conf::dcb_rx_conf

Port dcb RX configuration.

2.16.2.9 struct rte_eth_vmdq_rx_conf rte_eth_conf::vmdq_rx_conf

Port vmdq RX configuration.

2.16.2.10 union { ... } rte_eth_conf::rx_adv_conf

Port RX filtering configuration (union).

2.16.2.11 struct rte_eth_vmdq_dcb_tx_conf rte_eth_conf::vmdq_dcb_tx_conf

Port vmdq+dcb TX configuration.

2.16.2.12 struct rte_eth_dcb_tx_conf rte_eth_conf::dcb_tx_conf

Port dcb TX configuration.

2.16.2.13 struct rte_eth_vmdq_tx_conf rte_eth_conf::vmdq_tx_conf

Port vmdq TX configuration.

2.16.2.14 union { ... } rte_eth_conf::tx_adv_conf

Port TX DCB configuration (union).

2.16.2.15 uint32_t rte_eth_conf::dcb_capability_en

Currently,Priority Flow Control(PFC) are supported,if DCB with PFC is needed,and the variable must be set ETH_DCB_PFC_SUPPORT.

2.16.2.16 struct rte_fdir_conf rte_eth_conf::fdir_conf

FDIR configuration.

2.16.2.17 struct rte_intr_conf rte_eth_conf::intr_conf

Interrupt mode configuration.

2.17 rte_eth_dcb_rx_conf Struct Reference

Data Fields

- enum rte_eth_nb_tcs nb_tcs
- uint8_t dcb_queue [ETH_DCB_NUM_USER_PRIORITIES]

2.17.1 Field Documentation

2.17.1.1 enum rte_eth_nb_tcs rte_eth_dcb_rx_conf::nb_tcs

Possible DCB TCs, 4 or 8 TCs

2.17.1.2 uint8_t rte_eth_dcb_rx_conf::dcb_queue[ETH_DCB_NUM_USER_PRIORITIES]

Possible DCB queue,4 or 8.

2.18 rte_eth_dcb_tx_conf Struct Reference

Data Fields

- enum rte_eth_nb_tcs nb_tcs
- uint8_t dcb_queue [ETH_DCB_NUM_USER_PRIORITIES]

2.18.1 Field Documentation

2.18.1.1 enum rte_eth_nb_tcs rte_eth_dcb_tx_conf::nb_tcs

Possible DCB TCs, 4 or 8 TCs.

2.18.1.2 uint8_t rte_eth_dcb_tx_conf::dcb_queue[ETH_DCB_NUM_USER_PRIORITIES]

Possible DCB queue,4 or 8.

2.19 rte_eth_dev Struct Reference

Data Fields

- eth_rx_burst_t rx_pkt_burst
- eth_tx_burst_t tx_pkt_burst
- struct rte_eth_dev_data * data
- struct eth_driver * driver
- struct eth_dev_ops * dev_ops
- struct rte_pci_device * pci_dev
- struct rte_eth_dev_cb_list callbacks

2.19.1 Field Documentation

2.19.1.1 eth_rx_burst_t rte_eth_dev::rx_pkt_burst

Pointer to PMD receive function.

2.19.1.2 eth_tx_burst_t rte_eth_dev::tx_pkt_burst

Pointer to PMD transmit function.

2.19.1.3 struct rte_eth_dev_data* rte_eth_dev::data

Pointer to device data

2.19.1.4 struct eth_driver* rte_eth_dev::driver

Driver for this device

2.19.1.5 struct eth_dev_ops* rte_eth_dev::dev_ops

Functions exported by PMD

2.19.1.6 struct rte_pci_device* rte_eth_dev::pci_dev

PCI info. supplied by probing

2.19.1.7 struct rte_eth_dev_cb_list rte_eth_dev::callbacks

User application callbacks

2.20 rte_eth_dev_data Struct Reference

Data Fields

- void ** rx_queues
- void ** tx_queues
- uint16_t nb_rx_queues
- uint16_t nb_tx_queues
- struct rte_eth_dev_sriov sriov
- void * dev_private
- struct rte_eth_link dev_link
- struct rte_eth_conf dev_conf
- uint16_t max_frame_size
- uint64_t rx_mbuf_alloc_failed
- struct ether_addr * mac_addrs
- struct ether_addr * hash_mac_addrs
- uint8_t port_id
- uint8_t promiscuous: 1
- uint8_t scattered_rx: 1
- uint8_t all_multicast: 1
- uint8_t dev_started: 1

2.20.1 Field Documentation

2.20.1.1 void** rte_eth_dev_data::rx_queues

Array of pointers to RX queues.

2.20.1.2 void** rte_eth_dev_data::tx_queues

Array of pointers to TX queues.

2.20.1.3 uint16_t rte_eth_dev_data::nb_rx_queues

Number of RX queues.

2.20.1.4 uint16_t rte_eth_dev_data::nb_tx_queues

Number of TX queues.

2.20.1.5 struct rte_eth_dev_sriov rte_eth_dev_data::sriov

SRIOV data

2.20.1.6 void* rte_eth_dev_data::dev_private

PMD-specific private data

2.20.1.7 struct rte_eth_link rte_eth_dev_data::dev_link

Link-level information & status

2.20.1.8 struct rte_eth_conf rte_eth_dev_data::dev_conf

Configuration applied to device.

2.20.1.9 uint16_t rte_eth_dev_data::max_frame_size

Default is ETHER_MAX_LEN (1518).

2.20.1.10 uint64_t rte_eth_dev_data::rx_mbuf_alloc_failed

RX ring mbuf allocation failures.

2.20.1.11 struct ether_addr* rte_eth_dev_data::mac_addrs

Device Ethernet Link address.

2.20.1.12 struct ether_addr* rte_eth_dev_data::hash_mac_addrs

bitmap array of associating Ethernet MAC addresses to pools

2.20.1.13 uint8_t rte_eth_dev_data::port_id

Device Ethernet MAC addresses of hash filtering. Device [external] port identifier.

2.20.1.14 uint8_t rte_eth_dev_data::promiscuous

RX promiscuous mode ON(1) / OFF(0).

2.20.1.15 uint8_t rte_eth_dev_data::scattered_rx

RX of scattered packets is ON(1) / OFF(0)

2.20.1.16 uint8_t rte_eth_dev_data::all_multicast

RX all multicast mode ON(1) / OFF(0).

2.20.1.17 uint8_t rte_eth_dev_data::dev_started

Device state: STARTED(1) / STOPPED(0).

2.21 rte_eth_dev_info Struct Reference

Data Fields

- struct rte_pci_device * pci_dev
- const char * driver_name
- uint32_t min_rx_bufsize
- uint32_t max_rx_pktlen
- uint16_t max_rx_queues
- uint16_t max_tx_queues
- uint32_t max_mac_addrs
- uint16_t max_vfs
- uint16_t max_vmdq_pools

2.21.1 Detailed Description

A structure used to retrieve the contextual information of an Ethernet device, such as the controlling driver of the device, its PCI context, etc...

2.21.2 Field Documentation

2.21.2.1 struct rte_pci_device* rte_eth_dev_info::pci_dev

Device PCI information.

2.21.2.2 const char* rte_eth_dev_info::driver_name

Device Driver name.

2.21.2.3 uint32_t rte_eth_dev_info::min_rx_bufsize

Minimum size of RX buffer.

2.21.2.4 uint32_t rte_eth_dev_info::max_rx_pktlen

Maximum configurable length of RX pkt.

2.21.2.5 uint16_t rte_eth_dev_info::max_rx_queues

Maximum number of RX queues.

2.21.2.6 uint16_t rte_eth_dev_info::max_tx_queues

Maximum number of TX queues.

2.21.2.7 uint32_t rte_eth_dev_info::max_mac_addrs

Maximum number of MAC addresses.

2.21.2.8 uint16_t rte_eth_dev_info::max_vfs

Maximum number of hash MAC addresses for MTA and UTA. Maximum number of VFs.

2.21.2.9 uint16_t rte_eth_dev_info::max_vmdq_pools

Maximum number of VMDq pools.

2.22 rte_eth_dev_sriov Struct Reference

Data Fields

- uint8_t active
- uint8_t nb_q_per_pool
- uint16_t def_vmdq_idx
- uint16_t def_pool_q_idx

2.22.1 Field Documentation

2.22.1.1 uint8_t rte_eth_dev_sriov::active

SRIOV is active with 16, 32 or 64 pools

2.22.1.2 uint8_t rte_eth_dev_sriov::nb_q_per_pool

rx queue number per pool

2.22.1.3 uint16_t rte_eth_dev_sriov::def_vmdq_idx

Default pool num used for PF

2.22.1.4 uint16_t rte_eth_dev_sriov::def_pool_q_idx

Default pool queue start reg index

2.23 rte_eth_fc_conf Struct Reference

Data Fields

- uint32_t high_water
- uint32_t low_water
- uint16_t pause_time
- uint16_t send_xon
- enum rte_eth_fc_mode mode
- uint8_t mac_ctrl_frame_fwd

2.23.1 Detailed Description

A structure used to configure Ethernet flow control parameter. These parameters will be configured into the register of the NIC. Please refer to the corresponding data sheet for proper value.

2.23.2 Field Documentation

2.23.2.1 uint32_t rte_eth_fc_conf::high_water

High threshold value to trigger XOFF

2.23.2.2 uint32_t rte_eth_fc_conf::low_water

Low threshold value to trigger XON

2.23.2.3 uint16_t rte_eth_fc_conf::pause_time

Pause quota in the Pause frame

2.23.2.4 uint16_t rte_eth_fc_conf::send_xon

Is XON frame need be sent

2.23.2.5 enum rte_eth_fc_mode rte_eth_fc_conf::mode

Link flow control mode

2.23.2.6 uint8_t rte_eth_fc_conf::mac_ctrl_frame_fwd

Forward MAC control frames

2.24 rte_eth_fdir Struct Reference

Data Fields

- uint16_t collision
- uint16_t free
- uint16_t maxhash
- uint8_t maxlen
- uint64_t add
- uint64_t remove
- uint64_t f_add
- uint64_t f_remove

2.24.1 Detailed Description

A structure used to report the status of the flow director filters in use.

2.24.2 Field Documentation

2.24.2.1 uint16_t rte_eth_fdir::collision

Number of filters with collision indication.

2.24.2.2 uint16_t rte_eth_fdir::free

Number of free (non programmed) filters.

2.24.2.3 uint16_t rte_eth_fdir::maxhash

The Lookup hash value of the added filter that updated the value of the MAXLEN field

2.24.2.4 uint8_t rte_eth_fdir::maxlen

Longest linked list of filters in the table.

2.24.2.5 uint64_t rte_eth_fdir::add

Number of added filters.

2.24.2.6 uint64_t rte_eth_fdir::remove

Number of removed filters.

2.24.2.7 uint64_t rte_eth_fdir::f_add

Number of failed added filters (no more space in device).

2.24.2.8 uint64_t rte_eth_fdir::f_remove

Number of failed removed filters.

2.25 rte_eth_link Struct Reference

Data Fields

- uint16_t link_speed
- uint16_t link_duplex
- uint8_t link_status: 1

2.25.1 Detailed Description

A structure used to retrieve link-level information of an Ethernet port. aligned for atomic64 read/write

2.25.2 Field Documentation

2.25.2.1 uint16_t rte_eth_link::link_speed

ETH_LINK_SPEED_[10, 100, 1000, 10000]

2.25.2.2 uint16_t rte_eth_link::link_duplex

ETH_LINK_[HALF_DUPLEX, FULL_DUPLEX]

2.25.2.3 uint8_t rte_eth_link::link_status

1 -> link up, 0 -> link down

2.26 rte_eth_pfc_conf Struct Reference

Data Fields

- struct rte_eth_fc_conf fc
- uint8_t priority

2.26.1 Detailed Description

A structure used to configure Ethernet priority flow control parameter. These parameters will be configured into the register of the NIC. Please refer to the corresponding data sheet for proper value.

2.26.2 Field Documentation

2.26.2.1 struct rte_eth_fc_conf rte_eth_pfc_conf::fc

General flow control parameter.

2.26.2.2 uint8_t rte_eth_pfc_conf::priority

VLAN User Priority.

2.27 rte_eth_rss_conf Struct Reference

Data Fields

- uint8_t ∗ rss_key
- uint16_t rss_hf

2.27.1 Detailed Description

A structure used to configure the Receive Side Scaling (RSS) feature of an Ethernet port. If not NULL, the ∗rss_key∗ pointer of the ∗rss_conf∗ structure points to an array of 40 bytes holding the RSS key to use for hashing specific header fields of received packets. Otherwise, a default random hash key is used by the device driver.

The ∗rss_hf∗ field of the ∗rss_conf∗ structure indicates the different types of IPv4/IPv6 packets to which the RSS hashing must be applied. Supplying an ∗rss_hf∗ equal to zero disables the RSS feature.

2.27.2 Field Documentation

2.27.2.1 uint8_t∗ rte_eth_rss_conf::rss_key

If not NULL, 40-byte hash key.

2.27.2.2 uint16_t rte_eth_rss_conf::rss_hf

Hash functions to apply - see below.

2.28 rte_eth_rss_reta Struct Reference

Data Fields

- uint64_t mask_lo
- uint64_t mask_hi
- uint8_t reta [ETH_RSS_RETA_NUM_ENTRIES]

2.28.1 Detailed Description

A structure used to configure Redirection Table of the Receive Side Scaling (RSS) feature of an Ethernet port.

2.28.2 Field Documentation

2.28.2.1 uint64_t rte_eth_rss_reta::mask_lo

First 64 mask bits indicate which entry(s) need to updated/queried.

2.28.2.2 uint64_t rte_eth_rss_reta::mask_hi

Second 64 mask bits indicate which entry(s) need to updated/queried.

2.28.2.3 uint8_t rte_eth_rss_reta::reta[ETH_RSS_RETA_NUM_ENTRIES]

128 RETA entries

2.29 rte_eth_rxconf Struct Reference

Data Fields

- struct rte_eth_thresh rx_thresh
- uint16_t rx_free_thresh
- uint8_t rx_drop_en

2.29.1 Detailed Description

A structure used to configure an RX ring of an Ethernet port.

2.29.2 Field Documentation

2.29.2.1 struct rte_eth_thresh rte_eth_rxconf::rx_thresh

RX ring threshold registers.

2.29.2.2 uint16_t rte_eth_rxconf::rx_free_thresh

Drives the freeing of RX descriptors.

2.29.2.3 uint8_t rte_eth_rxconf::rx_drop_en

Drop packets if no descriptors are available.

2.30 rte_eth_rxmode Struct Reference

Data Fields

- enum rte_eth_rx_mq_mode mq_mode
- uint32_t max_rx_pkt_len
- uint16_t split_hdr_size
- uint8_t header_split: 1
- uint8_t hw_ip_checksum: 1
- uint8_t hw_vlan_filter: 1
- uint8_t hw_vlan_strip: 1
- uint8_t hw_vlan_extend: 1
- uint8_t jumbo_frame: 1
- uint8_t hw_strip_crc: 1

2.30.1 Detailed Description

A structure used to configure the RX features of an Ethernet port.

2.30.2 Field Documentation

2.30.2.1 enum rte_eth_rx_mq_mode rte_eth_rxmode::mq_mode

The multi-queue packet distribution mode to be used, e.g. RSS.

2.30.2.2 uint32_t rte_eth_rxmode::max_rx_pkt_len

Only used if jumbo_frame enabled.

2.30.2.3 uint16_t rte_eth_rxmode::split_hdr_size

hdr buf size (header_split enabled).

2.30.2.4 uint8_t rte_eth_rxmode::header_split

Header Split enable.

2.30.2.5 uint8_t rte_eth_rxmode::hw_ip_checksum

IP/UDP/TCP checksum offload enable.

2.30.2.6 uint8_t rte_eth_rxmode::hw_vlan_filter

VLAN filter enable.

2.30.2.7 uint8_t rte_eth_rxmode::hw_vlan_strip

VLAN strip enable.

2.30.2.8 uint8_t rte_eth_rxmode::hw_vlan_extend

Extended VLAN enable.

2.30.2.9 uint8_t rte_eth_rxmode::jumbo_frame

Jumbo Frame Receipt enable.

2.30.2.10 uint8_t rte_eth_rxmode::hw_strip_crc

Enable CRC stripping by hardware.

2.31 rte_eth_stats Struct Reference

Data Fields

- uint64_t ipackets
- uint64_t opackets
- uint64_t ibytes
- uint64_t obytes

- uint64_t ierrors
- uint64_t oerrors
- uint64_t imcasts
- uint64_t rx_nombuf
- uint64_t fdirmatch
- uint64_t fdirmiss
- uint64_t q_ipackets [RTE_ETHDEV_QUEUE_STAT_CNTRS]
- uint64_t q_opackets [RTE_ETHDEV_QUEUE_STAT_CNTRS]
- uint64_t q_ibytes [RTE_ETHDEV_QUEUE_STAT_CNTRS]
- uint64_t q_obytes [RTE_ETHDEV_QUEUE_STAT_CNTRS]
- uint64_t q_errors [RTE_ETHDEV_QUEUE_STAT_CNTRS]
- uint64_t ilbpackets
- uint64_t olbpackets
- uint64_t ilbbytes
- uint64_t olbbytes

2.31.1 Detailed Description

A structure used to retrieve statistics for an Ethernet port.

2.31.2 Field Documentation

2.31.2.1 uint64_t rte_eth_stats::ipackets

Total number of successfully received packets.

2.31.2.2 uint64_t rte_eth_stats::opackets

Total number of successfully transmitted packets.

2.31.2.3 uint64_t rte_eth_stats::ibytes

Total number of successfully received bytes.

2.31.2.4 uint64_t rte_eth_stats::obytes

Total number of successfully transmitted bytes.

2.31.2.5 uint64_t rte_eth_stats::ierrors

Total number of erroneous received packets.

2.31.2.6 uint64_t rte_eth_stats::oerrors

Total number of failed transmitted packets.

2.31.2.7 uint64_t rte_eth_stats::imcasts

Total number of multicast received packets.

2.31.2.8 uint64_t rte_eth_stats::rx_nombuf

Total number of RX mbuf allocation failures.

2.31.2.9 uint64_t rte_eth_stats::fdirmatch

Total number of RX packets matching a filter.

2.31.2.10 uint64_t rte_eth_stats::fdirmiss

Total number of RX packets not matching any filter.

2.31.2.11 uint64_t rte_eth_stats::q_ipackets[RTE_ETHDEV_QUEUE_STAT_CNTRS]

Total number of queue RX packets.

2.31.2.12 uint64_t rte_eth_stats::q_opackets[RTE_ETHDEV_QUEUE_STAT_CNTRS]

Total number of queue TX packets.

2.31.2.13 uint64_t rte_eth_stats::q_ibytes[RTE_ETHDEV_QUEUE_STAT_CNTRS]

Total number of successfully received queue bytes.

2.31.2.14 uint64_t rte_eth_stats::q_obytes[RTE_ETHDEV_QUEUE_STAT_CNTRS]

Total number of successfully transmitted queue bytes.

2.31.2.15 uint64_t rte_eth_stats::q_errors[RTE_ETHDEV_QUEUE_STAT_CNTRS]

Total number of queue packets received that are dropped.

2.31.2.16 uint64_t rte_eth_stats::ilbpackets

Total number of good packets received from loopback,VF Only

2.31.2.17 uint64_t rte_eth_stats::olbpackets

Total number of good packets transmitted to loopback,VF Only

2.31.2.18 uint64_t rte_eth_stats::ilbbytes

Total number of good bytes received from loopback,VF Only

2.31.2.19 uint64_t rte_eth_stats::olbbytes

Total number of good bytes transmitted to loopback,VF Only

2.32 rte_eth_thresh Struct Reference

Data Fields

- uint8_t pthresh
- uint8_t hthresh
- uint8_t wthresh

2.32.1 Detailed Description

A structure used to configure the ring threshold registers of an RX/TX queue for an Ethernet port.

2.32.2 Field Documentation

2.32.2.1 uint8_t rte_eth_thresh::pthresh

Ring prefetch threshold.

2.32.2.2 uint8_t rte_eth_thresh::hthresh

Ring host threshold.

2.32.2.3 uint8_t rte_eth_thresh::wthresh

Ring writeback threshold.

2.33 rte_eth_txconf Struct Reference

Data Fields

- struct rte_eth_thresh tx_thresh
- uint16_t tx_rs_thresh
- uint16_t tx_free_thresh
- uint32_t txq_flags

2.33.1 Detailed Description

A structure used to configure a TX ring of an Ethernet port.

2.33.2 Field Documentation

2.33.2.1 struct rte_eth_thresh rte_eth_txconf::tx_thresh

TX ring threshold registers.

2.33.2.2 uint16_t rte_eth_txconf::tx_rs_thresh

Drives the setting of RS bit on TXDs.

2.33.2.3 uint16_t rte_eth_txconf::tx_free_thresh

Drives the freeing of TX buffers.

2.33.2.4 uint32_t rte_eth_txconf::txq_flags

Set flags for the Tx queue

2.34 rte_eth_txmode Struct Reference

Data Fields

- enum rte_eth_tx_mq_mode mq_mode

2.34.1 Detailed Description

A structure used to configure the TX features of an Ethernet port.

2.34.2 Field Documentation

2.34.2.1 enum rte_eth_tx_mq_mode rte_eth_txmode::mq_mode

TX multi-queues mode.

2.35 rte_eth_vlan_mirror Struct Reference

Data Fields

- uint64_t vlan_mask

2.35.1 Detailed Description

A structure used to configure VLAN traffic mirror of an Ethernet port.

2.35.2 Field Documentation

2.35.2.1 uint64_t rte_eth_vlan_mirror::vlan_mask

mask for valid VLAN ID.

2.36 rte_eth_vmdq_dcb_conf Struct Reference

Data Fields

- enum rte_eth_nb_pools nb_queue_pools
- uint8_t enable_default_pool
- uint8_t default_pool
- uint8_t nb_pool_maps
- struct {
    uint16_t vlan_id
    uint64_t pools
  } pool_map [ETH_VMDQ_MAX_VLAN_FILTERS]

- uint8_t dcb_queue [ETH_DCB_NUM_USER_PRIORITIES]

2.36.1 Detailed Description

A structure used to configure the VMDQ+DCB feature of an Ethernet port.

Using this feature, packets are routed to a pool of queues, based on the vlan id in the vlan tag, and then to a specific queue within that pool, using the user priority vlan tag field.

A default pool may be used, if desired, to route all traffic which does not match the vlan filter rules.

2.36.2 Field Documentation

2.36.2.1 enum rte_eth_nb_pools rte_eth_vmdq_dcb_conf::nb_queue_pools

With DCB, 16 or 32 pools

2.36.2.2 uint8_t rte_eth_vmdq_dcb_conf::enable_default_pool

If non-zero, use a default pool

2.36.2.3 uint8_t rte_eth_vmdq_dcb_conf::default_pool

The default pool, if applicable

2.36.2.4 uint8_t rte_eth_vmdq_dcb_conf::nb_pool_maps

We can have up to 64 filters/mappings

2.36.2.5 uint16_t rte_eth_vmdq_dcb_conf::vlan_id

The vlan id of the received frame

2.36.2.6 uint64_t rte_eth_vmdq_dcb_conf::pools

Bitmask of pools for packet rx

2.36.2.7 struct { ... } rte_eth_vmdq_dcb_conf::pool_map[ETH_VMDQ_MAX_VLAN_FILTERS]

VMDq vlan pool maps.

2.36.2.8 uint8_t rte_eth_vmdq_dcb_conf::dcb_queue[ETH_DCB_NUM_USER_PRIORITIES]

Selects a queue in a pool

2.37 rte_eth_vmdq_dcb_tx_conf Struct Reference

Data Fields

- enum rte_eth_nb_pools nb_queue_pools
- uint8_t dcb_queue [ETH_DCB_NUM_USER_PRIORITIES]

2.37.1 Field Documentation

2.37.1.1 enum rte_eth_nb_pools rte_eth_vmdq_dcb_tx_conf::nb_queue_pools

With DCB, 16 or 32 pools.

2.37.1.2 uint8_t rte_eth_vmdq_dcb_tx_conf::dcb_queue[ETH_DCB_NUM_USER_PRIORITIES]

Possible DCB queue,4 or 8.

2.38 rte_eth_vmdq_mirror_conf Struct Reference

Data Fields

- uint8_t rule_type_mask
- uint8_t dst_pool
- uint64_t pool_mask
- struct rte_eth_vlan_mirror vlan

2.38.1 Detailed Description

A structure used to configure traffic mirror of an Ethernet port.

2.38.2 Field Documentation

2.38.2.1 uint8_t rte_eth_vmdq_mirror_conf::rule_type_mask

Mirroring rule type mask we want to set

2.38.2.2 uint8_t rte_eth_vmdq_mirror_conf::dst_pool

Destination pool for this mirror rule.

2.38.2.3 uint64_t rte_eth_vmdq_mirror_conf::pool_mask

Bitmap of pool for pool mirroring

2.38.2.4 struct rte_eth_vlan_mirror rte_eth_vmdq_mirror_conf::vlan

VLAN ID setting for VLAN mirroring

2.39 rte_eth_vmdq_rx_conf Struct Reference

Data Fields

- enum rte_eth_nb_pools nb_queue_pools
- uint8_t enable_default_pool
- uint8_t default_pool
- uint8_t nb_pool_maps
- struct {
  uint16_t vlan_id
  uint64_t pools
  } pool_map [ETH_VMDQ_MAX_VLAN_FILTERS]

2.39.1 Field Documentation

2.39.1.1 enum rte_eth_nb_pools rte_eth_vmdq_rx_conf::nb_queue_pools

VMDq only mode, 8 or 64 pools

2.39.1.2 uint8_t rte_eth_vmdq_rx_conf::enable_default_pool

If non-zero, use a default pool

2.39.1.3 uint8_t rte_eth_vmdq_rx_conf::default_pool

The default pool, if applicable

2.39.1.4 uint8_t rte_eth_vmdq_rx_conf::nb_pool_maps

We can have up to 64 filters/mappings

2.39.1.5 uint16_t rte_eth_vmdq_rx_conf::vlan_id

The vlan id of the received frame

2.39.1.6 uint64_t rte_eth_vmdq_rx_conf::pools

Bitmask of pools for packet rx

2.39.1.7 struct { ... } rte_eth_vmdq_rx_conf::pool_map[ETH_VMDQ_MAX_VLAN_FILTERS]

VMDq vlan pool maps.

2.40 rte_eth_vmdq_tx_conf Struct Reference

Data Fields

- enum rte_eth_nb_pools nb_queue_pools

2.40.1 Field Documentation

2.40.1.1 enum rte_eth_nb_pools rte_eth_vmdq_tx_conf::nb_queue_pools

VMDq mode, 64 pools.

2.41 rte_fbk_hash_entry Union Reference

Data Fields

- uint64_t whole_entry
- struct {
  uint16_t is_entry
  uint16_t value
  uint32_t key
  } entry

2.41.1 Detailed Description

Individual entry in the four-byte key hash table.

2.41.2 Field Documentation

2.41.2.1 uint64_t rte_fbk_hash_entry::whole_entry

For accessing entire entry.

2.41.2.2 uint16_t rte_fbk_hash_entry::is_entry

Non-zero if entry is active.

2.41.2.3 uint16_t rte_fbk_hash_entry::value

Value returned by lookup.

2.41.2.4 uint32_t rte_fbk_hash_entry::key

Key used to find value.

2.41.2.5 struct { ... } rte_fbk_hash_entry::entry

For accessing each entry part.

2.42 rte_fbk_hash_params Struct Reference

Data Fields

- const char * name
- uint32_t entries
- uint32_t entries_per_bucket
- int socket_id
- rte_fbk_hash_fn hash_func
- uint32_t init_val

2.42.1 Detailed Description

Parameters used when creating four-byte key hash table.

2.42.2 Field Documentation

2.42.2.1 const char* rte_fbk_hash_params::name

Name of the hash table.

2.42.2.2 uint32_t rte_fbk_hash_params::entries

Total number of entries.

2.42.2.3 uint32_t rte_fbk_hash_params::entries_per_bucket

Number of entries in a bucket.

2.42.2.4 int rte_fbk_hash_params::socket_id

Socket to allocate memory on.

2.42.2.5 rte_fbk_hash_fn rte_fbk_hash_params::hash_func

The hash function.

2.42.2.6 uint32_t rte_fbk_hash_params::init_val

For initialising hash function.

2.43 rte_fbk_hash_table Struct Reference

Data Fields

- TAILQ_ENTRY next
- char name [RTE_FBK_HASH_NAMESIZE]
- uint32_t entries
- uint32_t entries_per_bucket
- uint32_t used_entries
- uint32_t bucket_mask
- uint32_t bucket_shift
- rte_fbk_hash_fn hash_func
- uint32_t init_val
- union rte_fbk_hash_entry t [0]

2.43.1 Detailed Description

The four-byte key hash table structure.

2.43.2 Field Documentation

2.43.2.1 TAILQ_ENTRY rte_fbk_hash_table::next

Linked list.

2.43.2.2 char rte_fbk_hash_table::name[RTE_FBK_HASH_NAMESIZE]

Name of the hash.

2.43.2.3 uint32_t rte_fbk_hash_table::entries

Total number of entries.

2.43.2.4 uint32_t rte_fbk_hash_table::entries_per_bucket

Number of entries in a bucket.

2.43.2.5 uint32_t rte_fbk_hash_table::used_entries

How many entries are used.

2.43.2.6 uint32_t rte_fbk_hash_table::bucket_mask

To find which bucket the key is in.

2.43.2.7 uint32_t rte_fbk_hash_table::bucket_shift

Convert bucket to table offset.

2.43.2.8 rte_fbk_hash_fn rte_fbk_hash_table::hash_func

The hash function.

2.43.2.9 uint32_t rte_fbk_hash_table::init_val

For initialising hash function.

2.43.2.10 union rte_fbk_hash_entry rte_fbk_hash_table::t[0]

A flat table of all buckets.

2.44 rte_fdir_conf Struct Reference

Data Fields

- enum rte_fdir_mode mode
- enum rte_fdir_pballoc_type pballoc
- enum rte_fdir_status_mode status
- uint8_t flexbytes_offset
- uint8_t drop_queue

2.44.1 Detailed Description

A structure used to configure the Flow Director (FDIR) feature of an Ethernet port.

If mode is RTE_FDIR_DISABLE, the pballoc value is ignored.

2.44.2 Field Documentation

2.44.2.1 enum rte_fdir_mode rte_fdir_conf::mode

Flow Director mode.

2.44.2.2 enum rte_fdir_pballoc_type rte_fdir_conf::pballoc

Space for FDIR filters.

2.44.2.3 enum rte_fdir_status_mode rte_fdir_conf::status

How to report FDIR hash.

2.44.2.4 uint8_t rte_fdir_conf::flexbytes_offset

Offset of flexbytes field in RX packets (in 16-bit word units).

2.44.2.5 uint8_t rte_fdir_conf::drop_queue

RX queue of packets matching a "drop" filter in perfect mode.

2.45 rte_fdir_filter Struct Reference

Data Fields

- uint16_t flex_bytes
- uint16_t vlan_id
- uint16_t port_src
- uint16_t port_dst
- union {
    uint32_t ipv4_addr
    uint32_t ipv6_addr [4]
  } ip_src

- union {
    uint32_t ipv4_addr
    uint32_t ipv6_addr [4]
  } ip_dst

- enum rte_l4type l4type
- enum rte_iptype iptype

2.45.1 Detailed Description

A structure used to define a FDIR packet filter.

2.45.2 Field Documentation

2.45.2.1 uint16_t rte_fdir_filter::flex_bytes

Flex bytes value to match.

2.45.2.2 uint16_t rte_fdir_filter::vlan_id

VLAN ID value to match, 0 otherwise.

2.45.2.3 uint16_t rte_fdir_filter::port_src

Source port to match, 0 otherwise.

2.45.2.4 uint16_t rte_fdir_filter::port_dst

Destination port to match, 0 otherwise.

2.45.2.5 uint32_t rte_fdir_filter::ipv4_addr

IPv4 source address to match.

IPv4 destination address to match.

2.45.2.6 uint32_t rte_fdir_filter::ipv6_addr[4]

IPv6 source address to match.

IPv6 destination address to match

2.45.2.7 union { ... } rte_fdir_filter::ip_src

IPv4/IPv6 source address to match (union of above).

2.45.2.8 union { ... } rte_fdir_filter::ip_dst

IPv4/IPv6 destination address to match (union of above).

2.45.2.9 enum rte_l4type rte_fdir_filter::l4type

l4type to match: NONE/UDP/TCP/SCTP.

2.45.2.10 enum rte_iptype rte_fdir_filter::iptype

IP packet type to match: IPv4 or IPv6.

2.46 rte_fdir_masks Struct Reference

Data Fields

- uint8_t only_ip_flow
- uint8_t vlan_id
- uint8_t vlan_prio
- uint8_t flexbytes
- uint8_t set_ipv6_mask
- uint8_t comp_ipv6_dst

- uint32_t dst_ipv4_mask
- uint32_t src_ipv4_mask
- uint16_t dst_ipv6_mask
- uint16_t src_ipv6_mask
- uint16_t src_port_mask
- uint16_t dst_port_mask

2.46.1 Detailed Description

A structure used to configure FDIR masks that are used by the device to match the various fields of RX packet headers.

Note

The only_ip_flow field has the opposite meaning compared to other masks!

2.46.2 Field Documentation

2.46.2.1 uint8_t rte_fdir_masks::only_ip_flow

When set to 1, packet l4type is **NOT** relevant in filters, and source and destination port masks must be set to zero.

2.46.2.2 uint8_t rte_fdir_masks::vlan_id

If set to 1, vlan_id is relevant in filters.

2.46.2.3 uint8_t rte_fdir_masks::vlan_prio

If set to 1, vlan_prio is relevant in filters.

2.46.2.4 uint8_t rte_fdir_masks::flexbytes

If set to 1, flexbytes is relevant in filters.

2.46.2.5 uint8_t rte_fdir_masks::set_ipv6_mask

If set to 1, set the IPv6 masks. Otherwise set the IPv4 masks.

2.46.2.6 uint8_t rte_fdir_masks::comp_ipv6_dst

When set to 1, comparison of destination IPv6 address with IP6AT registers is meaningful.

2.46.2.7 uint32_t rte_fdir_masks::dst_ipv4_mask

Mask of Destination IPv4 Address. All bits set to 1 define the relevant bits to use in the destination address of an IPv4 packet when matching it against FDIR filters.

2.46.2.8 uint32_t rte_fdir_masks::src_ipv4_mask

Mask of Source IPv4 Address. All bits set to 1 define the relevant bits to use in the source address of an IPv4 packet when matching it against FDIR filters.

2.46.2.9 uint16_t rte_fdir_masks::dst_ipv6_mask

Mask of Source IPv6 Address. All bits set to 1 define the relevant BYTES to use in the source address of an IPv6 packet when matching it against FDIR filters.

2.46.2.10 uint16_t rte_fdir_masks::src_ipv6_mask

Mask of Destination IPv6 Address. All bits set to 1 define the relevant BYTES to use in the destination address of an IPv6 packet when matching it against FDIR filters.

2.46.2.11 uint16_t rte_fdir_masks::src_port_mask

Mask of Source Port. All bits set to 1 define the relevant bits to use in the source port of an IP packets when matching it against FDIR filters.

2.46.2.12 uint16_t rte_fdir_masks::dst_port_mask

Mask of Destination Port. All bits set to 1 define the relevant bits to use in the destination port of an IP packet when matching it against FDIR filters.

2.47 rte_hash Struct Reference

Data Fields

- TAILQ_ENTRY next
- char name [RTE_HASH_NAMESIZE]
- uint32_t entries
- uint32_t bucket_entries
- uint32_t key_len
- rte_hash_function hash_func
- uint32_t hash_func_init_val
- uint32_t num_buckets

- uint32_t bucket_bitmask
- hash_sig_t sig_msb
- uint8_t * sig_tbl
- uint32_t sig_tbl_bucket_size
- uint8_t * key_tbl
- uint32_t key_tbl_key_size

2.47.1 Detailed Description

A hash table structure.

2.47.2 Field Documentation

2.47.2.1 TAILQ_ENTRY rte_hash::next

Next in list.

2.47.2.2 char rte_hash::name[RTE_HASH_NAMESIZE]

Name of the hash.

2.47.2.3 uint32_t rte_hash::entries

Total table entries.

2.47.2.4 uint32_t rte_hash::bucket_entries

Bucket entries.

2.47.2.5 uint32_t rte_hash::key_len

Length of hash key.

2.47.2.6 rte_hash_function rte_hash::hash_func

Function used to calculate hash.

2.47.2.7 uint32_t rte_hash::hash_func_init_val

Init value used by hash_func.

2.47.2.8 uint32_t rte_hash::num_buckets

Number of buckets in table.

2.47.2.9 uint32_t rte_hash::bucket_bitmask

Bitmask for getting bucket index from hash signature.

2.47.2.10 hash_sig_t rte_hash::sig_msb

MSB is always set in valid signatures.

2.47.2.11 uint8_t* rte_hash::sig_tbl

Flat array of hash signature buckets.

2.47.2.12 uint32_t rte_hash::sig_tbl_bucket_size

Signature buckets may be padded for alignment reasons, and this is the bucket size used by sig_tbl.

2.47.2.13 uint8_t* rte_hash::key_tbl

Flat array of key value buckets.

2.47.2.14 uint32_t rte_hash::key_tbl_key_size

Keys may be padded for alignment reasons, and this is the key size used by key_tbl.

2.48 rte_hash_parameters Struct Reference

Data Fields

- const char * name
- uint32_t entries
- uint32_t bucket_entries
- uint32_t key_len
- rte_hash_function hash_func
- uint32_t hash_func_init_val
- int socket_id

2.48.1 Detailed Description

Parameters used when creating the hash table. The total table entries and bucket entries must be a power of 2.

2.48.2 Field Documentation

2.48.2.1 const char* rte_hash_parameters::name

Name of the hash.

2.48.2.2 uint32_t rte_hash_parameters::entries

Total hash table entries.

2.48.2.3 uint32_t rte_hash_parameters::bucket_entries

Bucket entries.

2.48.2.4 uint32_t rte_hash_parameters::key_len

Length of hash key.

2.48.2.5 rte_hash_function rte_hash_parameters::hash_func

Function used to calculate hash.

2.48.2.6 uint32_t rte_hash_parameters::hash_func_init_val

Init value used by hash_func.

2.48.2.7 int rte_hash_parameters::socket_id

NUMA Socket ID for memory.

2.49 rte_intr_conf Struct Reference

Data Fields

- uint16_t lsc

2.49.1 Detailed Description

A structure used to enable/disable specific device interrupts.

2.49.2 Field Documentation

2.49.2.1 uint16_t rte_intr_conf::lsc

enable/disable lsc interrupt. 0 (default) - disable, 1 enable

2.50 rte_ivshmem_metadata Struct Reference

Data Fields

- int magic_number
- char name [IVSHMEM_NAME_LEN]
- struct rte_ivshmem_metadata_entry entry [RTE_LIBRTE_IVSHMEM_MAX_ENTRIES]

2.50.1 Detailed Description

Structure that holds IVSHMEM metadata.

2.50.2 Field Documentation

2.50.2.1 int rte_ivshmem_metadata::magic_number

magic number

2.50.2.2 char rte_ivshmem_metadata::name[IVSHMEM_NAME_LEN]

name of the metadata file

2.50.2.3 struct rte_ivshmem_metadata_entry rte_ivshmem_metadata::entry[RTE_LIBRTE_IVSHMEM_MAX_ENTRIES]

metadata entries

2.51 rte_ivshmem_metadata_entry Struct Reference

Data Fields

- struct rte_memzone mz
- uint64_t offset

2.51.1 Detailed Description

Structure that holds IVSHMEM shared metadata entry.

2.51.2 Field Documentation

2.51.2.1 struct rte_memzone rte_ivshmem_metadata_entry::mz

shared memzone

2.51.2.2 uint64_t rte_ivshmem_metadata_entry::offset

offset of memzone within IVSHMEM device

2.52 rte_kni_conf Struct Reference

2.52.1 Detailed Description

Structure for configuring KNI device.

2.53 rte_kni_ops Struct Reference

2.53.1 Detailed Description

Structure which has the function pointers for KNI interface.

2.54 rte_logs Struct Reference

Data Fields

- uint32_t type

- uint32_t level
- FILE * file

2.54.1 Detailed Description

The rte_log structure.

2.54.2 Field Documentation

2.54.2.1 uint32_t rte_logs::type

Bitfield with enabled logs.

2.54.2.2 uint32_t rte_logs::level

Log level.

2.54.2.3 FILE* rte_logs::file

Pointer to current FILE* for logs.

2.55 rte_lpm Struct Reference

Data Fields

- TAILQ_ENTRY next
- char name [RTE_LPM_NAMESIZE]
- int mem_location
- uint32_t max_rules
- struct rte_lpm_rule_info rule_info [RTE_LPM_MAX_DEPTH]
- struct rte_lpm_tbl24_entry struct rte_lpm_tbl8_entry tbl8 [RTE_LPM_TBL8_NUM_ENTRIES]
- struct rte_lpm_tbl24_entry struct rte_lpm_tbl8_entry struct rte_lpm_rule rules_tbl [0]

2.55.1 Field Documentation

2.55.1.1 TAILQ_ENTRY rte_lpm::next

Next in list.

2.55.1.2 char rte_lpm::name[RTE_LPM_NAMESIZE]

Name of the lpm.

2.55.1.3 int rte_lpm::mem_location

Deprecated

See also

RTE_LPM_HEAP and RTE_LPM_MEMZONE.

2.55.1.4 uint32_t rte_lpm::max_rules

Max. balanced rules per lpm.

2.55.1.5 struct rte_lpm_rule_info rte_lpm::rule_info[RTE_LPM_MAX_DEPTH]

Rule info table.

2.55.1.6 struct rte_lpm_tbl24_entry struct rte_lpm_tbl8_entry rte_lpm::tbl8[RTE_LPM_TBL8_NUM_ENTRIES]

LPM tbl24 table.

2.55.1.7 struct rte_lpm_tbl24_entry struct rte_lpm_tbl8_entry struct rte_lpm_rule rte_lpm::rules_tbl[0]

LPM tbl8 table.

2.56 rte_lpm6_config Struct Reference

Data Fields

- uint32_t max_rules
- uint32_t number_tbl8s
- int flags

2.56.1 Detailed Description

LPM configuration structure.

2.56.2 Field Documentation

2.56.2.1 uint32_t rte_lpm6_config::max_rules

Max number of rules.

2.56.2.2 uint32_t rte_lpm6_config::number_tbl8s

Number of tbl8s to allocate.

2.56.2.3 int rte_lpm6_config::flags

This field is currently unused.

2.57 rte_lpm_rule Struct Reference

Data Fields

- uint32_t ip
- uint8_t next_hop

2.57.1 Field Documentation

2.57.1.1 uint32_t rte_lpm_rule::ip

Rule IP address.

2.57.1.2 uint8_t rte_lpm_rule::next_hop

Rule next hop.

2.58 rte_lpm_rule_info Struct Reference

Data Fields

- uint32_t used_rules
- uint32_t first_rule

2.58.1 Field Documentation

2.58.1.1 uint32_t rte_lpm_rule_info::used_rules

Used rules so far.

2.58.1.2 uint32_t rte_lpm_rule_info::first_rule

Indexes the first rule of a given depth.

2.59 rte_lpm_tbl24_entry Struct Reference

Data Fields

- uint8_t valid:1
- uint8_t ext_entry:1
- uint8_t depth:6

2.59.1 Field Documentation

2.59.1.1 uint8_t rte_lpm_tbl24_entry::valid

Validation flag.

2.59.1.2 uint8_t rte_lpm_tbl24_entry::ext_entry

External entry.

2.59.1.3 uint8_t rte_lpm_tbl24_entry::depth

Rule depth.

2.60 rte_lpm_tbl8_entry Struct Reference

Data Fields

- uint8_t next_hop
- uint8_t valid:1
- uint8_t valid_group:1
- uint8_t depth:6

2.60.1 Field Documentation

2.60.1.1 uint8_t rte_lpm_tbl8_entry::next_hop

next hop.

2.60.1.2 uint8_t rte_lpm_tbl8_entry::valid

Validation flag.

2.60.1.3 uint8_t rte_lpm_tbl8_entry::valid_group

Group validation flag.

2.60.1.4 uint8_t rte_lpm_tbl8_entry::depth

Rule depth.

2.61 rte_malloc_socket_stats Struct Reference

Data Fields

- size_t heap_totalsz_bytes
- size_t heap_freesz_bytes
- size_t greatest_free_size
- unsigned free_count
- unsigned alloc_count
- size_t heap_allocsz_bytes

2.61.1 Detailed Description

Structure to hold heap statistics obtained from rte_malloc_get_socket_stats function.

2.61.2 Field Documentation

2.61.2.1 size_t rte_malloc_socket_stats::heap_totalsz_bytes

Total bytes on heap

2.61.2.2 size_t rte_malloc_socket_stats::heap_freesz_bytes

Total free bytes on heap

2.61.2.3 size_t rte_malloc_socket_stats::greatest_free_size

Size in bytes of largest free block

2.61.2.4 unsigned rte_malloc_socket_stats::free_count

Number of free elements on heap

2.61.2.5 unsigned rte_malloc_socket_stats::alloc_count

Number of allocated elements on heap

2.61.2.6 size_t rte_malloc_socket_stats::heap_allocsz_bytes

Total allocated bytes on heap

2.62 rte_mbuf Struct Reference

Data Fields

- struct rte_mempool * pool
- void * buf_addr
- phys_addr_t buf_physaddr
- uint16_t buf_len
- union {
  rte_atomic16_t refcnt_atomic
  uint16_t refcnt
  };
- uint8_t type
- uint8_t reserved
- uint16_t ol_flags

2.62.1 Detailed Description

The generic rte_mbuf, containing a packet mbuf or a control mbuf.

2.62.2 Field Documentation

2.62.2.1 struct rte_mempool* rte_mbuf::pool

Pool from which mbuf was allocated.

2.62.2.2 void* rte_mbuf::buf_addr

Virtual address of segment buffer.

2.62.2.3 phys_addr_t rte_mbuf::buf_physaddr

Physical address of segment buffer.

2.62.2.4 uint16_t rte_mbuf::buf_len

Length of segment buffer.

2.62.2.5 rte_atomic16_t rte_mbuf::refcnt_atomic

Atomically accessed refcnt

2.62.2.6 uint16_t rte_mbuf::refcnt

Non-atomically accessed refcnt

2.62.2.7 union { ... }

16-bit Reference counter. It should only be accessed using the following functions: rte_mbuf_refcnt_update(), rte_mbuf_refcnt_read(), and rte_mbuf_refcnt_set(). The functionality of these functions (atomic, or non-atomic) is controlled by the CONFIG_RTE_MBUF_REFCNT_ATOMIC config option.

2.62.2.8 uint8_t rte_mbuf::type

Type of mbuf.

2.62.2.9 uint8_t rte_mbuf::reserved

Unused field. Required for padding.

2.62.2.10 uint16_t rte_mbuf::ol_flags

Offload features.

2.63 rte_mem_config Struct Reference

Data Fields

- volatile uint32_t magic
- uint32_t nchannel
- uint32_t nrank
- rte_rwlock_t mlock
- rte_rwlock_t qlock
- rte_rwlock_t mplock
- uint32_t memzone_idx
- struct rte_memseg memseg [32,]
- struct rte_memzone memzone [512,]
- struct rte_tailq_head tailq_head [32,]

2.63.1 Detailed Description

the structure for the memory configuration for the RTE. Used by the rte_config structure. It is separated out, as for multi-process support, the memory details should be shared across instances

2.63.2 Field Documentation

2.63.2.1 volatile uint32_t rte_mem_config::magic

Magic number - Sanity check.

2.63.2.2 uint32_t rte_mem_config::nchannel

Number of channels (0 if unknown).

2.63.2.3 uint32_t rte_mem_config::nrank

Number of ranks (0 if unknown).

2.63.2.4 rte_rwlock_t rte_mem_config::mlock

current lock nest order

- qlock->mlock (ring/hash/lpm)
- mplock->qlock->mlock (mempool) Notice: *ALWAYS* obtain qlock first if having to obtain both qlock and mlock only used by memzone LIB for thread-safe.

2.63.2.5 rte_rwlock_t rte_mem_config::qlock

used for tailq operation for thread safe.

2.63.2.6 rte_rwlock_t rte_mem_config::mplock

only used by mempool LIB for thread-safe.

2.63.2.7 uint32_t rte_mem_config::memzone_idx

Index of memzone

2.63.2.8 struct rte_memseg rte_mem_config::memseg[32,]

Physmem descriptors.

2.63.2.9 struct rte_memzone rte_mem_config::memzone[512,]

Memzone descriptors.

2.63.2.10 struct rte_tailq_head rte_mem_config::tailq_head[32,]

Tailqs for objects

2.64 rte_mempool Struct Reference

Data Fields

- TAILQ_ENTRY next
- char name [RTE_MEMPOOL_NAMESIZE]
- struct rte_ring * ring

- phys_addr_t phys_addr
- int flags
- uint32_t size
- uint32_t cache_size
- uint32_t cache_flushthresh
- uint32_t elt_size
- uint32_t header_size
- uint32_t trailer_size
- unsigned private_data_size
- struct rte_mempool_cache local_cache [32,]
- uint32_t pg_num
- uint32_t uint32_t pg_shift
- uintptr_t pg_mask
- uintptr_t elt_va_start
- uintptr_t elt_va_end
- phys_addr_t elt_pa [MEMPOOL_PG_NUM_DEFAULT]

2.64.1 Detailed Description

The RTE mempool structure.

2.64.2 Field Documentation

2.64.2.1 TAILQ_ENTRY rte_mempool::next

Next in list.

2.64.2.2 char rte_mempool::name[RTE_MEMPOOL_NAMESIZE]

Name of mempool.

2.64.2.3 struct rte_ring* rte_mempool::ring

Ring to store objects.

2.64.2.4 phys_addr_t rte_mempool::phys_addr

Phys. addr. of mempool struct.

2.64.2.5 int rte_mempool::flags

Flags of the mempool.

2.64.2.6 uint32_t rte_mempool::size

Size of the mempool.

2.64.2.7 uint32_t rte_mempool::cache_size

Size of per-lcore local cache.

2.64.2.8 uint32_t rte_mempool::cache_flushthresh

Threshold before we flush excess elements.

2.64.2.9 uint32_t rte_mempool::elt_size

Size of an element.

2.64.2.10 uint32_t rte_mempool::header_size

Size of header (before elt).

2.64.2.11 uint32_t rte_mempool::trailer_size

Size of trailer (after elt).

2.64.2.12 unsigned rte_mempool::private_data_size

Size of private data.

2.64.2.13 struct rte_mempool_cache rte_mempool::local_cache[32,]

Per-lcore local cache.

2.64.2.14 uint32_t rte_mempool::pg_num

Number of elements in the elt_pa array.

2.64.2.15 uint32_t uint32_t rte_mempool::pg_shift

LOG2 of the physical pages.

2.64.2.16 uintptr_t rte_mempool::pg_mask

physical page mask value.

2.64.2.17 uintptr_t rte_mempool::elt_va_start

Virtual address of the first mempool object.

2.64.2.18 uintptr_t rte_mempool::elt_va_end

Virtual address of the <size + 1> mempool object.

2.64.2.19 phys_addr_t rte_mempool::elt_pa[MEMPOOL_PG_NUM_DEFAULT]

Array of physical pages addresses for the mempool objects buffer.

2.65 rte_mempool_cache Struct Reference

Data Fields

- unsigned len
- void * objs [512,*3]

2.65.1 Detailed Description

A structure that stores a per-core object cache.

2.65.2 Field Documentation

2.65.2.1 unsigned rte_mempool_cache::len

Cache len

2.65.2.2 void* rte_mempool_cache::objs[512,*3]

Cache objects

2.66 rte_mempool_objsz Struct Reference

Data Fields

- uint32_t elt_size
- uint32_t header_size
- uint32_t trailer_size
- uint32_t total_size

2.66.1 Field Documentation

2.66.1.1 uint32_t rte_mempool_objsz::elt_size

Size of an element.

2.66.1.2 uint32_t rte_mempool_objsz::header_size

Size of header (before elt).

2.66.1.3 uint32_t rte_mempool_objsz::trailer_size

Size of trailer (after elt).

2.66.1.4 uint32_t rte_mempool_objsz::total_size

Total size of an object (header + elt + trailer).

2.67 rte_memseg Struct Reference

Data Fields

- phys_addr_t phys_addr
- size_t len
- size_t hugepage_sz
- int32_t socket_id
- uint32_t nchannel
- uint32_t nrank
- void * addr
- uint64_t addr_64

2.67.1 Detailed Description

Physical memory segment descriptor.

2.67.2 Field Documentation

2.67.2.1 phys_addr_t rte_memseg::phys_addr

Start physical address.

2.67.2.2 void* rte_memseg::addr

Start virtual address.

2.67.2.3 uint64_t rte_memseg::addr_64

Makes sure addr is always 64 bits

2.67.2.4 size_t rte_memseg::len

Length of the segment.

2.67.2.5 size_t rte_memseg::hugepage_sz

The pagesize of underlying memory

2.67.2.6 int32_t rte_memseg::socket_id

NUMA socket ID.

2.67.2.7 uint32_t rte_memseg::nchannel

Number of channels.

2.67.2.8 uint32_t rte_memseg::nrank

Number of ranks.

2.68 rte_memzone Struct Reference

Data Fields

- char name [RTE_MEMZONE_NAMESIZE]
- phys_addr_t phys_addr
- size_t len
- size_t hugepage_sz
- int32_t socket_id
- uint32_t flags
- void * addr
- uint64_t addr_64

2.68.1 Detailed Description

A structure describing a memzone, which is a contiguous portion of physical memory identified by a name.

2.68.2 Field Documentation

2.68.2.1 char rte_memzone::name[RTE_MEMZONE_NAMESIZE]

Name of the memory zone.

2.68.2.2 phys_addr_t rte_memzone::phys_addr

Start physical address.

2.68.2.3 void* rte_memzone::addr

Start virtual address.

2.68.2.4 uint64_t rte_memzone::addr_64

Makes sure addr is always 64-bits

2.68.2.5 size_t rte_memzone::len

Length of the memzone.

2.68.2.6 size_t rte_memzone::hugepage_sz

The page size of underlying memory

2.68.2.7 int32_t rte_memzone::socket_id

NUMA socket ID.

2.68.2.8 uint32_t rte_memzone::flags

Characteristics of this memzone.

2.69 rte_meter_srtcm Struct Reference

2.70 rte_meter_srtcm_params Struct Reference

Data Fields

- uint64_t cir
- uint64_t cbs
- uint64_t ebs

2.70.1 Detailed Description

srTCM parameters per metered traffic flow. The CIR, CBS and EBS parameters only count bytes of IP packets and do not include link specific headers. At least one of the CBS or EBS parameters has to be greater than zero.

2.70.2 Field Documentation

2.70.2.1 uint64_t rte_meter_srtcm_params::cir

Committed Information Rate (CIR). Measured in bytes per second.

2.70.2.2 uint64_t rte_meter_srtcm_params::cbs

Committed Burst Size (CBS). Measured in bytes.

2.70.2.3 uint64_t rte_meter_srtcm_params::ebs

Excess Burst Size (EBS). Measured in bytes.

2.71 rte_meter_trtcm Struct Reference

2.72 rte_meter_trtcm_params Struct Reference

Data Fields

- uint64_t cir
- uint64_t pir
- uint64_t cbs
- uint64_t pbs

2.72.1 Detailed Description

trTCM parameters per metered traffic flow. The CIR, PIR, CBS and PBS parameters only count bytes of IP packets and do not include link specific headers. PIR has to be greater than or equal to CIR. Both CBS or EBS have to be greater than zero.

2.72.2 Field Documentation

2.72.2.1 uint64_t rte_meter_trtcm_params::cir

Committed Information Rate (CIR). Measured in bytes per second.

2.72.2.2 uint64_t rte_meter_trtcm_params::pir

Peak Information Rate (PIR). Measured in bytes per second.

2.72.2.3 uint64_t rte_meter_trtcm_params::cbs

Committed Burst Size (CBS). Measured in byes.

2.72.2.4 uint64_t rte_meter_trtcm_params::pbs

Peak Burst Size (PBS). Measured in bytes.

2.73 rte_pci_addr Struct Reference

Data Fields

- uint16_t domain
- uint8_t bus
- uint8_t devid
- uint8_t function

2.73.1 Detailed Description

A structure describing the location of a PCI device.

2.73.2 Field Documentation

2.73.2.1 uint16_t rte_pci_addr::domain

Device domain

2.73.2.2 uint8_t rte_pci_addr::bus

Device bus

2.73.2.3 uint8_t rte_pci_addr::devid

Device ID

2.73.2.4 uint8_t rte_pci_addr::function

Device function.

2.74 rte_pci_device Struct Reference

Data Fields

- TAILQ_ENTRY next
- struct rte_pci_addr addr
- struct rte_pci_id id
- struct rte_pci_resource mem_resource [PCI_MAX_RESOURCE]
- struct rte_intr_handle intr_handle

- struct rte_pci_driver * driver
- uint16_t max_vfs
- int numa_node
- unsigned int blacklisted:1

2.74.1 Detailed Description

A structure describing a PCI device.

2.74.2 Field Documentation

2.74.2.1 TAILQ_ENTRY rte_pci_device::next

Next probed PCI device.

2.74.2.2 struct rte_pci_addr rte_pci_device::addr

PCI location.

2.74.2.3 struct rte_pci_id rte_pci_device::id

PCI ID.

2.74.2.4 struct rte_pci_resource rte_pci_device::mem_resource[PCI_MAX_RESOURCE]

PCI Memory Resource

2.74.2.5 struct rte_intr_handle rte_pci_device::intr_handle

Interrupt handle

2.74.2.6 struct rte_pci_driver* rte_pci_device::driver

Associated driver

2.74.2.7 uint16_t rte_pci_device::max_vfs

sriov enable if not zero

2.74.2.8 int rte_pci_device::numa_node

NUMA node connection

2.74.2.9 unsigned int rte_pci_device::blacklisted

Device is blacklisted

2.75 rte_pci_driver Struct Reference

Data Fields

- TAILQ_ENTRY next
- const char * name
- pci_devinit_t * devinit
- struct rte_pci_id * id_table
- uint32_t drv_flags

2.75.1 Detailed Description

A structure describing a PCI driver.

2.75.2 Field Documentation

2.75.2.1 TAILQ_ENTRY rte_pci_driver::next

Next in list.

2.75.2.2 const char* rte_pci_driver::name

Driver name.

2.75.2.3 pci_devinit_t* rte_pci_driver::devinit

Device init. function.

2.75.2.4 struct rte_pci_id* rte_pci_driver::id_table

ID table, NULL terminated.

2.75.2.5 uint32_t rte_pci_driver::drv_flags

Flags contolling handling of device.

2.76 rte_pci_id Struct Reference

Data Fields

- uint16_t vendor_id
- uint16_t device_id
- uint16_t subsystem_vendor_id
- uint16_t subsystem_device_id

2.76.1 Detailed Description

A structure describing an ID for a PCI driver. Each driver provides a table of these IDs for each device that it supports.

2.76.2 Field Documentation

2.76.2.1 uint16_t rte_pci_id::vendor_id

Vendor ID or PCI_ANY_ID.

2.76.2.2 uint16_t rte_pci_id::device_id

Device ID or PCI_ANY_ID.

2.76.2.3 uint16_t rte_pci_id::subsystem_vendor_id

Subsystem vendor ID or PCI_ANY_ID.

2.76.2.4 uint16_t rte_pci_id::subsystem_device_id

Subsystem device ID or PCI_ANY_ID.

2.77 rte_pci_resource Struct Reference

Data Fields

- uint64_t phys_addr
- uint64_t len
- void ∗ addr

2.77.1 Detailed Description

A structure describing a PCI resource.

2.77.2 Field Documentation

2.77.2.1 uint64_t rte_pci_resource::phys_addr

Physical address, 0 if no resource.

2.77.2.2 uint64_t rte_pci_resource::len

Length of the resource.

2.77.2.3 void∗ rte_pci_resource::addr

Virtual address, NULL when not mapped.

2.78 rte_pktmbuf Struct Reference

Data Fields

- struct rte_mbuf ∗ next
- void ∗ data
- uint16_t data_len
- uint8_t nb_segs
- uint8_t in_port
- uint32_t pkt_len
- union {
  uint32_t rss
  struct {
  } fdir
  uint32_t sched
  } hash

2.78.1 Detailed Description

A packet message buffer.

2.78.2 Field Documentation

2.78.2.1 struct rte_mbuf* rte_pktmbuf::next

Next segment of scattered packet.

2.78.2.2 void* rte_pktmbuf::data

Start address of data in segment buffer.

2.78.2.3 uint16_t rte_pktmbuf::data_len

Amount of data in segment buffer.

2.78.2.4 uint8_t rte_pktmbuf::nb_segs

Number of segments.

2.78.2.5 uint8_t rte_pktmbuf::in_port

Input port.

2.78.2.6 uint32_t rte_pktmbuf::pkt_len

Total pkt len: sum of all segment data_len.

2.78.2.7 uint32_t rte_pktmbuf::rss

RSS hash result if RSS enabled

2.78.2.8 struct { ... } rte_pktmbuf::fdir

Filter identifier if FDIR enabled

2.78.2.9 uint32_t rte_pktmbuf::sched

Hierarchical scheduler

2.78.2.10 union { ... } rte_pktmbuf::hash

hash information

2.79 rte_pktmbuf_pool_private Struct Reference

Data Fields

- uint16_t mbuf_data_room_size

2.79.1 Detailed Description

Private data in case of pktmbuf pool.

A structure that contains some pktmbuf_pool-specific data that are appended after the mempool structure (in private data).

2.79.2 Field Documentation

2.79.2.1 uint16_t rte_pktmbuf_pool_private::mbuf_data_room_size

Size of data space in each mbuf.

2.80 rte_red Struct Reference

Data Fields

- uint32_t avg
- uint32_t count
- uint64_t q_time

2.80.1 Detailed Description

RED run-time data

2.80.2 Field Documentation

2.80.2.1 uint32_t rte_red::avg

Average queue size (avg), scaled in fixed-point format

2.80.2.2 uint32_t rte_red::count

Number of packets since last marked packet (count)

2.80.2.3 uint64_t rte_red::q_time

Start of the queue idle time (q_time)

2.81 rte_red_config Struct Reference

Data Fields

- uint32_t min_th
- uint32_t max_th
- uint32_t pa_const
- uint8_t maxp_inv
- uint8_t wq_log2

2.81.1 Detailed Description

RED configuration parameters

2.81.2 Field Documentation

2.81.2.1 uint32_t rte_red_config::min_th

min_th scaled in fixed-point format

2.81.2.2 uint32_t rte_red_config::max_th

max_th scaled in fixed-point format

2.81.2.3 uint32_t rte_red_config::pa_const

Precomputed constant value used for pa calculation (scaled in fixed-point format)

2.81.2.4 uint8_t rte_red_config::maxp_inv

maxp_inv

2.81.2.5 uint8_t rte_red_config::wq_log2

wq_log2

2.82 rte_red_params Struct Reference

Data Fields

- uint16_t min_th
- uint16_t max_th
- uint16_t maxp_inv
- uint16_t wq_log2

2.82.1 Detailed Description

RED configuration parameters passed by user

2.82.2 Field Documentation

2.82.2.1 uint16_t rte_red_params::min_th

Minimum threshold for queue (max_th)

2.82.2.2 uint16_t rte_red_params::max_th

Maximum threshold for queue (max_th)

2.82.2.3 uint16_t rte_red_params::maxp_inv

Inverse of packet marking probability maximum value (maxp = 1 / maxp_inv)

2.82.2.4 uint16_t rte_red_params::wq_log2

Negated log2 of queue weight (wq = 1 / (2 ^ wq_log2))

2.83 rte_ring Struct Reference

Data Structures

- struct cons
- struct prod

Data Fields

- TAILQ_ENTRY next
- char name [RTE_RING_NAMESIZE]
- int flags

2.83.1 Detailed Description

An RTE ring structure.

The producer and the consumer have a head and a tail index. The particularity of these index is that they are not between 0 and size(ring). These indexes are between 0 and 2^32, and we mask their value when we access the ring[] field. Thanks to this assumption, we can do subtractions between 2 index values in a modulo-32bit base: that's why the overflow of the indexes is not a problem.

2.83.2 Field Documentation

2.83.2.1 TAILQ_ENTRY rte_ring::next

Next in list.

2.83.2.2 char rte_ring::name[RTE_RING_NAMESIZE]

Name of the ring.

2.83.2.3 int rte_ring::flags

Flags supplied at creation.

2.84 rte_rwlock_t Struct Reference

Data Fields

- volatile int32_t cnt

2.84.1 Detailed Description

The rte_rwlock_t type.

cnt is -1 when write lock is held, and > 0 when read locks are held.

2.84.2 Field Documentation

2.84.2.1 volatile int32_t rte_rwlock_t::cnt

-1 when W lock held, > 0 when R locks held.

2.85 rte_sched_pipe_params Struct Reference

Data Fields

- uint32_t tb_rate
- uint32_t tb_size
- uint32_t tc_rate [RTE_SCHED_TRAFFIC_CLASSES_PER_PIPE]
- uint32_t tc_period
- uint8_t wrr_weights [RTE_SCHED_QUEUES_PER_PIPE]

2.85.1 Detailed Description

Pipe configuration parameters. The period and credits_per_period parameters are measured in bytes, with one byte meaning the time duration associated with the transmission of one byte on the physical medium of the output port, with pipe or pipe traffic class rate (measured as percentage of output port rate) determined as credits_per_period divided by period. One credit represents one byte.

2.85.2 Field Documentation

2.85.2.1 uint32_t rte_sched_pipe_params::tb_rate

Pipe token bucket rate (measured in bytes per second)

2.85.2.2 uint32_t rte_sched_pipe_params::tb_size

Pipe token bucket size (measured in credits)

2.85.2.3 uint32_t rte_sched_pipe_params::tc_rate[RTE_SCHED_TRAFFIC_CLASSES_PER_PIPE]

Pipe traffic class rates (measured in bytes per second)

2.85.2.4 uint32_t rte_sched_pipe_params::tc_period

Enforcement period for pipe traffic class rates (measured in milliseconds)

2.85.2.5 uint8_t rte_sched_pipe_params::wrr_weights[RTE_SCHED_QUEUES_PER_PIPE]

WRR weights for the queues of the current pipe

2.86 rte_sched_port_hierarchy Struct Reference

Data Fields

- uint32_t queue:2
- uint32_t traffic_class:2
- uint32_t pipe:20
- uint32_t subport:6
- uint32_t color:2

2.86.1 Detailed Description

Path through the scheduler hierarchy used by the scheduler enqueue operation to identify the destination queue for the current packet. Stored in the field pkt.hash.sched of struct rte_mbuf of each packet, typically written by the classification stage and read by scheduler enqueue.

2.86.2 Field Documentation

2.86.2.1 uint32_t rte_sched_port_hierarchy::queue

Queue ID (0 .. 3)

2.86.2.2 uint32_t rte_sched_port_hierarchy::traffic_class

Traffic class ID (0 .. 3)

2.86.2.3 uint32_t rte_sched_port_hierarchy::pipe

Pipe ID

2.86.2.4 uint32_t rte_sched_port_hierarchy::subport

Subport ID

2.86.2.5 uint32_t rte_sched_port_hierarchy::color

Color

2.87 rte_sched_port_params Struct Reference

Data Fields

- const char * name
- int socket
- uint32_t rate
- uint32_t mtu
- uint32_t frame_overhead
- uint32_t n_subports_per_port
- uint32_t n_pipes_per_subport
- uint16_t qsize [RTE_SCHED_TRAFFIC_CLASSES_PER_PIPE]
- struct rte_sched_pipe_params * pipe_profiles
- uint32_t n_pipe_profiles

2.87.1 Detailed Description

Port configuration parameters.

2.87.2 Field Documentation

2.87.2.1 const char* rte_sched_port_params::name

Literal string to be associated to the current port scheduler instance

2.87.2.2 int rte_sched_port_params::socket

CPU socket ID where the memory for port scheduler should be allocated

2.87.2.3 uint32_t rte_sched_port_params::rate

Output port rate (measured in bytes per second)

2.87.2.4 uint32_t rte_sched_port_params::mtu

Maximum Ethernet frame size (measured in bytes). Should not include the framing overhead.

2.87.2.5 uint32_t rte_sched_port_params::frame_overhead

Framing overhead per packet (measured in bytes)

2.87.2.6 uint32_t rte_sched_port_params::n_subports_per_port

Number of subports for the current port scheduler instance

2.87.2.7 uint32_t rte_sched_port_params::n_pipes_per_subport

Number of pipes for each port scheduler subport

2.87.2.8 uint16_t rte_sched_port_params::qsize[RTE_SCHED_TRAFFIC_CLASSES_PER_PIPE]

Packet queue size for each traffic class. All queues within the same pipe traffic class have the same size. Queues from different pipes serving the same traffic class have the same size.

2.87.2.9 struct rte_sched_pipe_params* rte_sched_port_params::pipe_profiles

Pipe profile table defined for current port scheduler instance. Every pipe of the current port scheduler is configured using one of the profiles from this table.

2.87.2.10 uint32_t rte_sched_port_params::n_pipe_profiles

Number of profiles in the pipe profile table

2.88 rte_sched_queue_stats Struct Reference

Data Fields

- uint32_t n_pkts
- uint32_t n_pkts_dropped
- uint32_t n_bytes
- uint32_t n_bytes_dropped

2.88.1 Detailed Description

Queue statistics

2.88.2 Field Documentation

2.88.2.1 uint32_t rte_sched_queue_stats::n_pkts

Number of packets successfully written to current queue

2.88.2.2 uint32_t rte_sched_queue_stats::n_pkts_dropped

Number of packets dropped due to current queue being full or congested

2.88.2.3 uint32_t rte_sched_queue_stats::n_bytes

Number of bytes successfully written to current queue

2.88.2.4 uint32_t rte_sched_queue_stats::n_bytes_dropped

Number of bytes dropped due to current queue being full or congested

2.89 rte_sched_subport_params Struct Reference

Data Fields

- uint32_t tb_rate
- uint32_t tb_size
- uint32_t tc_rate [RTE_SCHED_TRAFFIC_CLASSES_PER_PIPE]
- uint32_t tc_period

2.89.1 Detailed Description

Subport configuration parameters. The period and credits_per_period parameters are measured in bytes, with one byte meaning the time duration associated with the transmission of one byte on the physical medium of the output port, with pipe or pipe traffic class rate (measured as percentage of output port rate) determined as credits_per_period divided by period. One credit represents one byte.

2.89.2 Field Documentation

2.89.2.1 uint32_t rte_sched_subport_params::tb_rate

Subport token bucket rate (measured in bytes per second)

2.89.2.2 uint32_t rte_sched_subport_params::tb_size

Subport token bucket size (measured in credits)

2.89.2.3 uint32_t rte_sched_subport_params::tc_rate[RTE_SCHED_TRAFFIC_CLASSES_PER_PIPE]

Subport traffic class rates (measured in bytes per second)

2.89.2.4 uint32_t rte_sched_subport_params::tc_period

Enforcement period for traffic class rates (measured in milliseconds)

2.90 rte_sched_subport_stats Struct Reference

Data Fields

- uint32_t n_pkts_tc [RTE_SCHED_TRAFFIC_CLASSES_PER_PIPE]
- uint32_t n_pkts_tc_dropped [RTE_SCHED_TRAFFIC_CLASSES_PER_PIPE]
- uint32_t n_bytes_tc [RTE_SCHED_TRAFFIC_CLASSES_PER_PIPE]
- uint32_t n_bytes_tc_dropped [RTE_SCHED_TRAFFIC_CLASSES_PER_PIPE]

2.90.1 Detailed Description

Subport statistics

2.90.2 Field Documentation

2.90.2.1 uint32_t rte_sched_subport_stats::n_pkts_tc[RTE_SCHED_TRAFFIC_CLASSES_PER_PIPE]

Number of packets successfully written to current subport for each traffic class

2.90.2.2 uint32_t rte_sched_subport_stats::n_pkts_tc_dropped[RTE_SCHED_TRAFFIC_CLASSES_PER_PIPE]

Number of packets dropped by the current subport for each traffic class due to subport queues being full or congested

2.90.2.3 uint32_t rte_sched_subport_stats::n_bytes_tc[RTE_SCHED_TRAFFIC_CLASSES_PER_PIPE]

Number of bytes successfully written to current subport for each traffic class

2.90.2.4 uint32_t rte_sched_subport_stats::n_bytes_tc_dropped[RTE_SCHED_TRAFFIC_CLASSES_PER_PIPE]

Number of bytes dropped by the current subport for each traffic class due to subport queues being full or congested

2.91 rte_spinlock_recursive_t Struct Reference

Data Fields

- rte_spinlock_t sl
- volatile int user
- volatile int count

2.91.1 Detailed Description

The rte_spinlock_recursive_t type.

2.91.2 Field Documentation

2.91.2.1 rte_spinlock_t rte_spinlock_recursive_t::sl

the actual spinlock

2.91.2.2 volatile int rte_spinlock_recursive_t::user

core id using lock, -1 for unused

2.91.2.3 volatile int rte_spinlock_recursive_t::count

count of time this lock has been called

2.92 rte_spinlock_t Struct Reference

Data Fields

- volatile int locked

2.92.1 Detailed Description

The rte_spinlock_t type.

2.92.2 Field Documentation

2.92.2.1 volatile int rte_spinlock_t::locked

lock status 0 = unlocked, 1 = locked

2.93 rte_tailq_head Struct Reference

Data Fields

- struct rte_dummy_head tailq_head

2.93.1 Detailed Description

The structure defining a tailq header entry for storing in the rte_config structure in shared memory. Each tailq is identified by name. Any library storing a set of objects e.g. rings, mempools, hash-tables, is recommended to use an entry here, so as to make it easy for a multi-process app to find already-created elements in shared memory.

2.93.2 Field Documentation

2.93.2.1 struct rte_dummy_head rte_tailq_head::tailq_head

NOTE: must be first element

2.94 rte_timer Struct Reference

Data Fields

- uint64_t expire
- union rte_timer_status status
- uint64_t period
- rte_timer_cb_t * f
- void * arg

2.94.1 Detailed Description

A structure describing a timer in RTE.

2.94.2 Field Documentation

2.94.2.1 uint64_t rte_timer::expire

Time when timer expire.

2.94.2.2 union rte_timer_status rte_timer::status

Status of timer.

2.94.2.3 uint64_t rte_timer::period

Period of timer (0 if not periodic).

2.94.2.4 rte_timer_cb_t* rte_timer::f

Callback function.

2.94.2.5 void* rte_timer::arg

Argument to callback function.

2.95 rte_timer_status Union Reference

Data Fields

- uint32_t u32
- uint16_t state
- int16_t owner

2.95.1 Detailed Description

Timer status: A union of the state (stopped, pending, running, config) and an owner (the id of the lcore that owns the timer).

2.95.2 Field Documentation

2.95.2.1 uint16_t rte_timer_status::state

Stop, pending, running, config.

2.95.2.2 int16_t rte_timer_status::owner

The lcore that owns the timer.

2.95.2.3 uint32_t rte_timer_status::u32

To atomic-set status + owner.

2.96 rte_vlan_macip Union Reference

2.96.1 Detailed Description

Offload features

2.96.2 Field Documentation

2.96.2.1 uint16_t rte_vlan_macip::l3_len

L3 (IP) Header Length.

2.96.2.2 uint16_t rte_vlan_macip::l2_len

L2 (MAC) Header Length.

2.96.2.3 uint16_t rte_vlan_macip::vlan_tci

VLAN Tag Control Identifier (CPU order).

2.97 sctp_hdr Struct Reference

Data Fields

- uint16_t src_port

- uint16_t dst_port
- uint32_t tag
- uint32_t cksum

2.97.1 Detailed Description

SCTP Header

2.97.2 Field Documentation

2.97.2.1 uint16_t sctp_hdr::src_port

Source port.

2.97.2.2 uint16_t sctp_hdr::dst_port

Destin port.

2.97.2.3 uint32_t sctp_hdr::tag

Validation tag.

2.97.2.4 uint32_t sctp_hdr::cksum

Checksum.

2.98 tcp_hdr Struct Reference

Data Fields

- uint16_t src_port
- uint16_t dst_port
- uint32_t sent_seq
- uint32_t recv_ack
- uint8_t data_off
- uint8_t tcp_flags
- uint16_t rx_win
- uint16_t cksum
- uint16_t tcp_urp

2.98.1 Detailed Description

TCP Header

2.98.2 Field Documentation

2.98.2.1 uint16_t tcp_hdr::src_port

TCP source port.

2.98.2.2 uint16_t tcp_hdr::dst_port

TCP destination port.

2.98.2.3 uint32_t tcp_hdr::sent_seq

TX data sequence number.

2.98.2.4 uint32_t tcp_hdr::recv_ack

RX data acknowledgement sequence number.

2.98.2.5 uint8_t tcp_hdr::data_off

Data offset.

2.98.2.6 uint8_t tcp_hdr::tcp_flags

TCP flags

2.98.2.7 uint16_t tcp_hdr::rx_win

RX flow control window.

2.98.2.8 uint16_t tcp_hdr::cksum

TCP checksum.

2.98.2.9 uint16_t tcp_hdr::tcp_urp

TCP urgent pointer, if any.

2.99 udp_hdr Struct Reference

Data Fields

- uint16_t src_port
- uint16_t dst_port
- uint16_t dgram_len
- uint16_t dgram_cksum

2.99.1 Detailed Description

UDP Header

2.99.2 Field Documentation

2.99.2.1 uint16_t udp_hdr::src_port

UDP source port.

2.99.2.2 uint16_t udp_hdr::dst_port

UDP destination port.

2.99.2.3 uint16_t udp_hdr::dgram_len

UDP datagram length

2.99.2.4 uint16_t udp_hdr::dgram_cksum

UDP datagram checksum

2.100 vlan_hdr Struct Reference

Data Fields

- uint16_t vlan_tci

- uint16_t eth_proto

2.100.1 Detailed Description

Ethernet VLAN Header. Contains the 16-bit VLAN Tag Control Identifier and the Ethernet type of the encapsulated frame.

2.100.2 Field Documentation

2.100.2.1 uint16_t vlan_hdr::vlan_tci

Priority (3) + CFI (1) + Identifier Code (12)

2.100.2.2 uint16_t vlan_hdr::eth_proto

Ethernet type of encapsulated frame.

Chapter 3

File Documentation

3.1 rte_alarm.h File Reference

Typedefs

- typedef void(* rte_eal_alarm_callback)(void *arg)

Functions

- int rte_eal_alarm_set (uint64_t us, rte_eal_alarm_callback cb, void *cb_arg)
- int rte_eal_alarm_cancel (rte_eal_alarm_callback cb_fn, void *cb_arg)

3.1.1 Detailed Description

Alarm functions

Simple alarm-clock functionality supplied by eal. Does not require hpet support.

3.1.2 Typedef Documentation

3.1.2.1 typedef void(* rte_eal_alarm_callback)(void *arg)

Signature of callback back function called when an alarm goes off.

3.1.3 Function Documentation

3.1.3.1 int rte_eal_alarm_set (uint64_t *us*, rte_eal_alarm_callback *cb*, void * *cb_arg*)

Function to set a callback to be triggered when us microseconds have expired. Accuracy of timing to the microsecond is not guaranteed. The alarm function will not be called *before* the requested time, but may be called a short period of time afterwards. The alarm handler will be called only once. There is no need to call "rte_eal_alarm_cancel" from within the callback function.

Parameters

us	The time in microseconds before the callback is called
cb	The function to be called when the alarm expires
cb_arg	Pointer parameter to be passed to the callback function

Returns

On success, zero. On failure, a negative error number

3.1.3.2 int rte_eal_alarm_cancel (rte_eal_alarm_callback *cb_fn*, void * *cb_arg*)

Function to cancel an alarm callback which has been registered before.

Parameters

cb_fn	alarm callback
cb_arg	Pointer parameter to be passed to the callback function. To remove all copies of a given callback function, irrespective of parameter, (void *)-1 can be used here.

Returns

- The number of callbacks removed

3.2 rte_atomic.h File Reference

Data Structures

- struct rte_atomic16_t
- struct rte_atomic32_t
- struct rte_atomic64_t

Defines

- #define MPLOCKED
- #define rte_mb()
- #define rte_wmb()

- #define rte_rmb()
- #define rte_compiler_barrier()
- #define RTE_ATOMIC16_INIT(val)
- #define RTE_ATOMIC32_INIT(val)
- #define RTE_ATOMIC64_INIT(val)

Functions

- static int rte_atomic16_cmpset (volatile uint16_t *dst, uint16_t exp, uint16_t src)
- static void rte_atomic16_init (rte_atomic16_t *v)
- static int16_t rte_atomic16_read (const rte_atomic16_t *v)
- static void rte_atomic16_set (rte_atomic16_t *v, int16_t new_value)
- static void rte_atomic16_add (rte_atomic16_t *v, int16_t inc)
- static void rte_atomic16_sub (rte_atomic16_t *v, int16_t dec)
- static void rte_atomic16_inc (rte_atomic16_t *v)
- static void rte_atomic16_dec (rte_atomic16_t *v)
- static int16_t rte_atomic16_add_return (rte_atomic16_t *v, int16_t inc)
- static int16_t rte_atomic16_sub_return (rte_atomic16_t *v, int16_t dec)
- static int rte_atomic16_inc_and_test (rte_atomic16_t *v)
- static int rte_atomic16_dec_and_test (rte_atomic16_t *v)
- static int rte_atomic16_test_and_set (rte_atomic16_t *v)
- static void rte_atomic16_clear (rte_atomic16_t *v)
- static int rte_atomic32_cmpset (volatile uint32_t *dst, uint32_t exp, uint32_t src)
- static void rte_atomic32_init (rte_atomic32_t *v)
- static int32_t rte_atomic32_read (const rte_atomic32_t *v)
- static void rte_atomic32_set (rte_atomic32_t *v, int32_t new_value)
- static void rte_atomic32_add (rte_atomic32_t *v, int32_t inc)
- static void rte_atomic32_sub (rte_atomic32_t *v, int32_t dec)
- static void rte_atomic32_inc (rte_atomic32_t *v)
- static void rte_atomic32_dec (rte_atomic32_t *v)
- static int32_t rte_atomic32_add_return (rte_atomic32_t *v, int32_t inc)
- static int32_t rte_atomic32_sub_return (rte_atomic32_t *v, int32_t dec)
- static int rte_atomic32_inc_and_test (rte_atomic32_t *v)
- static int rte_atomic32_dec_and_test (rte_atomic32_t *v)
- static int rte_atomic32_test_and_set (rte_atomic32_t *v)
- static void rte_atomic32_clear (rte_atomic32_t *v)
- static int rte_atomic64_cmpset (volatile uint64_t *dst, uint64_t exp, uint64_t src)
- static void rte_atomic64_init (rte_atomic64_t *v)
- static int64_t rte_atomic64_read (rte_atomic64_t *v)
- static void rte_atomic64_set (rte_atomic64_t *v, int64_t new_value)
- static void rte_atomic64_add (rte_atomic64_t *v, int64_t inc)
- static void rte_atomic64_sub (rte_atomic64_t *v, int64_t dec)
- static void rte_atomic64_inc (rte_atomic64_t *v)
- static void rte_atomic64_dec (rte_atomic64_t *v)
- static int64_t rte_atomic64_add_return (rte_atomic64_t *v, int64_t inc)

- static int64_t rte_atomic64_sub_return (rte_atomic64_t *v, int64_t dec)
- static int rte_atomic64_inc_and_test (rte_atomic64_t *v)
- static int rte_atomic64_dec_and_test (rte_atomic64_t *v)
- static int rte_atomic64_test_and_set (rte_atomic64_t *v)
- static void rte_atomic64_clear (rte_atomic64_t *v)

3.2.1 Detailed Description

Atomic Operations

This file defines a generic API for atomic operations. The implementation is architecture-specific.

See lib/librte_eal/common/include/i686/arch/rte_atomic.h See lib/librte_eal/common/include/x86_64/arch/rte_atomic.h

Atomic Operations on x86_64

3.2.2 Define Documentation

3.2.2.1 #define MPLOCKED

Insert MP lock prefix.

3.2.2.2 #define rte_mb( )

General memory barrier.

Guarantees that the LOAD and STORE operations generated before the barrier occur before the LOAD and STORE operations generated after.

3.2.2.3 #define rte_wmb( )

Write memory barrier.

Guarantees that the STORE operations generated before the barrier occur before the STORE operations generated after.

3.2.2.4 #define rte_rmb( )

Read memory barrier.

Guarantees that the LOAD operations generated before the barrier occur before the LOAD operations generated after.

3.2.2.5 #define rte_compiler_barrier( )

Compiler barrier.

Guarantees that operation reordering does not occur at compile time for operations directly before and after the barrier.

3.2.2.6 #define RTE_ATOMIC16_INIT(*val*)

Static initializer for an atomic counter.

3.2.2.7 #define RTE_ATOMIC32_INIT(*val*)

Static initializer for an atomic counter.

3.2.2.8 #define RTE_ATOMIC64_INIT(*val*)

Static initializer for an atomic counter.

3.2.3 Function Documentation

3.2.3.1 static int rte_atomic16_cmpset (volatile uint16_t * *dst*, uint16_t *exp*, uint16_t *src*) [static]

Atomic compare and set.

(atomic) equivalent to: if (*dst == exp) *dst = src (all 16-bit words)

Parameters

dst	The destination location into which the value will be written.
exp	The expected value.
src	The new value.

Returns

Non-zero on success; 0 on failure.

3.2.3.2 static void rte_atomic16_init (rte_atomic16_t * *v*) [static]

Initialize an atomic counter.

Parameters

v	A pointer to the atomic counter.

3.2.3.3 static int16_t rte_atomic16_read (const rte_atomic16_t * v) [static]

Atomically read a 16-bit value from a counter.

Parameters

v	A pointer to the atomic counter.

Returns

The value of the counter.

3.2.3.4 static void rte_atomic16_set (rte_atomic16_t * v, int16_t new_value) [static]

Atomically set a counter to a 16-bit value.

Parameters

v	A pointer to the atomic counter.
new_value	The new value for the counter.

3.2.3.5 static void rte_atomic16_add (rte_atomic16_t * v, int16_t inc) [static]

Atomically add a 16-bit value to an atomic counter.

Parameters

v	A pointer to the atomic counter.
inc	The value to be added to the counter.

3.2.3.6 static void rte_atomic16_sub (rte_atomic16_t * v, int16_t dec) [static]

Atomically subtract a 16-bit value from an atomic counter.

Parameters

v	A pointer to the atomic counter.
dec	The value to be subtracted from the counter.

3.2.3.7 static void rte_atomic16_inc (rte_atomic16_t * v) [static]

Atomically increment a counter by one.

Parameters

v	A pointer to the atomic counter.

3.2.3.8 static void rte_atomic16_dec (rte_atomic16_t * v) [static]

Atomically decrement a counter by one.

Parameters

v	A pointer to the atomic counter.

3.2.3.9 static int16_t rte_atomic16_add_return (rte_atomic16_t * v, int16_t inc) [static]

Atomically add a 16-bit value to a counter and return the result.

Atomically adds the 16-bits value (inc) to the atomic counter (v) and returns the value of v after addition.

Parameters

v	A pointer to the atomic counter.
inc	The value to be added to the counter.

Returns

The value of v after the addition.

3.2.3.10 static int16_t rte_atomic16_sub_return (rte_atomic16_t * v, int16_t dec) [static]

Atomically subtract a 16-bit value from a counter and return the result.

Atomically subtracts the 16-bit value (inc) from the atomic counter (v) and returns the value of v after the subtraction.

Parameters

v	A pointer to the atomic counter.
dec	The value to be subtracted from the counter.

Returns

The value of v after the subtraction.

3.2.3.11 static int rte_atomic16_inc_and_test (rte_atomic16_t * v) [static]

Atomically increment a 16-bit counter by one and test.

Atomically increments the atomic counter (v) by one and returns true if the result is 0, or false in all other cases.

Parameters

v	A pointer to the atomic counter.

Returns

True if the result after the increment operation is 0; false otherwise.

3.2.3.12 static int rte_atomic16_dec_and_test (rte_atomic16_t * v) [static]

Atomically decrement a 16-bit counter by one and test.

Atomically decrements the atomic counter (v) by one and returns true if the result is 0, or false in all other cases.

Parameters

v	A pointer to the atomic counter.

Returns

True if the result after the decrement operation is 0; false otherwise.

3.2.3.13 static int rte_atomic16_test_and_set (rte_atomic16_t * v) [static]

Atomically test and set a 16-bit atomic counter.

If the counter value is already set, return 0 (failed). Otherwise, set the counter value to 1 and return 1 (success).

Parameters

v	A pointer to the atomic counter.

Returns

0 if failed; else 1, success.

3.2.3.14 static void rte_atomic16_clear (rte_atomic16_t * v) [static]

Atomically set a 16-bit counter to 0.

Parameters

v	A pointer to the atomic counter.

3.2.3.15 static int rte_atomic32_cmpset (volatile uint32_t * *dst*, uint32_t *exp*, uint32_t *src*) [static]

Atomic compare and set.

(atomic) equivalent to: if (*dst == exp) *dst = src (all 32-bit words)

Parameters

dst	The destination location into which the value will be written.
exp	The expected value.
src	The new value.

Returns

Non-zero on success; 0 on failure.

3.2.3.16 static void rte_atomic32_init (rte_atomic32_t * v) [static]

Initialize an atomic counter.

Parameters

v	A pointer to the atomic counter.

3.2.3.17 static int32_t rte_atomic32_read (const rte_atomic32_t * v) [static]

Atomically read a 32-bit value from a counter.

Parameters

v	A pointer to the atomic counter.

Returns

The value of the counter.

3.2.3.18 static void rte_atomic32_set (rte_atomic32_t * v, int32_t new_value) [static]

Atomically set a counter to a 32-bit value.

Parameters

v	A pointer to the atomic counter.
new_value	The new value for the counter.

3.2.3.19 static void rte_atomic32_add (rte_atomic32_t * v, int32_t inc) [static]

Atomically add a 32-bit value to an atomic counter.

Parameters

v	A pointer to the atomic counter.
inc	The value to be added to the counter.

3.2.3.20 static void rte_atomic32_sub (rte_atomic32_t * v, int32_t dec) [static]

Atomically subtract a 32-bit value from an atomic counter.

Parameters

v	A pointer to the atomic counter.
dec	The value to be subtracted from the counter.

3.2.3.21 static void rte_atomic32_inc (rte_atomic32_t * v) [static]

Atomically increment a counter by one.

Parameters

v	A pointer to the atomic counter.

3.2.3.22 static void rte_atomic32_dec (rte_atomic32_t * v) [static]

Atomically decrement a counter by one.

Parameters

v	A pointer to the atomic counter.

3.2.3.23 static int32_t rte_atomic32_add_return (rte_atomic32_t * v, int32_t inc) [static]

Atomically add a 32-bit value to a counter and return the result.

Atomically adds the 32-bits value (inc) to the atomic counter (v) and returns the value of v after addition.

Parameters

v	A pointer to the atomic counter.
inc	The value to be added to the counter.

Returns

The value of v after the addition.

3.2.3.24 static int32_t rte_atomic32_sub_return (rte_atomic32_t * v, int32_t dec) [static]

Atomically subtract a 32-bit value from a counter and return the result.

Atomically subtracts the 32-bit value (inc) from the atomic counter (v) and returns the value of v after the subtraction.

Parameters

v	A pointer to the atomic counter.
dec	The value to be subtracted from the counter.

Returns

The value of v after the subtraction.

3.2.3.25 static int rte_atomic32_inc_and_test (rte_atomic32_t * v) [static]

Atomically increment a 32-bit counter by one and test.

Atomically increments the atomic counter (v) by one and returns true if the result is 0, or false in all other cases.

Parameters

v	A pointer to the atomic counter.

Returns

True if the result after the increment operation is 0; false otherwise.

3.2.3.26 static int rte_atomic32_dec_and_test (rte_atomic32_t * v) [static]

Atomically decrement a 32-bit counter by one and test.

Atomically decrements the atomic counter (v) by one and returns true if the result is 0, or false in all other cases.

Parameters

v	A pointer to the atomic counter.

Returns

True if the result after the decrement operation is 0; false otherwise.

3.2.3.27 static int rte_atomic32_test_and_set (rte_atomic32_t * v) [static]

Atomically test and set a 32-bit atomic counter.

If the counter value is already set, return 0 (failed). Otherwise, set the counter value to 1 and return 1 (success).

Parameters

v	A pointer to the atomic counter.

Returns

0 if failed; else 1, success.

3.2.3.28 static void rte_atomic32_clear (rte_atomic32_t * v) [static]

Atomically set a 32-bit counter to 0.

Parameters

v	A pointer to the atomic counter.

3.2.3.29 static int rte_atomic64_cmpset (volatile uint64_t * *dst*, uint64_t *exp*, uint64_t *src*) [static]

An atomic compare and set function used by the mutex functions. (atomic) equivalent to: if (*dst == exp) *dst = src (all 64-bit words)

Parameters

dst	The destination into which the value will be written.
exp	The expected value.
src	The new value.

Returns

Non-zero on success; 0 on failure.

3.2.3.30 static void rte_atomic64_init (rte_atomic64_t * *v*) [static]

Initialize the atomic counter.

Parameters

v	A pointer to the atomic counter.

3.2.3.31 static int64_t rte_atomic64_read (rte_atomic64_t * *v*) [static]

Atomically read a 64-bit counter.

Parameters

v	A pointer to the atomic counter.

Returns

The value of the counter.

3.2.3.32 static void rte_atomic64_set (rte_atomic64_t * *v*, int64_t *new_value*) [static]

Atomically set a 64-bit counter.

Parameters

v	A pointer to the atomic counter.
new_value	The new value of the counter.

3.2.3.33 static void rte_atomic64_add (rte_atomic64_t * v, int64_t inc) [static]

Atomically add a 64-bit value to a counter.

Parameters

v	A pointer to the atomic counter.
inc	The value to be added to the counter.

3.2.3.34 static void rte_atomic64_sub (rte_atomic64_t * v, int64_t dec) [static]

Atomically subtract a 64-bit value from a counter.

Parameters

v	A pointer to the atomic counter.
dec	The value to be subtracted from the counter.

3.2.3.35 static void rte_atomic64_inc (rte_atomic64_t * v) [static]

Atomically increment a 64-bit counter by one and test.

Parameters

v	A pointer to the atomic counter.

3.2.3.36 static void rte_atomic64_dec (rte_atomic64_t * v) [static]

Atomically decrement a 64-bit counter by one and test.

Parameters

v	A pointer to the atomic counter.

3.2.3.37 static int64_t rte_atomic64_add_return (rte_atomic64_t * v, int64_t inc) [static]

Add a 64-bit value to an atomic counter and return the result.

Atomically adds the 64-bit value (inc) to the atomic counter (v) and returns the value of v after the addition.

Parameters

v	A pointer to the atomic counter.
inc	The value to be added to the counter.

Returns

The value of v after the addition.

3.2.3.38 static int64_t rte_atomic64_sub_return (rte_atomic64_t * v, int64_t dec) [static]

Subtract a 64-bit value from an atomic counter and return the result.

Atomically subtracts the 64-bit value (dec) from the atomic counter (v) and returns the value of v after the subtraction.

Parameters

v	A pointer to the atomic counter.
dec	The value to be subtracted from the counter.

Returns

The value of v after the subtraction.

3.2.3.39 static int rte_atomic64_inc_and_test (rte_atomic64_t * v) [static]

Atomically increment a 64-bit counter by one and test.

Atomically increments the atomic counter (v) by one and returns true if the result is 0, or false in all other cases.

Parameters

v	A pointer to the atomic counter.

Returns

True if the result after the addition is 0; false otherwise.

3.2.3.40 static int rte_atomic64_dec_and_test (rte_atomic64_t * v) [static]

Atomically decrement a 64-bit counter by one and test.

Atomically decrements the atomic counter (v) by one and returns true if the result is 0, or false in all other cases.

Parameters

v	A pointer to the atomic counter.

Returns

True if the result after subtraction is 0; false otherwise.

3.2.3.41 static int rte_atomic64_test_and_set (rte_atomic64_t * v) [static]

Atomically test and set a 64-bit atomic counter.

If the counter value is already set, return 0 (failed). Otherwise, set the counter value to 1 and return 1 (success).

Parameters

v	A pointer to the atomic counter.

Returns

0 if failed; else 1, success.

3.2.3.42 static void rte_atomic64_clear (rte_atomic64_t * v) [static]

Atomically set a 64-bit counter to 0.

Parameters

v	A pointer to the atomic counter.

3.3 rte_branch_prediction.h File Reference

Defines

- #define likely(x)
- #define unlikely(x)

3.3.1 Detailed Description

Branch Prediction Helpers in RTE

3.3.2 Define Documentation

3.3.2.1 #define likely(x)

Check if a branch is likely to be taken.

This compiler builtin allows the developer to indicate if a branch is likely to be taken. Example:
if (likely(x > 1)) do_stuff();

3.3.2.2 #define unlikely(x)

Check if a branch is unlikely to be taken.
This compiler builtin allows the developer to indicate if a branch is unlikely to be taken. Example:
if (unlikely(x < 1)) do_stuff();

3.4 rte_byteorder.h File Reference

Defines

- #define rte_bswap16(x)
- #define rte_bswap32(x)
- #define rte_bswap64(x)
- #define rte_cpu_to_le_16(x)
- #define rte_cpu_to_le_32(x)
- #define rte_cpu_to_le_64(x)
- #define rte_cpu_to_be_16(x)
- #define rte_cpu_to_be_32(x)
- #define rte_cpu_to_be_64(x)
- #define rte_le_to_cpu_16(x)
- #define rte_le_to_cpu_32(x)
- #define rte_le_to_cpu_64(x)
- #define rte_be_to_cpu_16(x)
- #define rte_be_to_cpu_32(x)
- #define rte_be_to_cpu_64(x)

3.4.1 Detailed Description

Byte Swap Operations
This file defines a generic API for byte swap operations. Part of the implementation is architecture-specific.

3.4.2 Define Documentation

3.4.2.1 #define rte_bswap16(x)

Swap bytes in a 16-bit value.

3.4.2.2 #define rte_bswap32(x)

Swap bytes in a 32-bit value.

3.4.2.3 #define rte_bswap64(x)

Swap bytes in a 64-bit value.

3.4.2.4 #define rte_cpu_to_le_16(x)

Convert a 16-bit value from CPU order to little endian.

3.4.2.5 #define rte_cpu_to_le_32(x)

Convert a 32-bit value from CPU order to little endian.

3.4.2.6 #define rte_cpu_to_le_64(x)

Convert a 64-bit value from CPU order to little endian.

3.4.2.7 #define rte_cpu_to_be_16(x)

Convert a 16-bit value from CPU order to big endian.

3.4.2.8 #define rte_cpu_to_be_32(x)

Convert a 32-bit value from CPU order to big endian.

3.4.2.9 #define rte_cpu_to_be_64(x)

Convert a 64-bit value from CPU order to big endian.

3.4.2.10 #define rte_le_to_cpu_16(x)

Convert a 16-bit value from little endian to CPU order.

3.4.2.11 #define rte_le_to_cpu_32(x)

Convert a 32-bit value from little endian to CPU order.

3.4.2.12 #define rte_le_to_cpu_64(x)

Convert a 64-bit value from little endian to CPU order.

3.4.2.13 #define rte_be_to_cpu_16(x)

Convert a 16-bit value from big endian to CPU order.

3.4.2.14 #define rte_be_to_cpu_32(x)

Convert a 32-bit value from big endian to CPU order.

3.4.2.15 #define rte_be_to_cpu_64(x)

Convert a 64-bit value from big endian to CPU order.

3.5 rte_common.h File Reference

Defines

- #define __rte_unused
- #define RTE_SET_USED(x)
- #define RTE_PTR_ADD(ptr, x)
- #define RTE_PTR_SUB(ptr, x)
- #define RTE_PTR_DIFF(ptr1, ptr2)
- #define RTE_PTR_ALIGN_FLOOR(ptr, align)
- #define RTE_ALIGN_FLOOR(val, align)
- #define RTE_PTR_ALIGN_CEIL(ptr, align)
- #define RTE_ALIGN_CEIL(val, align)
- #define RTE_PTR_ALIGN(ptr, align)
- #define RTE_ALIGN(val, align)
- #define RTE_BUILD_BUG_ON(condition)
- #define RTE_MIN(a, b)
- #define RTE_MAX(a, b)
- #define offsetof(TYPE, MEMBER)
- #define RTE_STR(x)
- #define RTE_LEN2MASK(ln, tp)
- #define RTE_DIM(a)

Functions

- static uintptr_t rte_align_floor_int (uintptr_t ptr, uintptr_t align)
- static int rte_is_aligned (void *ptr, unsigned align)
- static int rte_is_power_of_2 (uint32_t n)
- static uint32_t rte_align32pow2 (uint32_t x)
- static uint64_t rte_align64pow2 (uint64_t v)
- static uint32_t rte_bsf32 (uint32_t v)
- static uint64_t rte_str_to_size (const char *str)
- void rte_exit (int exit_code, const char *format,...)

3.5.1 Detailed Description

Generic, commonly-used macro and inline function definitions for Intel DPDK.

3.5.2 Define Documentation

3.5.2.1 #define __rte_unused

short definition to mark a function parameter unused

3.5.2.2 #define RTE_SET_USED(*x*)

definition to mark a variable or function parameter as used so as to avoid a compiler warning

3.5.2.3 #define RTE_PTR_ADD(*ptr, x*)

add a byte-value offset from a pointer

3.5.2.4 #define RTE_PTR_SUB(*ptr, x*)

subtract a byte-value offset from a pointer

3.5.2.5 #define RTE_PTR_DIFF(*ptr1, ptr2*)

get the difference between two pointer values, i.e. how far apart in bytes are the locations they point two. It is assumed that ptr1 is greater than ptr2.

3.5.2.6 #define RTE_PTR_ALIGN_FLOOR(ptr, align)

Macro to align a pointer to a given power-of-two. The resultant pointer will be a pointer of the same type as the first parameter, and point to an address no higher than the first parameter. Second parameter must be a power-of-two value.

3.5.2.7 #define RTE_ALIGN_FLOOR(val, align)

Macro to align a value to a given power-of-two. The resultant value will be of the same type as the first parameter, and will be no bigger than the first parameter. Second parameter must be a power-of-two value.

3.5.2.8 #define RTE_PTR_ALIGN_CEIL(ptr, align)

Macro to align a pointer to a given power-of-two. The resultant pointer will be a pointer of the same type as the first parameter, and point to an address no lower than the first parameter. Second parameter must be a power-of-two value.

3.5.2.9 #define RTE_ALIGN_CEIL(val, align)

Macro to align a value to a given power-of-two. The resultant value will be of the same type as the first parameter, and will be no lower than the first parameter. Second parameter must be a power-of-two value.

3.5.2.10 #define RTE_PTR_ALIGN(ptr, align)

Macro to align a pointer to a given power-of-two. The resultant pointer will be a pointer of the same type as the first parameter, and point to an address no lower than the first parameter. Second parameter must be a power-of-two value. This function is the same as RTE_PTR_ALIGN_CEIL

3.5.2.11 #define RTE_ALIGN(val, align)

Macro to align a value to a given power-of-two. The resultant value will be of the same type as the first parameter, and will be no lower than the first parameter. Second parameter must be a power-of-two value. This function is the same as RTE_ALIGN_CEIL

3.5.2.12 #define RTE_BUILD_BUG_ON(condition)

Triggers an error at compilation time if the condition is true.

3.5.2.13 #define RTE_MIN(a, b)

Macro to return the minimum of two numbers

3.5.2.14 #define RTE_MAX(*a, b*)

Macro to return the maximum of two numbers

3.5.2.15 #define offsetof(*TYPE, MEMBER*)

Return the offset of a field in a structure.

3.5.2.16 #define RTE_STR(*x*)

Take a macro value and get a string version of it

3.5.2.17 #define RTE_LEN2MASK(*ln, tp*)

Mask value of type <tp> for the first <ln> bit set.

3.5.2.18 #define RTE_DIM(*a*)

Number of elements in the array.

3.5.3 Function Documentation

3.5.3.1 static uintptr_t rte_align_floor_int (uintptr_t *ptr,* uintptr_t *align*) [static]

Function which rounds an unsigned int down to a given power-of-two value. Takes uintptr_t types as parameters, as this type of operation is most commonly done for pointer alignment. (See also RTE_ALIGN_FLOOR, RTE_ALIGN_CEIL, RTE_ALIGN, RTE_PTR_ALIGN_FLOOR, RTE_PTR_ALIGN_CEL, RTE_PTR_ALIGN macros)

Parameters

ptr	The value to be rounded down
align	The power-of-two of which the result must be a multiple.

Returns

Function returns a properly aligned value where align is a power-of-two. If align is not a power-of-two, result will be incorrect.

3.5.3.2 static int rte_is_aligned (void * *ptr,* unsigned *align*) [static]

Checks if a pointer is aligned to a given power-of-two value

Parameters

ptr	The pointer whose alignment is to be checked
align	The power-of-two value to which the ptr should be aligned

Returns

True(1) where the pointer is correctly aligned, false(0) otherwise

3.5.3.3 static int rte_is_power_of_2 (uint32_t *n*) [static]

Returns true if n is a power of 2

Parameters

n	Number to check

Returns

1 if true, 0 otherwise

3.5.3.4 static uint32_t rte_align32pow2 (uint32_t *x*) [static]

Aligns input parameter to the next power of 2

Parameters

x	The integer value to algin

Returns

Input parameter aligned to the next power of 2

3.5.3.5 static uint64_t rte_align64pow2 (uint64_t *v*) [static]

Aligns 64b input parameter to the next power of 2

Parameters

x	The 64b value to algin

Returns

Input parameter aligned to the next power of 2

3.5.3.6 static uint32_t rte_bsf32 (uint32_t *v*) [static]

Searches the input parameter for the least significant set bit (starting from zero). If a least significant 1 bit is found, its bit index is returned. If the content of the input parameter is zero, then the content of the return value is undefined.

Parameters

v	input parameter, should not be zero.

Returns

least significant set bit in the input parameter.

3.5.3.7 static uint64_t rte_str_to_size (const char * *str*) [static]

Converts a numeric string to the equivalent uint64_t value. As well as straight number conversion, also recognises the suffixes k, m and g for kilobytes, megabytes and gigabytes respectively.

If a negative number is passed in i.e. a string with the first non-black character being "-", zero is returned. Zero is also returned in the case of an error with the strtoull call in the function.

Parameters

str	String containing number to convert.

Returns

Number.

3.5.3.8 void rte_exit (int *exit_code*, const char * *format*, ...)

Function to terminate the application immediately, printing an error message and returning the exit_code back to the shell.

This function never returns

Parameters

exit_code	The exit code to be returned by the application
format	The format string to be used for printing the message. This can include printf format characters which will be expanded using any further parameters to the function.

3.6 rte_cpuflags.h File Reference

Enumerations

- enum rte_cpu_flag_t { RTE_CPUFLAG_SSE3, RTE_CPUFLAG_PCLMULQDQ, RTE_CPUFLAG_DTES64, RTE_CPUFLAG_MONITOR, RTE_CPUFLAG_DS_CPL, RTE_CPUFLAG_VMX, RTE_CPUFLAG_SMX, RTE_CPUFLAG_EIST, RTE_CPUFLAG_TM2, RTE_CPUFLAG_SSSE3, RTE_CPUFLAG_CNXT_ID, RTE_CPUFLAG_FMA, RTE_CPUFLAG_CMPXCHG16B, RTE_CPUFLAG_XTPR, RTE_CPUFLAG_PDCM, RTE_CPUFLAG_PCID, RTE_CPUFLAG_DCA, RTE_CPUFLAG_SSE4_1, RTE_CPUFLAG_SSE4_2, RTE_CPUFLAG_X2APIC, RTE_CPUFLAG_MOVBE, RTE_CPUFLAG_POPCNT, RTE_CPUFLAG_TSC_DEADLINE, RTE_CPUFLAG_AES, RTE_CPUFLAG_XSAVE, RTE_CPUFLAG_OSXSAVE, RTE_CPUFLAG_AVX, RTE_CPUFLAG_F16C, RTE_CPUFLAG_RDRAND, RTE_CPUFLAG_FPU, RTE_CPUFLAG_VME, RTE_CPUFLAG_DE, RTE_CPUFLAG_PSE, RTE_CPUFLAG_TSC, RTE_CPUFLAG_MSR, RTE_CPUFLAG_PAE, RTE_CPUFLAG_MCE, RTE_CPUFLAG_CX8, RTE_CPUFLAG_APIC, RTE_CPUFLAG_SEP, RTE_CPUFLAG_MTRR, RTE_CPUFLAG_PGE, RTE_CPUFLAG_MCA, RTE_CPUFLAG_CMOV, RTE_CPUFLAG_PAT, RTE_CPUFLAG_PSE36, RTE_CPUFLAG_PSN, RTE_CPUFLAG_CLFSH, RTE_CPUFLAG_DS, RTE_CPUFLAG_ACPI, RTE_CPUFLAG_MMX, RTE_CPUFLAG_FXSR, RTE_CPUFLAG_SSE, RTE_CPUFLAG_SSE2, RTE_CPUFLAG_SS, RTE_CPUFLAG_HTT, RTE_CPUFLAG_TM, RTE_CPUFLAG_PBE, RTE_CPUFLAG_DIGTEMP, RTE_CPUFLAG_TRBOBST, RTE_CPUFLAG_ARAT, RTE_CPUFLAG_PLN, RTE_CPUFLAG_ECMD, RTE_CPUFLAG_PTM, RTE_CPUFLAG_MPERF_APERF_MSR, RTE_CPUFLAG_ACNT2, RTE_CPUFLAG_ENERGY_EFF, RTE_CPUFLAG_FSGSBASE, RTE_CPUFLAG_BMI1, RTE_CPUFLAG_HLE, RTE_CPUFLAG_AVX2, RTE_CPUFLAG_SMEP, RTE_CPUFLAG_BMI2, RTE_CPUFLAG_ERMS, RTE_CPUFLAG_INVPCID, RTE_CPUFLAG_RTM, RTE_CPUFLAG_LAHF_SAHF, RTE_CPUFLAG_LZCNT, RTE_CPUFLAG_SYSCALL, RTE_CPUFLAG_XD, RTE_CPUFLAG_1GB_PG, RTE_CPUFLAG_RDTSCP, RTE_CPUFLAG_EM64T, RTE_CPUFLAG_INVTSC, RTE_CPUFLAG_NUMFLAGS }

Functions

- int rte_cpu_get_flag_enabled (enum rte_cpu_flag_t flag)
- void rte_cpu_check_supported (void)

3.6.1 Detailed Description

Simple API to determine available CPU features at runtime.

3.6.2 Enumeration Type Documentation

3.6.2.1 enum rte_cpu_flag_t

Enumeration of all CPU features supported

Enumerator:

RTE_CPUFLAG_SSE3 SSE3

RTE_CPUFLAG_PCLMULQDQ PCLMULQDQ

RTE_CPUFLAG_DTES64 DTES64

RTE_CPUFLAG_MONITOR MONITOR
RTE_CPUFLAG_DS_CPL DS_CPL
RTE_CPUFLAG_VMX VMX
RTE_CPUFLAG_SMX SMX
RTE_CPUFLAG_EIST EIST
RTE_CPUFLAG_TM2 TM2
RTE_CPUFLAG_SSSE3 SSSE3
RTE_CPUFLAG_CNXT_ID CNXT_ID
RTE_CPUFLAG_FMA FMA
RTE_CPUFLAG_CMPXCHG16B CMPXCHG16B
RTE_CPUFLAG_XTPR XTPR
RTE_CPUFLAG_PDCM PDCM
RTE_CPUFLAG_PCID PCID
RTE_CPUFLAG_DCA DCA
RTE_CPUFLAG_SSE4_1 SSE4_1
RTE_CPUFLAG_SSE4_2 SSE4_2
RTE_CPUFLAG_X2APIC X2APIC
RTE_CPUFLAG_MOVBE MOVBE
RTE_CPUFLAG_POPCNT POPCNT
RTE_CPUFLAG_TSC_DEADLINE TSC_DEADLINE
RTE_CPUFLAG_AES AES
RTE_CPUFLAG_XSAVE XSAVE
RTE_CPUFLAG_OSXSAVE OSXSAVE
RTE_CPUFLAG_AVX AVX
RTE_CPUFLAG_F16C F16C
RTE_CPUFLAG_RDRAND RDRAND
RTE_CPUFLAG_FPU FPU
RTE_CPUFLAG_VME VME
RTE_CPUFLAG_DE DE
RTE_CPUFLAG_PSE PSE
RTE_CPUFLAG_TSC TSC
RTE_CPUFLAG_MSR MSR
RTE_CPUFLAG_PAE PAE
RTE_CPUFLAG_MCE MCE
RTE_CPUFLAG_CX8 CX8
RTE_CPUFLAG_APIC APIC
RTE_CPUFLAG_SEP SEP
RTE_CPUFLAG_MTRR MTRR

RTE_CPUFLAG_PGE PGE
RTE_CPUFLAG_MCA MCA
RTE_CPUFLAG_CMOV CMOV
RTE_CPUFLAG_PAT PAT
RTE_CPUFLAG_PSE36 PSE36
RTE_CPUFLAG_PSN PSN
RTE_CPUFLAG_CLFSH CLFSH
RTE_CPUFLAG_DS DS
RTE_CPUFLAG_ACPI ACPI
RTE_CPUFLAG_MMX MMX
RTE_CPUFLAG_FXSR FXSR
RTE_CPUFLAG_SSE SSE
RTE_CPUFLAG_SSE2 SSE2
RTE_CPUFLAG_SS SS
RTE_CPUFLAG_HTT HTT
RTE_CPUFLAG_TM TM
RTE_CPUFLAG_PBE PBE
RTE_CPUFLAG_DIGTEMP DIGTEMP
RTE_CPUFLAG_TRBOBST TRBOBST
RTE_CPUFLAG_ARAT ARAT
RTE_CPUFLAG_PLN PLN
RTE_CPUFLAG_ECMD ECMD
RTE_CPUFLAG_PTM PTM
RTE_CPUFLAG_MPERF_APERF_MSR MPERF_APERF_MSR
RTE_CPUFLAG_ACNT2 ACNT2
RTE_CPUFLAG_ENERGY_EFF ENERGY_EFF
RTE_CPUFLAG_FSGSBASE FSGSBASE
RTE_CPUFLAG_BMI1 BMI1
RTE_CPUFLAG_HLE Hardware Lock elision
RTE_CPUFLAG_AVX2 AVX2
RTE_CPUFLAG_SMEP SMEP
RTE_CPUFLAG_BMI2 BMI2
RTE_CPUFLAG_ERMS ERMS
RTE_CPUFLAG_INVPCID INVPCID
RTE_CPUFLAG_RTM Transactional memory
RTE_CPUFLAG_LAHF_SAHF LAHF_SAHF
RTE_CPUFLAG_LZCNT LZCNT
RTE_CPUFLAG_SYSCALL SYSCALL

RTE_CPUFLAG_XD XD
RTE_CPUFLAG_1GB_PG 1GB_PG
RTE_CPUFLAG_RDTSCP RDTSCP
RTE_CPUFLAG_EM64T EM64T
RTE_CPUFLAG_INVTSC INVTSC
RTE_CPUFLAG_NUMFLAGS This should always be the last!

3.6.3 Function Documentation

3.6.3.1 int rte_cpu_get_flag_enabled (enum rte_cpu_flag_t *flag*)

Function for checking a CPU flag availability

Parameters

flag	CPU flag to query CPU for

Returns

1 if flag is available 0 if flag is not available -ENOENT if flag is invalid

3.6.3.2 void rte_cpu_check_supported (void)

This function checks that the currently used CPU supports the CPU features that were specified at compile time. It is called automatically within the EAL, so does not need to be used by applications.

3.7 rte_cycles.h File Reference

Functions

- static uint64_t rte_rdtsc (void)
- uint64_t rte_get_tsc_hz (void)
- static uint64_t rte_get_tsc_cycles (void)
- uint64_t rte_get_hpet_cycles (void)
- uint64_t rte_get_hpet_hz (void)
- int rte_eal_hpet_init (int make_default)
- static uint64_t rte_get_timer_cycles (void)
- static uint64_t rte_get_timer_hz (void)
- void rte_delay_us (unsigned us)
- static void rte_delay_ms (unsigned ms)

3.7.1 Detailed Description

Simple Time Reference Functions (Cycles and HPET).

3.7.2 Function Documentation

3.7.2.1 static uint64_t rte_rdtsc (void) [static]

Read the TSC register.

Returns

The TSC for this lcore.

3.7.2.2 uint64_t rte_get_tsc_hz (void)

Get the measured frequency of the RDTSC counter

Returns

The TSC frequency for this lcore

3.7.2.3 static uint64_t rte_get_tsc_cycles (void) [static]

Return the number of TSC cycles since boot

Returns

the number of cycles

3.7.2.4 uint64_t rte_get_hpet_cycles (void)

Return the number of HPET cycles since boot

This counter is global for all execution units. The number of cycles in one second can be retrieved using rte_get_hpet_hz().

Returns

the number of cycles

3.7.2.5 uint64_t rte_get_hpet_hz (void)

Get the number of HPET cycles in one second.

Returns

The number of cycles in one second.

3.7.2.6 int rte_eal_hpet_init (int *make_default*)

Initialise the HPET for use. This must be called before the rte_get_hpet_hz and rte_get_hpet_cycles APIs are called. If this function does not succeed, then the HPET functions are unavailable and should not be called.

Parameters

make_default	If set, the hpet timer becomes the default timer whose values are returned by the rte_-get_timer_hz/cycles API calls

Returns

0 on success, -1 on error, and the make_default parameter is ignored.

3.7.2.7 static uint64_t rte_get_timer_cycles (void) [static]

Get the number of cycles since boot from the default timer.

Returns

The number of cycles

3.7.2.8 static uint64_t rte_get_timer_hz (void) [static]

Get the number of cycles in one second for the default timer.

Returns

The number of cycles in one second.

3.7.2.9 void rte_delay_us (unsigned *us*)

Wait at least us microseconds.

Parameters

us	The number of microseconds to wait.

3.7.2.10 static void rte_delay_ms (unsigned *ms*) [static]

Wait at least ms milliseconds.

Parameters

ms	The number of milliseconds to wait.

3.8 rte_debug.h File Reference

Defines

- #define rte_panic_(func, format,...)

Functions

- void rte_dump_stack (void)
- void rte_dump_registers (void)

3.8.1 Detailed Description

Debug Functions in RTE

This file defines a generic API for debug operations. Part of the implementation is architecture-specific.

3.8.2 Define Documentation

3.8.2.1 #define rte_panic_ (*func, format, ...*)

Provide notification of a critical non-recoverable error and terminate execution abnormally.

Display the format string and its expanded arguments (printf-like).

In a linuxapp environment, this function dumps the stack and calls abort() resulting in a core dump if enabled.

The function never returns.

Parameters

format	The format string
args	The variable list of arguments.

3.8.3 Function Documentation

3.8.3.1 void rte_dump_stack (void)

Dump the stack of the calling core to the console.

3.8.3.2 void rte_dump_registers (void)

Dump the registers of the calling core to the console.

Note: Not implemented in a userapp environment; use gdb instead.

3.9 rte_eal.h File Reference

Data Structures

- struct rte_config

Defines

- #define RTE_VERSION
- #define RTE_MAGIC
- #define EAL_FLG_HIGH_IOPL
- #define RTE_EAL_TAILQ_RWLOCK
- #define RTE_EAL_MEMPOOL_RWLOCK
- #define RTE_EAL_TAILQ_INSERT_TAIL(idx, type, elm)
- #define RTE_EAL_TAILQ_REMOVE(idx, type, elm)
- #define RTE_EAL_TAILQ_EXIST_CHECK(idx)

Typedefs

- typedef void(* rte_usage_hook_t)(const char *prgname)

Enumerations

- enum rte_lcore_role_t
- enum rte_proc_type_t

Functions

- struct rte_config * rte_eal_get_configuration (void)
- enum rte_lcore_role_t rte_eal_lcore_role (unsigned lcore_id)
- enum rte_proc_type_t rte_eal_process_type (void)
- int rte_eal_init (int argc, char **argv)
- rte_usage_hook_t rte_set_application_usage_hook (rte_usage_hook_t usage_func)

3.9.1 Detailed Description

EAL Configuration API

3.9.2 Define Documentation

3.9.2.1 #define RTE_VERSION

The version of the RTE configuration structure.

3.9.2.2 #define RTE_MAGIC

Magic number written by the main partition when ready.

3.9.2.3 #define EAL_FLG_HIGH_IOPL

indicates high IO privilage in a linux env

3.9.2.4 #define RTE_EAL_TAILQ_RWLOCK

macro to get the lock of tailq in mem_config

3.9.2.5 #define RTE_EAL_MEMPOOL_RWLOCK

macro to get the multiple lock of mempool shared by mutiple-instance

3.9.2.6 #define RTE_EAL_TAILQ_INSERT_TAIL(*idx, type, elm*)

Utility macro to do a thread-safe tailq 'INSERT' of rte_mem_config

Parameters

idx	a kind of tailq define in enum rte_tailq_t
type	type of list(tailq head)
elm	The element will be added into the list

3.9.2.7 #define RTE_EAL_TAILQ_REMOVE(idx, type, elm)

Utility macro to do a thread-safe tailq 'REMOVE' of rte_mem_config

Parameters

idx	a kind of tailq define in enum rte_tailq_t
type	type of list(tailq head)
elm	The element will be remove from the list

3.9.2.8 #define RTE_EAL_TAILQ_EXIST_CHECK(idx)

macro to check TAILQ exist

Parameters

idx	a kind of tailq define in enum rte_tailq_t

3.9.3 Typedef Documentation

3.9.3.1 typedef void(* rte_usage_hook_t)(const char *prgname)

Usage function typedef used by the application usage function.

Use this function typedef to define and call rte_set_applcation_usage_hook() routine.

3.9.4 Enumeration Type Documentation

3.9.4.1 enum rte_lcore_role_t

The lcore role (used in RTE or not).

3.9.4.2 enum rte_proc_type_t

The type of process in a linuxapp, multi-process setup

3.9.5 Function Documentation

3.9.5.1 struct rte_config* rte_eal_get_configuration (void) [read]

Get the global configuration structure.

Returns

A pointer to the global configuration structure.

3.9.5.2 enum rte_lcore_role_t rte_eal_lcore_role (unsigned *lcore_id*)

Get a lcore's role.

Parameters

lcore_id	The identifier of the lcore.

Returns

The role of the lcore.

3.9.5.3 enum rte_proc_type_t rte_eal_process_type (void)

Get the process type in a multi-process setup

Returns

The process type

3.9.5.4 int rte_eal_init (int *argc*, char ** *argv*)

Initialize the Environment Abstraction Layer (EAL).

This function is to be executed on the MASTER lcore only, as soon as possible in the application's main() function.

The function finishes the initialization process that was started during boot (in case of baremetal) or before main() is called (in case of linuxapp). It puts the SLAVE lcores in the WAIT state.

When the multi-partition feature is supported, depending on the configuration (if CONFIG_RTE_EAL_MAIN_PARTITION is disabled), this function waits to ensure that the magic number is set before returning. See also the rte_eal_get_configuration() function. Note: This behavior may change in the future.

Parameters

argc	The argc argument that was given to the main() function.
argv	The argv argument that was given to the main() function.

Returns

- On success, the number of parsed arguments, which is greater or equal to zero. After the call to rte_eal_init(), all arguments argv[x] with x < ret may be modified and should not be accessed by the application.
- On failure, a negative error value.

3.9.5.5 rte_usage_hook_t rte_set_application_usage_hook (rte_usage_hook_t *usage_func*)

Add application usage routine callout from the eal_usage() routine.

This function allows the application to include its usage message in the EAL system usage message. - The routine rte_set_application_usage_hook() needs to be called before the rte_eal_init() routine in the application.

This routine is optional for the application and will behave as if the set routine was never called as the default behavior.

Parameters

func	The func argument is a function pointer to the application usage routine. Called function is defined using rte_usage_hook_t typedef, which is of the form void rte_usage_-func(const char * prgname).

Calling this routine with a NULL value will reset the usage hook routine and return the current value, which could be NULL.

Returns

- Returns the current value of the rte_application_usage pointer to allow the caller to daisy chain the usage routines if needing more then one.

3.10 rte_errno.h File Reference

Defines

- #define rte_errno
- #define __ELASTERROR

Enumerations

- enum { RTE_MIN_ERRNO, E_RTE_SECONDARY, E_RTE_NO_CONFIG, E_RTE_NO_TAILQ, R-TE_MAX_ERRNO }

Functions

- RTE_DECLARE_PER_LCORE (int, _rte_errno)

- const char * rte_strerror (int errnum)

3.10.1 Detailed Description

API for error cause tracking

3.10.2 Define Documentation

3.10.2.1 #define rte_errno

Error number value, stored per-thread, which can be queried after calls to certain functions to determine why those functions failed.

Uses standard values from errno.h wherever possible, with a small number of additional possible values for RTE-specific conditions.

3.10.2.2 #define __ELASTERROR

Check if we have a defined value for the max system-defined errno values. if no max defined, start from 1000 to prevent overlap with standard values

3.10.3 Enumeration Type Documentation

3.10.3.1 anonymous enum

Error types

Enumerator:

RTE_MIN_ERRNO Start numbering above std errno vals

E_RTE_SECONDARY Operation not allowed in secondary processes

E_RTE_NO_CONFIG Missing rte_config

E_RTE_NO_TAILQ Uninitialised TAILQ

RTE_MAX_ERRNO Max RTE error number

3.10.4 Function Documentation

3.10.4.1 RTE_DECLARE_PER_LCORE (int , _rte_errno)

Per core error number.

3.10.4.2 const char* rte_strerror (int *errnum*)

Function which returns a printable string describing a particular error code. For non-RTE-specific error codes, this function returns the value from the libc strerror function.

Parameters

errnum	The error number to be looked up - generally the value of rte_errno

Returns

A pointer to a thread-local string containing the text describing the error.

3.11 rte_ethdev.h File Reference

Data Structures

- struct rte_eth_stats
- struct rte_eth_link
- struct rte_eth_thresh
- struct rte_eth_rxmode
- struct rte_eth_rss_conf
- struct rte_eth_vlan_mirror
- struct rte_eth_vmdq_mirror_conf
- struct rte_eth_rss_reta
- struct rte_eth_dcb_rx_conf
- struct rte_eth_vmdq_dcb_tx_conf
- struct rte_eth_dcb_tx_conf
- struct rte_eth_vmdq_tx_conf
- struct rte_eth_vmdq_dcb_conf
- struct rte_eth_vmdq_rx_conf
- struct rte_eth_txmode
- struct rte_eth_rxconf
- struct rte_eth_txconf
- struct rte_eth_fc_conf
- struct rte_eth_pfc_conf
- struct rte_fdir_conf
- struct rte_fdir_filter
- struct rte_fdir_masks
- struct rte_eth_fdir
- struct rte_intr_conf
- struct rte_eth_conf
- struct rte_eth_dev_info
- struct eth_dev_ops
- struct rte_eth_dev
- struct rte_eth_dev_sriov
- struct rte_eth_dev_data
- struct eth_driver

Defines

- #define ETH_LINK_SPEED_AUTONEG
- #define ETH_LINK_SPEED_10
- #define ETH_LINK_SPEED_100
- #define ETH_LINK_SPEED_1000
- #define ETH_LINK_SPEED_10000
- #define ETH_LINK_AUTONEG_DUPLEX
- #define ETH_LINK_HALF_DUPLEX
- #define ETH_LINK_FULL_DUPLEX
- #define ETH_RSS
- #define ETH_DCB_NONE
- #define ETH_RSS_IPV4
- #define ETH_RSS_IPV4_TCP
- #define ETH_RSS_IPV6
- #define ETH_RSS_IPV6_EX
- #define ETH_RSS_IPV6_TCP
- #define ETH_RSS_IPV6_TCP_EX
- #define ETH_RSS_IPV4_UDP
- #define ETH_RSS_IPV6_UDP
- #define ETH_RSS_IPV6_UDP_EX
- #define ETH_VMDQ_MAX_VLAN_FILTERS
- #define ETH_DCB_NUM_USER_PRIORITIES
- #define ETH_VMDQ_DCB_NUM_QUEUES
- #define ETH_DCB_NUM_QUEUES
- #define ETH_DCB_PG_SUPPORT
- #define ETH_DCB_PFC_SUPPORT
- #define ETH_VLAN_STRIP_OFFLOAD
- #define ETH_VLAN_FILTER_OFFLOAD
- #define ETH_VLAN_EXTEND_OFFLOAD
- #define ETH_VLAN_STRIP_MASK
- #define ETH_VLAN_FILTER_MASK
- #define ETH_VLAN_EXTEND_MASK
- #define ETH_VLAN_ID_MAX
- #define ETH_NUM_RECEIVE_MAC_ADDR
- #define ETH_VMDQ_NUM_UC_HASH_ARRAY
- #define ETH_VMDQ_ACCEPT_UNTAG
- #define ETH_VMDQ_ACCEPT_HASH_MC
- #define ETH_VMDQ_ACCEPT_HASH_UC
- #define ETH_VMDQ_ACCEPT_BROADCAST
- #define ETH_VMDQ_ACCEPT_MULTICAST
- #define ETH_VMDQ_NUM_MIRROR_RULE
- #define ETH_VMDQ_POOL_MIRROR
- #define ETH_VMDQ_UPLINK_MIRROR
- #define ETH_VMDQ_DOWNLIN_MIRROR
- #define ETH_VMDQ_VLAN_MIRROR

- #define ETH_TXQ_FLAGS_NOMULTSEGS
- #define ETH_TXQ_FLAGS_NOREFCOUNT
- #define ETH_TXQ_FLAGS_NOMULTMEMP
- #define ETH_TXQ_FLAGS_NOVLANOFFL
- #define ETH_TXQ_FLAGS_NOXSUMSCTP
- #define ETH_TXQ_FLAGS_NOXSUMUDP
- #define ETH_TXQ_FLAGS_NOXSUMTCP

Typedefs

- typedef uint32_t(∗ eth_rx_queue_count_t)(struct rte_eth_dev ∗dev, uint16_t rx_queue_id)
- typedef int(∗ eth_rx_descriptor_done_t)(void ∗rxq, uint16_t offset)
- typedef void(∗ rte_eth_dev_cb_fn)(uint8_t port_id, enum rte_eth_event_type event, void ∗cb_arg)

Enumerations

- enum rte_eth_rx_mq_mode { ETH_MQ_RX_NONE, ETH_MQ_RX_RSS, ETH_MQ_RX_DCB, ETH_MQ_RX_DCB_RSS, ETH_MQ_RX_VMDQ_ONLY, ETH_MQ_RX_VMDQ_RSS, ETH_MQ_RX_VMDQ_DCB, ETH_MQ_RX_VMDQ_DCB_RSS }
- enum rte_eth_tx_mq_mode { ETH_MQ_TX_NONE, ETH_MQ_TX_DCB, ETH_MQ_TX_VMDQ_DCB, ETH_MQ_TX_VMDQ_ONLY }
- enum rte_eth_nb_tcs { ETH_4_TCS, ETH_8_TCS }
- enum rte_eth_nb_pools { ETH_8_POOLS, ETH_16_POOLS, ETH_32_POOLS, ETH_64_POOLS }
- enum rte_eth_fc_mode { RTE_FC_NONE, RTE_FC_RX_PAUSE, RTE_FC_TX_PAUSE, RTE_FC_FULL }
- enum rte_fdir_mode { RTE_FDIR_MODE_NONE, RTE_FDIR_MODE_SIGNATURE, RTE_FDIR_MODE_PERFECT }
- enum rte_fdir_pballoc_type { RTE_FDIR_PBALLOC_64K, RTE_FDIR_PBALLOC_128K, RTE_FDIR_PBALLOC_256K }
- enum rte_fdir_status_mode { RTE_FDIR_NO_REPORT_STATUS, RTE_FDIR_REPORT_STATUS, RTE_FDIR_REPORT_STATUS_ALWAYS }
- enum rte_l4type { RTE_FDIR_L4TYPE_NONE, RTE_FDIR_L4TYPE_UDP, RTE_FDIR_L4TYPE_TCP, RTE_FDIR_L4TYPE_SCTP }
- enum rte_iptype { RTE_FDIR_IPTYPE_IPV4, RTE_FDIR_IPTYPE_IPV6 }
- enum rte_eth_event_type { RTE_ETH_EVENT_UNKNOWN, RTE_ETH_EVENT_INTR_LSC, RTE_ETH_EVENT_MAX }

Functions

- uint8_t rte_eth_dev_count (void)
- struct rte_eth_dev ∗ rte_eth_dev_allocate (void)
- int rte_igb_pmd_init (void)
- int rte_em_pmd_init (void)
- int rte_igbvf_pmd_init (void)
- int rte_ixgbe_pmd_init (void)

- int rte_ixgbevf_pmd_init (void)
- int rte_virtio_pmd_init (void)
- int rte_vmxnet3_pmd_init (void)
- static int rte_pmd_init_all (void)
- int rte_eth_dev_configure (uint8_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 (uint8_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 (uint8_t port_id, uint16_t tx_queue_id, uint16_t nb_tx_desc, unsigned int socket_id, const struct rte_eth_txconf *tx_conf)
- int rte_eth_dev_start (uint8_t port_id)
- void rte_eth_dev_stop (uint8_t port_id)
- void rte_eth_dev_close (uint8_t port_id)
- void rte_eth_promiscuous_enable (uint8_t port_id)
- void rte_eth_promiscuous_disable (uint8_t port_id)
- int rte_eth_promiscuous_get (uint8_t port_id)
- void rte_eth_allmulticast_enable (uint8_t port_id)
- void rte_eth_allmulticast_disable (uint8_t port_id)
- int rte_eth_allmulticast_get (uint8_t port_id)
- void rte_eth_link_get (uint8_t port_id, struct rte_eth_link *link)
- void rte_eth_link_get_nowait (uint8_t port_id, struct rte_eth_link *link)
- void rte_eth_stats_get (uint8_t port_id, struct rte_eth_stats *stats)
- void rte_eth_stats_reset (uint8_t port_id)
- int rte_eth_dev_set_tx_queue_stats_mapping (uint8_t port_id, uint16_t tx_queue_id, uint8_t stat_idx)
- int rte_eth_dev_set_rx_queue_stats_mapping (uint8_t port_id, uint16_t rx_queue_id, uint8_t stat_idx)
- void rte_eth_macaddr_get (uint8_t port_id, struct ether_addr *mac_addr)
- void rte_eth_dev_info_get (uint8_t port_id, struct rte_eth_dev_info *dev_info)
- int rte_eth_dev_vlan_filter (uint8_t port_id, uint16_t vlan_id, int on)
- int rte_eth_dev_set_vlan_strip_on_queue (uint8_t port_id, uint16_t rx_queue_id, int on)
- int rte_eth_dev_set_vlan_ether_type (uint8_t port_id, uint16_t tag_type)
- int rte_eth_dev_set_vlan_offload (uint8_t port_id, int offload_mask)
- int rte_eth_dev_get_vlan_offload (uint8_t port_id)
- static uint16_t rte_eth_rx_burst (uint8_t port_id, uint16_t queue_id, struct rte_mbuf **rx_pkts, uint16_t nb_pkts)
- static uint32_t rte_eth_rx_queue_count (uint8_t port_id, uint16_t queue_id)
- static int rte_eth_rx_descriptor_done (uint8_t port_id, uint16_t queue_id, uint16_t offset)
- static uint16_t rte_eth_tx_burst (uint8_t port_id, uint16_t queue_id, struct rte_mbuf **tx_pkts, uint16_t nb_pkts)
- int rte_eth_dev_fdir_add_signature_filter (uint8_t port_id, struct rte_fdir_filter *fdir_filter, uint8_t rx_queue)
- int rte_eth_dev_fdir_update_signature_filter (uint8_t port_id, struct rte_fdir_filter *fdir_ftr, uint8_t rx_queue)
- int rte_eth_dev_fdir_remove_signature_filter (uint8_t port_id, struct rte_fdir_filter *fdir_ftr)
- int rte_eth_dev_fdir_get_infos (uint8_t port_id, struct rte_eth_fdir *fdir)
- int rte_eth_dev_fdir_add_perfect_filter (uint8_t port_id, struct rte_fdir_filter *fdir_filter, uint16_t soft_id, uint8_t rx_queue, uint8_t drop)

- int rte_eth_dev_fdir_update_perfect_filter (uint8_t port_id, struct rte_fdir_filter ∗fdir_filter, uint16_t soft-_id, uint8_t rx_queue, uint8_t drop)
- int rte_eth_dev_fdir_remove_perfect_filter (uint8_t port_id, struct rte_fdir_filter ∗fdir_filter, uint16_-t soft_id)
- int rte_eth_dev_fdir_set_masks (uint8_t port_id, struct rte_fdir_masks ∗fdir_mask)
- int rte_eth_dev_callback_register (uint8_t port_id, enum rte_eth_event_type event, rte_eth_dev_cb_fn cb_fn, void ∗cb_arg)
- int rte_eth_dev_callback_unregister (uint8_t port_id, enum rte_eth_event_type event, rte_eth_dev_-cb_fn cb_fn, void ∗cb_arg)
- int rte_eth_led_on (uint8_t port_id)
- int rte_eth_led_off (uint8_t port_id)
- int rte_eth_dev_flow_ctrl_set (uint8_t port_id, struct rte_eth_fc_conf ∗fc_conf)
- int rte_eth_dev_priority_flow_ctrl_set (uint8_t port_id, struct rte_eth_pfc_conf ∗pfc_conf)
- int rte_eth_dev_mac_addr_add (uint8_t port, struct ether_addr ∗mac_addr, uint32_t pool)
- int rte_eth_dev_mac_addr_remove (uint8_t port, struct ether_addr ∗mac_addr)
- int rte_eth_dev_rss_reta_update (uint8_t port, struct rte_eth_rss_reta ∗reta_conf)
- int rte_eth_dev_rss_reta_query (uint8_t port, struct rte_eth_rss_reta ∗reta_conf)
- int rte_eth_dev_uc_hash_table_set (uint8_t port, struct ether_addr ∗addr, uint8_t on)
- int rte_eth_dev_uc_all_hash_table_set (uint8_t port, uint8_t on)
- int rte_eth_dev_set_vf_rxmode (uint8_t port, uint16_t vf, uint16_t rx_mode, uint8_t on)
- int rte_eth_dev_set_vf_tx (uint8_t port, uint16_t vf, uint8_t on)
- int rte_eth_dev_set_vf_rx (uint8_t port, uint16_t vf, uint8_t on)
- int rte_eth_dev_set_vf_vlan_filter (uint8_t port, uint16_t vlan_id, uint64_t vf_mask, uint8_t vlan_on)
- int rte_eth_mirror_rule_set (uint8_t port_id, struct rte_eth_vmdq_mirror_conf ∗mirror_conf, uint8_t rule_id, uint8_t on)
- int rte_eth_mirror_rule_reset (uint8_t port_id, uint8_t rule_id)
- int rte_eth_dev_bypass_init (uint8_t port)
- int rte_eth_dev_bypass_state_show (uint8_t port, uint32_t ∗state)
- int rte_eth_dev_bypass_state_set (uint8_t port, uint32_t ∗new_state)
- int rte_eth_dev_bypass_event_show (uint8_t port, uint32_t event, uint32_t ∗state)
- int rte_eth_dev_bypass_event_store (uint8_t port, uint32_t event, uint32_t state)
- int rte_eth_dev_wd_timeout_store (uint8_t port, uint32_t timeout)
- int rte_eth_dev_bypass_ver_show (uint8_t port, uint32_t ∗ver)
- int rte_eth_dev_bypass_wd_timeout_show (uint8_t port, uint32_t ∗wd_timeout)
- int rte_eth_dev_bypass_wd_reset (uint8_t port)

3.11.1 Detailed Description

RTE Ethernet Device API

The Ethernet Device API is composed of two parts:

- The application-oriented Ethernet API that includes functions to setup an Ethernet device (configure it, setup its RX and TX queues and start it), to get its MAC address, the speed and the status of its physical link, to receive and to transmit packets, and so on.

- The driver-oriented Ethernet API that exports a function allowing an Ethernet Poll Mode Driver (PMD) to simultaneously register itself as an Ethernet device driver and as a PCI driver for a set of matching PCI [Ethernet] devices classes.

By default, all the functions of the Ethernet Device API exported by a PMD are lock-free functions which assume to not be invoked in parallel on different logical cores to work on the same target object. For instance, the receive function of a PMD cannot be invoked in parallel on two logical cores to poll the same RX queue [of the same port]. Of course, this function can be invoked in parallel by different logical cores on different RX queues. It is the responsibility of the upper level application to enforce this rule.

If needed, parallel accesses by multiple logical cores to shared queues shall be explicitly protected by dedicated inline lock-aware functions built on top of their corresponding lock-free functions of the PMD API.

In all functions of the Ethernet API, the Ethernet device is designated by an integer >= 0 named the device port identifier.

At the Ethernet driver level, Ethernet devices are represented by a generic data structure of type *rte_eth_dev*.

Ethernet devices are dynamically registered during the PCI probing phase performed at EAL initialization time. When an Ethernet device is being probed, an *rte_eth_dev* structure and a new port identifier are allocated for that device. Then, the eth_dev_init() function supplied by the Ethernet driver matching the probed PCI device is invoked to properly initialize the device.

The role of the device init function consists of resetting the hardware, checking access to Non-volatile - Memory (NVM), reading the MAC address from NVM etc.

If the device init operation is successful, the correspondence between the port identifier assigned to the new device and its associated *rte_eth_dev* structure is effectively registered. Otherwise, both the *rte_eth_dev* structure and the port identifier are freed.

The functions exported by the application Ethernet API to setup a device designated by its port identifier must be invoked in the following order:

- rte_eth_dev_configure()
- rte_eth_tx_queue_setup()
- rte_eth_rx_queue_setup()
- rte_eth_dev_start()

Then, the network application can invoke, in any order, the functions exported by the Ethernet API to get the MAC address of a given device, to get the speed and the status of a device physical link, to receive/transmit [burst of] packets, and so on.

If the application wants to change the configuration (i.e. call rte_eth_dev_configure(), rte_eth_tx_queue_setup(), or rte_eth_rx_queue_setup()), it must call rte_eth_dev_stop() first to stop the device and then do the reconfiguration before calling rte_eth_dev_start() again. The tramsit and receive functions should not be invoked when the device is stopped.

Please note that some configuration is not stored between calls to rte_eth_dev_stop()/rte_eth_dev_start(). The following configuration will be retained:

- flow control settings

- receive mode configuration (promiscuous mode, hardware checksum mode, RSS/VMDQ settings etc.)
- VLAN filtering configuration
- MAC addresses supplied to MAC address array
- flow director filtering mode (but not filtering rules)
- NIC queue statistics mappings

Any other configuration will not be stored and will need to be re-entered after a call to rte_eth_dev_start().

Finally, a network application can close an Ethernet device by invoking the rte_eth_dev_close() function.

Each function of the application Ethernet API invokes a specific function of the PMD that controls the target device designated by its port identifier. For this purpose, all device-specific functions of an Ethernet driver are supplied through a set of pointers contained in a generic structure of type ∗eth_dev_ops∗. The address of the ∗eth_dev_ops∗ structure is stored in the ∗rte_eth_dev∗ structure by the device init function of the Ethernet driver, which is invoked during the PCI probing phase, as explained earlier.

In other words, each function of the Ethernet API simply retrieves the ∗rte_eth_dev∗ structure associated with the device port identifier and performs an indirect invocation of the corresponding driver function supplied in the ∗eth_dev_ops∗ structure of the ∗rte_eth_dev∗ structure.

For performance reasons, the address of the burst-oriented RX and TX functions of the Ethernet driver are not contained in the ∗eth_dev_ops∗ structure. Instead, they are directly stored at the beginning of the ∗rte_eth_dev∗ structure to avoid an extra indirect memory access during their invocation.

RTE ethernet device drivers do not use interrupts for transmitting or receiving. Instead, Ethernet drivers export Poll-Mode receive and transmit functions to applications. Both receive and transmit functions are packet-burst oriented to minimize their cost per packet through the following optimizations:

- Sharing among multiple packets the incompressible cost of the invocation of receive/transmit functions.
- Enabling receive/transmit functions to take advantage of burst-oriented hardware features (L1 cache, prefetch instructions, NIC head/tail registers) to minimize the number of CPU cycles per packet, for instance, by avoiding useless read memory accesses to ring descriptors, or by systematically using arrays of pointers that exactly fit L1 cache line boundaries and sizes.

The burst-oriented receive function does not provide any error notification, to avoid the corresponding overhead. As a hint, the upper-level application might check the status of the device link once being systematically returned a 0 value by the receive function of the driver for a given number of tries.

3.11.2 Define Documentation

3.11.2.1 #define ETH_LINK_SPEED_AUTONEG

Auto-negotiate link speed.

3.11.2.2 #define ETH_LINK_SPEED_10

10 megabits/second.

3.11.2.3 #define ETH_LINK_SPEED_100

100 megabits/second.

3.11.2.4 #define ETH_LINK_SPEED_1000

1 gigabits/second.

3.11.2.5 #define ETH_LINK_SPEED_10000

10 gigabits/second.

3.11.2.6 #define ETH_LINK_AUTONEG_DUPLEX

Auto-negotiate duplex.

3.11.2.7 #define ETH_LINK_HALF_DUPLEX

Half-duplex connection.

3.11.2.8 #define ETH_LINK_FULL_DUPLEX

Full-duplex connection.

3.11.2.9 #define ETH_RSS

for rx mq mode backward compatible

3.11.2.10 #define ETH_DCB_NONE

for tx mq mode backward compatible

3.11.2.11 #define ETH_RSS_IPV4

IPv4 packet.

3.11.2.12 #define ETH_RSS_IPV4_TCP

IPv4/TCP packet.

3.11.2.13 #define ETH_RSS_IPV6

IPv6 packet.

3.11.2.14 #define ETH_RSS_IPV6_EX

IPv6 packet with extension headers.

3.11.2.15 #define ETH_RSS_IPV6_TCP

IPv6/TCP packet.

3.11.2.16 #define ETH_RSS_IPV6_TCP_EX

IPv6/TCP with extension headers.

3.11.2.17 #define ETH_RSS_IPV4_UDP

IPv4/UDP packet.

3.11.2.18 #define ETH_RSS_IPV6_UDP

IPv6/UDP packet.

3.11.2.19 #define ETH_RSS_IPV6_UDP_EX

IPv6/UDP with extension headers.

3.11.2.20 #define ETH_VMDQ_MAX_VLAN_FILTERS

Maximum nb. of VMDQ vlan filters.

3.11.2.21 #define ETH_DCB_NUM_USER_PRIORITIES

Maximum nb. of DCB priorities.

3.11.2.22 #define ETH_VMDQ_DCB_NUM_QUEUES

Maximum nb. of VMDQ DCB queues.

3.11.2.23 #define ETH_DCB_NUM_QUEUES

Maximum nb. of DCB queues.

3.11.2.24 #define ETH_DCB_PG_SUPPORT

Priority Group(ETS) support.

3.11.2.25 #define ETH_DCB_PFC_SUPPORT

Priority Flow Control support.

3.11.2.26 #define ETH_VLAN_STRIP_OFFLOAD

VLAN Strip On/Off

3.11.2.27 #define ETH_VLAN_FILTER_OFFLOAD

VLAN Filter On/Off

3.11.2.28 #define ETH_VLAN_EXTEND_OFFLOAD

VLAN Extend On/Off

3.11.2.29 #define ETH_VLAN_STRIP_MASK

VLAN Strip setting mask

3.11.2.30 #define ETH_VLAN_FILTER_MASK

VLAN Filter setting mask

3.11.2.31 #define ETH_VLAN_EXTEND_MASK

VLAN Extend setting mask

3.11.2.32 #define ETH_VLAN_ID_MAX

VLAN ID is in lower 12 bits

3.11.2.33 #define ETH_NUM_RECEIVE_MAC_ADDR

Maximum nb. of receive mac addr.

3.11.2.34 #define ETH_VMDQ_NUM_UC_HASH_ARRAY

Maximum nb. of UC hash array.

3.11.2.35 #define ETH_VMDQ_ACCEPT_UNTAG

accept untagged packets.

3.11.2.36 #define ETH_VMDQ_ACCEPT_HASH_MC

accept packets in multicast table .

3.11.2.37 #define ETH_VMDQ_ACCEPT_HASH_UC

accept packets in unicast table.

3.11.2.38 #define ETH_VMDQ_ACCEPT_BROADCAST

accept broadcast packets.

3.11.2.39 #define ETH_VMDQ_ACCEPT_MULTICAST

multicast promiscuous.

3.11.2.40 #define ETH_VMDQ_NUM_MIRROR_RULE

Maximum nb. of mirror rules. .

3.11.2.41 #define ETH_VMDQ_POOL_MIRROR

Virtual Pool Mirroring.

3.11.2.42 #define ETH_VMDQ_UPLINK_MIRROR

Uplink Port Mirroring.

3.11.2.43 #define ETH_VMDQ_DOWNLIN_MIRROR

Downlink Port Mirroring.

3.11.2.44 #define ETH_VMDQ_VLAN_MIRROR

VLAN Mirroring.

3.11.2.45 #define ETH_TXQ_FLAGS_NOMULTSEGS

nb_segs=1 for all mbufs

3.11.2.46 #define ETH_TXQ_FLAGS_NOREFCOUNT

refcnt can be ignored

3.11.2.47 #define ETH_TXQ_FLAGS_NOMULTMEMP

all bufs come from same mempool

3.11.2.48 #define ETH_TXQ_FLAGS_NOVLANOFFL

disable VLAN offload

3.11.2.49 #define ETH_TXQ_FLAGS_NOXSUMSCTP

disable SCTP checksum offload

3.11.2.50 #define ETH_TXQ_FLAGS_NOXSUMUDP

disable UDP checksum offload

3.11.2.51 #define ETH_TXQ_FLAGS_NOXSUMTCP

disable TCP checksum offload

3.11.3 Typedef Documentation

3.11.3.1 typedef uint32_t(* eth_rx_queue_count_t)(struct rte_eth_dev *dev, uint16_t rx_queue_id)

number of available descriptors on a receive queue of an Ethernet device.

3.11.3.2 typedef int(* eth_rx_descriptor_done_t)(void *rxq, uint16_t offset)

DD bit of specific RX descriptor

3.11.3.3 typedef void(* rte_eth_dev_cb_fn)(uint8_t port_id, enum rte_eth_event_type event, void *cb_arg)

user application callback to be registered for interrupts

3.11.4 Enumeration Type Documentation

3.11.4.1 enum rte_eth_rx_mq_mode

A set of values to identify what method is to be used to route packets to multiple queues.

Enumerator:

ETH_MQ_RX_NONE None of DCB,RSS or VMDQ mode

ETH_MQ_RX_RSS For RX side, only RSS is on

ETH_MQ_RX_DCB For RX side,only DCB is on.

ETH_MQ_RX_DCB_RSS Both DCB and RSS enable

ETH_MQ_RX_VMDQ_ONLY Only VMDQ, no RSS nor DCB

ETH_MQ_RX_VMDQ_RSS RSS mode with VMDQ

ETH_MQ_RX_VMDQ_DCB Use VMDQ+DCB to route traffic to queues

ETH_MQ_RX_VMDQ_DCB_RSS Enable both VMDQ and DCB in VMDq

3.11.4.2 enum rte_eth_tx_mq_mode

A set of values to identify what method is to be used to transmit packets using multi-TCs.

Enumerator:

ETH_MQ_TX_NONE It is in neither DCB nor VT mode.

ETH_MQ_TX_DCB For TX side,only DCB is on.

ETH_MQ_TX_VMDQ_DCB For TX side,both DCB and VT is on.

ETH_MQ_TX_VMDQ_ONLY Only VT on, no DCB

3.11.4.3 enum rte_eth_nb_tcs

This enum indicates the possible number of traffic classes in DCB configratioins

Enumerator:

ETH_4_TCS 4 TCs with DCB.

ETH_8_TCS 8 TCs with DCB.

3.11.4.4 enum rte_eth_nb_pools

This enum indicates the possible number of queue pools in VMDQ configurations.

Enumerator:

ETH_8_POOLS 8 VMDq pools.

ETH_16_POOLS 16 VMDq pools.

ETH_32_POOLS 32 VMDq pools.

ETH_64_POOLS 64 VMDq pools.

3.11.4.5 enum rte_eth_fc_mode

This enum indicates the flow control mode

Enumerator:

RTE_FC_NONE Disable flow control.

RTE_FC_RX_PAUSE RX pause frame, enable flowctrl on TX side.

RTE_FC_TX_PAUSE TX pause frame, enable flowctrl on RX side.

RTE_FC_FULL Enable flow control on both side.

3.11.4.6 enum rte_fdir_mode

Flow Director setting modes: none (default), signature or perfect.

Enumerator:

RTE_FDIR_MODE_NONE Disable FDIR support.

RTE_FDIR_MODE_SIGNATURE Enable FDIR signature filter mode.

RTE_FDIR_MODE_PERFECT Enable FDIR perfect filter mode.

3.11.4.7 enum rte_fdir_pballoc_type

Memory space that can be configured to store Flow Director filters in the board memory.

Enumerator:

RTE_FDIR_PBALLOC_64K 64k.

RTE_FDIR_PBALLOC_128K 128k.

RTE_FDIR_PBALLOC_256K 256k.

3.11.4.8 enum rte_fdir_status_mode

Select report mode of FDIR hash information in RX descriptors.

Enumerator:

RTE_FDIR_NO_REPORT_STATUS Never report FDIR hash.

RTE_FDIR_REPORT_STATUS Only report FDIR hash for matching pkts.

RTE_FDIR_REPORT_STATUS_ALWAYS Always report FDIR hash.

3.11.4.9 enum rte_l4type

Possible l4type of FDIR filters.

Enumerator:

RTE_FDIR_L4TYPE_NONE None.

RTE_FDIR_L4TYPE_UDP UDP.

RTE_FDIR_L4TYPE_TCP TCP.

RTE_FDIR_L4TYPE_SCTP SCTP.

3.11.4.10 enum rte_iptype

Select IPv4 or IPv6 FDIR filters.

Enumerator:

RTE_FDIR_IPTYPE_IPV4 IPv4.

RTE_FDIR_IPTYPE_IPV6 IPv6.

3.11.4.11 enum rte_eth_event_type

The eth device event type for interrupt, and maybe others in the future.

Enumerator:

RTE_ETH_EVENT_UNKNOWN unknown event type

RTE_ETH_EVENT_INTR_LSC lsc interrupt event

RTE_ETH_EVENT_MAX max value of this enum

3.11.5 Function Documentation

3.11.5.1 uint8_t rte_eth_dev_count (void)

Get the total number of Ethernet devices that have been successfully initialized by the [matching] Ethernet driver during the PCI probing phase. All devices whose port identifier is in the range [0, rte_eth_dev_count() - 1] can be operated on by network applications.

Returns

- The total number of usable Ethernet devices.

3.11.5.2 struct rte_eth_dev* rte_eth_dev_allocate (void) [read]

Function for internal use by dummy drivers primarily, e.g. ring-based driver. Allocates a new ethdev slot for an ethernet device and returns the pointer to that slot for the driver to use.

Returns

- Slot in the rte_dev_devices array for a new device;

3.11.5.3 int rte_igb_pmd_init (void)

The initialization function of the driver for Intel(r) IGB Gigabit Ethernet Controller devices. This function is invoked once at EAL start time.

Returns

0 on success

3.11.5.4 int rte_em_pmd_init (void)

The initialization function of the driver for Intel(r) EM Gigabit Ethernet Controller devices. This function is invoked once at EAL start time.

Returns

0 on success

3.11.5.5 int rte_igbvf_pmd_init (void)

The initialization function of the driver for 1Gbps Intel IGB_VF Ethernet devices. Invoked once at EAL start time.

Returns

0 on success

3.11.5.6 int rte_ixgbe_pmd_init (void)

The initialization function of the driver for 10Gbps Intel IXGBE Ethernet devices. Invoked once at EAL start time.

Returns

0 on success

3.11.5.7 int rte_ixgbevf_pmd_init (void)

The initialization function of the driver for 10Gbps Intel IXGBE_VF Ethernet devices. Invoked once at EAL start time.

Returns

0 on success

3.11.5.8 int rte_virtio_pmd_init (void)

The initialization function of the driver for Qumranet virtio-net Ethernet devices. Invoked once at EAL start time.

Returns

0 on success

3.11.5.9 int rte_vmxnet3_pmd_init (void)

The initialization function of the driver for VMware VMXNET3 Ethernet devices. Invoked once at EAL start time.

Returns

0 on success

3.11.5.10 static int rte_pmd_init_all (void) [static]

The initialization function of ∗all∗ supported and enabled drivers. Right now, the following PMDs are supported:

- igb
- igbvf
- em
- ixgbe
- ixgbevf
- virtio
- vmxnet3 This function is invoked once at EAL start time.

Returns

0 on success. Error code of the device initialization failure, -ENODEV if there are no drivers available (e.g. if all driver config options are = n).

3.11.5.11 int rte_eth_dev_configure (uint8_t *port_id*, uint16_t *nb_rx_queue*, uint16_t *nb_tx_queue*, const struct rte_eth_conf ∗ *eth_conf*)

Configure an Ethernet device. This function must be invoked first before any other function in the Ethernet API. This function can also be re-invoked when a device is in the stopped state.

Parameters

port_id	The port identifier of the Ethernet device to configure.
nb_rx_queue	The number of receive queues to set up for the Ethernet device.
nb_tx_queue	The number of transmit queues to set up for the Ethernet device.
eth_conf	The pointer to the configuration data to be used for the Ethernet device. The ∗rte_eth_conf∗ structure includes: • the hardware offload features to activate, with dedicated fields for each statically configurable offload hardware feature provided by Ethernet devices, such as IP checksum or VLAN tag stripping for example. • the Receive Side Scaling (RSS) configuration when using multiple RX queues per port.

Embedding all configuration information in a single data structure is the more flexible method that allows the addition of new features without changing the syntax of the API.

Returns

- 0: Success, device configured.
- <0: Error code returned by the driver configuration function.

3.11.5.12 int rte_eth_rx_queue_setup (uint8_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*)

Allocate and set up a receive queue for an Ethernet device.

The function allocates a contiguous block of memory for *nb_rx_desc* receive descriptors from a memory zone associated with *socket_id* and initializes each receive descriptor with a network buffer allocated from the memory pool *mb_pool*.

Parameters

port_id	The port identifier of the Ethernet device.
rx_queue_id	The index of the receive queue to set up. The value must be in the range [0, nb_rx_queue - 1] previously supplied to rte_eth_dev_configure().
nb_rx_desc	The number of receive descriptors to allocate for the receive ring.
socket_id	The *socket_id* argument is the socket identifier in case of NUMA. The value can be *SOCKET_ID_ANY* if there is no NUMA constraint for the DMA memory allocated for the receive descriptors of the ring.
rx_conf	The pointer to the configuration data to be used for the receive queue. The *rx_conf* structure contains an *rx_thresh* structure with the values of the Prefetch, Host, and Write-Back threshold registers of the receive ring.
mb_pool	The pointer to the memory pool from which to allocate *rte_mbuf* network memory buffers to populate each descriptor of the receive ring.

Returns

- 0: Success, receive queue correctly set up.
- -EINVAL: The size of network buffers which can be allocated from the memory pool does not fit the various buffer sizes allowed by the device controller.
- -ENOMEM: Unable to allocate the receive ring descriptors or to allocate network memory buffers from the memory pool when initializing receive descriptors.

3.11.5.13 int rte_eth_tx_queue_setup (uint8_t *port_id*, uint16_t *tx_queue_id*, uint16_t *nb_tx_desc*, unsigned int *socket_id*, const struct rte_eth_txconf * *tx_conf*)

Allocate and set up a transmit queue for an Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.
tx_queue_id	The index of the transmit queue to set up. The value must be in the range [0, nb_tx_queue - 1] previously supplied to rte_eth_dev_configure().
nb_tx_desc	The number of transmit descriptors to allocate for the transmit ring.
socket_id	The ∗socket_id∗ argument is the socket identifier in case of NUMA. Its value can be ∗SOCKET_ID_ANY∗ if there is no NUMA constraint for the DMA memory allocated for the transmit descriptors of the ring.
tx_conf	The pointer to the configuration data to be used for the transmit queue. The ∗tx_conf∗ structure contains the following data: • The ∗tx_thresh∗ structure with the values of the Prefetch, Host, and Write-Back threshold registers of the transmit ring. When setting Write-Back threshold to the value greater then zero, ∗tx_rs_thresh∗ value should be explicitly set to one. • The ∗tx_free_thresh∗ value indicates the [minimum] number of network buffers that must be pending in the transmit ring to trigger their [implicit] freeing by the driver transmit function. • The ∗tx_rs_thresh∗ value indicates the [minimum] number of transmit descriptors that must be pending in the transmit ring before setting the RS bit on a descriptor by the driver transmit function. The ∗tx_rs_thresh∗ value should be less or equal then ∗tx_free_thresh∗ value, and both of them should be less then ∗nb_tx_desc∗ - 3. • The ∗txq_flags∗ member contains flags to pass to the TX queue setup function to configure the behavior of the TX queue. This should be set to 0 if no special configuration is required.

Note that setting ∗tx_free_thresh∗ or ∗tx_rs_thresh∗ value to 0 forces the transmit function to use default values.

Returns

- 0: Success, the transmit queue is correctly set up.
- -ENOMEM: Unable to allocate the transmit ring descriptors.

3.11.5.14 int rte_eth_dev_start (uint8_t *port_id*)

Start an Ethernet device.

The device start step is the last one and consists of setting the configured offload features and in starting the transmit and the receive units of the device. On success, all basic functions exported by the Ethernet API (link status, receive/transmit, and so on) can be invoked.

Parameters

port_id	The port identifier of the Ethernet device.

Returns

- 0: Success, Ethernet device started.
- <0: Error code of the driver device start function.

3.11.5.15 void rte_eth_dev_stop (uint8_t *port_id*)

Stop an Ethernet device. The device can be restarted with a call to rte_eth_dev_start()

Parameters

port_id	The port identifier of the Ethernet device.

3.11.5.16 void rte_eth_dev_close (uint8_t *port_id*)

Close an Ethernet device. The device cannot be restarted!

Parameters

port_id	The port identifier of the Ethernet device.

3.11.5.17 void rte_eth_promiscuous_enable (uint8_t *port_id*)

Enable receipt in promiscuous mode for an Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.

3.11.5.18 void rte_eth_promiscuous_disable (uint8_t *port_id*)

Disable receipt in promiscuous mode for an Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.

3.11.5.19 int rte_eth_promiscuous_get (uint8_t *port_id*)

Return the value of promiscuous mode for an Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.

Returns

- (1) if promiscuous is enabled
- (0) if promiscuous is disabled.
- (-1) on error

3.11.5.20 void rte_eth_allmulticast_enable (uint8_t *port_id*)

Enable the receipt of any multicast frame by an Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.

3.11.5.21 void rte_eth_allmulticast_disable (uint8_t *port_id*)

Disable the receipt of all multicast frames by an Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.

3.11.5.22 int rte_eth_allmulticast_get (uint8_t *port_id*)

Return the value of allmulticast mode for an Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.

Returns

- (1) if allmulticast is enabled
- (0) if allmulticast is disabled.
- (-1) on error

3.11.5.23 void rte_eth_link_get (uint8_t *port_id*, struct rte_eth_link * *link*)

Retrieve the status (ON/OFF), the speed (in Mbps) and the mode (HALF-DUPLEX or FULL-DUPLEX) of the physical link of an Ethernet device. It might need to wait up to 9 seconds in it.

Parameters

port_id	The port identifier of the Ethernet device.
link	A pointer to an *rte_eth_link* structure to be filled with the status, the speed and the mode of the Ethernet device link.

3.11.5.24 void rte_eth_link_get_nowait (uint8_t *port_id*, struct rte_eth_link * *link*)

Retrieve the status (ON/OFF), the speed (in Mbps) and the mode (HALF-DUPLEX or FULL-DUPLEX) of the physical link of an Ethernet device. It is a no-wait version of rte_eth_link_get().

Parameters

port_id	The port identifier of the Ethernet device.
link	A pointer to an *rte_eth_link* structure to be filled with the status, the speed and the mode of the Ethernet device link.

3.11.5.25 void rte_eth_stats_get (uint8_t *port_id*, struct rte_eth_stats * *stats*)

Retrieve the general I/O statistics of an Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.
stats	A pointer to a structure of type *rte_eth_stats* to be filled with the values of device counters for the following set of statistics: • *ipackets* with the total of successfully received packets. • *opackets* with the total of successfully transmitted packets. • *ibytes* with the total of successfully received bytes. • *obytes* with the total of successfully transmitted bytes. • *ierrors* with the total of erroneous received packets. • *oerrors* with the total of failed transmitted packets.

3.11.5.26 void rte_eth_stats_reset (uint8_t *port_id*)

Reset the general I/O statistics of an Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.

3.11.5.27 int rte_eth_dev_set_tx_queue_stats_mapping (uint8_t *port_id*, uint16_t *tx_queue_id*, uint8_t *stat_idx*)

Set a mapping for the specified transmit queue to the specified per-queue statistics counter.

Parameters

port_id	The port identifier of the Ethernet device.
tx_queue_id	The index of the transmit queue for which a queue stats mapping is required. The value must be in the range [0, nb_tx_queue - 1] previously supplied to rte_eth_dev_configure().
stat_idx	The per-queue packet statistics functionality number that the transmit queue is to be assigned. The value must be in the range [0, RTE_MAX_ETHPORT_QUEUE_STATS_MAPS - 1].

Returns

Zero if successful. Non-zero otherwise.

3.11.5.28 int rte_eth_dev_set_rx_queue_stats_mapping (uint8_t *port_id*, uint16_t *rx_queue_id*, uint8_t *stat_idx*)

Set a mapping for the specified receive queue to the specified per-queue statistics counter.

Parameters

port_id	The port identifier of the Ethernet device.
rx_queue_id	The index of the receive queue for which a queue stats mapping is required. The value must be in the range [0, nb_rx_queue - 1] previously supplied to rte_eth_dev_configure().
stat_idx	The per-queue packet statistics functionality number that the receive queue is to be assigned. The value must be in the range [0, RTE_MAX_ETHPORT_QUEUE_STATS_MAPS - 1].

Returns

Zero if successful. Non-zero otherwise.

3.11.5.29 void rte_eth_macaddr_get (uint8_t *port_id*, struct ether_addr * *mac_addr*)

Retrieve the Ethernet address of an Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.
mac_addr	A pointer to a structure of type ∗ether_addr∗ to be filled with the Ethernet address of the Ethernet device.

3.11.5.30 void rte_eth_dev_info_get (uint8_t *port_id*, struct rte_eth_dev_info ∗ *dev_info*)

Retrieve the contextual information of an Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.
dev_info	A pointer to a structure of type ∗rte_eth_dev_info∗ to be filled with the contextual information of the Ethernet device.

3.11.5.31 int rte_eth_dev_vlan_filter (uint8_t *port_id*, uint16_t *vlan_id*, int *on*)

Enable/Disable hardware filtering by an Ethernet device of received VLAN packets tagged with a given VLAN Tag Identifier.

Parameters

port_id	The port identifier of the Ethernet device.
vlan_id	The VLAN Tag Identifier whose filtering must be enabled or disabled.
on	If > 0, enable VLAN filtering of VLAN packets tagged with ∗vlan_id∗. Otherwise, disable VLAN filtering of VLAN packets tagged with ∗vlan_id∗.

Returns

- (0) if successful.
- (-ENOSUP) if hardware-assisted VLAN filtering not configured.
- (-ENODEV) if ∗port_id∗ invalid.
- (-ENOSYS) if VLAN filtering on ∗port_id∗ disabled.
- (-EINVAL) if ∗vlan_id∗ > 4095.

3.11.5.32 int rte_eth_dev_set_vlan_strip_on_queue (uint8_t *port_id*, uint16_t *rx_queue_id*, int *on*)

Enable/Disable hardware VLAN Strip by a rx queue of an Ethernet device. 82599/X540 can support VLAN stripping at the rx queue level

Parameters

port_id	The port identifier of the Ethernet device.
rx_queue_id	The index of the receive queue for which a queue stats mapping is required. The value must be in the range [0, nb_rx_queue - 1] previously supplied to rte_eth_dev_configure().
on	If 1, Enable VLAN Stripping of the receive queue of the Ethernet port. If 0, Disable VLAN Stripping of the receive queue of the Ethernet port.

Returns

- (0) if successful.
- (-ENOSUP) if hardware-assisted VLAN stripping not configured.
- (-ENODEV) if ∗port_id∗ invalid.
- (-EINVAL) if ∗rx_queue_id∗ invalid.

3.11.5.33 int rte_eth_dev_set_vlan_ether_type (uint8_t *port_id*, uint16_t *tag_type*)

Set the Outer VLAN Ether Type by an Ethernet device, it can be inserted to the VLAN Header. This is a register setup available on some Intel NIC, not but all, please check the data sheet for availability.

Parameters

port_id	The port identifier of the Ethernet device.
tag_type	The Tag Protocol ID

Returns

- (0) if successful.
- (-ENOSUP) if hardware-assisted VLAN TPID setup is not supported.
- (-ENODEV) if ∗port_id∗ invalid.

3.11.5.34 int rte_eth_dev_set_vlan_offload (uint8_t *port_id*, int *offload_mask*)

Set VLAN offload configuration on an Ethernet device Enable/Disable Extended VLAN by an Ethernet device, This is a register setup available on some Intel NIC, not but all, please check the data sheet for availability. Enable/Disable VLAN Strip can be done on rx queue for certain NIC, but here the configuration is applied on the port level.

Parameters

port_id	The port identifier of the Ethernet device.
offload_mask	The VLAN Offload bit mask can be mixed use with "OR" ETH_VLAN_STRIP_OFFLOAD ETH_VLAN_FILTER_OFFLOAD ETH_VLAN_EXTEND_OFFLOAD

Returns

- (0) if successful.
- (-ENOSUP) if hardware-assisted VLAN filtering not configured.
- (-ENODEV) if ∗port_id∗ invalid.

3.11.5.35 int rte_eth_dev_get_vlan_offload (uint8_t *port_id*)

Read VLAN Offload configuration from an Ethernet device

Parameters

port_id	The port identifier of the Ethernet device.

Returns

- (>0) if successful. Bit mask to indicate ETH_VLAN_STRIP_OFFLOAD ETH_VLAN_FILTER_OFFLOAD ETH_VLAN_EXTEND_OFFLOAD
- (-ENODEV) if ∗port_id∗ invalid.

3.11.5.36 static uint16_t rte_eth_rx_burst (uint8_t *port_id*, uint16_t *queue_id*, struct rte_mbuf ∗∗ *rx_pkts*, uint16_t *nb_pkts*) [static]

Retrieve a burst of input packets from a receive queue of an Ethernet device. The retrieved packets are stored in ∗rte_mbuf∗ structures whose pointers are supplied in the ∗rx_pkts∗ array.

The rte_eth_rx_burst() function loops, parsing the RX ring of the receive queue, up to ∗nb_pkts∗ packets, and for each completed RX descriptor in the ring, it performs the following operations:

- Initialize the ∗rte_mbuf∗ data structure associated with the RX descriptor according to the information provided by the NIC into that RX descriptor.
- Store the ∗rte_mbuf∗ data structure into the next entry of the ∗rx_pkts∗ array.
- Replenish the RX descriptor with a new ∗rte_mbuf∗ buffer allocated from the memory pool associated with the receive queue at initialization time.

When retrieving an input packet that was scattered by the controller into multiple receive descriptors, the rte_eth_rx_burst() function appends the associated ∗rte_mbuf∗ buffers to the first buffer of the packet.

The rte_eth_rx_burst() function returns the number of packets actually retrieved, which is the number of ∗rte_mbuf∗ data structures effectively supplied into the ∗rx_pkts∗ array. A return value equal to ∗nb_pkts∗ indicates that the RX queue contained at least ∗rx_pkts∗ packets, and this is likely to signify that other received packets remain in the input queue. Applications implementing a "retrieve as much received packets as possible" policy can check this specific case and keep invoking the rte_eth_rx_burst() function until a value less than ∗nb_pkts∗ is returned.

This receive method has the following advantages:

- It allows a run-to-completion network stack engine to retrieve and to immediately process received packets in a fast burst-oriented approach, avoiding the overhead of unnecessary intermediate packet queue/dequeue operations.

- Conversely, it also allows an asynchronous-oriented processing method to retrieve bursts of received packets and to immediately queue them for further parallel processing by another logical core, for instance. However, instead of having received packets being individually queued by the driver, this approach allows the invoker of the rte_eth_rx_burst() function to queue a burst of retrieved packets at a time and therefore dramatically reduce the cost of enqueue/dequeue operations per packet.

- It allows the rte_eth_rx_burst() function of the driver to take advantage of burst-oriented hardware features (CPU cache, prefetch instructions, and so on) to minimize the number of CPU cycles per packet.

To summarize, the proposed receive API enables many burst-oriented optimizations in both synchronous and asynchronous packet processing environments with no overhead in both cases.

The rte_eth_rx_burst() function does not provide any error notification to avoid the corresponding overhead. As a hint, the upper-level application might check the status of the device link once being systematically returned a 0 value for a given number of tries.

Parameters

port_id	The port identifier of the Ethernet device.
queue_id	The index of the receive queue from which to retrieve input packets. The value must be in the range [0, nb_rx_queue - 1] previously supplied to rte_eth_dev_configure().
rx_pkts	The address of an array of pointers to ∗rte_mbuf∗ structures that must be large enough to store ∗nb_pkts∗ pointers in it.
nb_pkts	The maximum number of packets to retrieve.

Returns

The number of packets actually retrieved, which is the number of pointers to ∗rte_mbuf∗ structures effectively supplied to the ∗rx_pkts∗ array.

3.11.5.37 static uint32_t rte_eth_rx_queue_count (uint8_t *port_id*, uint16_t *queue_id*) [static]

Get the number of used descriptors in a specific queue

Parameters

port_id	The port identifier of the Ethernet device.
queue_id	The queue id on the specific port.

Returns

The number of used descriptors in the specific queue.

3.11.5.38 static int rte_eth_rx_descriptor_done (uint8_t *port_id*, uint16_t *queue_id*, uint16_t *offset*) [static]

Check if the DD bit of the specific RX descriptor in the queue has been set

Parameters

port_id	The port identifier of the Ethernet device.
queue_id	The queue id on the specific port. The offset of the descriptor ID from tail.

Returns

- (1) if the specific DD bit is set.
- (0) if the specific DD bit is not set.
- (-ENODEV) if *port_id* invalid.

3.11.5.39 static uint16_t rte_eth_tx_burst (uint8_t *port_id*, uint16_t *queue_id*, struct rte_mbuf ** *tx_pkts*, uint16_t *nb_pkts*) [static]

Send a burst of output packets on a transmit queue of an Ethernet device.

The rte_eth_tx_burst() function is invoked to transmit output packets on the output queue *queue_id* of the Ethernet device designated by its *port_id*. The *nb_pkts* parameter is the number of packets to send which are supplied in the *tx_pkts* array of *rte_mbuf* structures. The rte_eth_tx_burst() function loops, sending *nb_pkts* packets, up to the number of transmit descriptors available in the TX ring of the transmit queue. For each packet to send, the rte_eth_tx_burst() function performs the following operations:

- Pick up the next available descriptor in the transmit ring.
- Free the network buffer previously sent with that descriptor, if any.
- Initialize the transmit descriptor with the information provided in the *rte_mbuf data structure.

In the case of a segmented packet composed of a list of *rte_mbuf* buffers, the rte_eth_tx_burst() function uses several transmit descriptors of the ring.

The rte_eth_tx_burst() function returns the number of packets it actually sent. A return value equal to *nb-_pkts* means that all packets have been sent, and this is likely to signify that other output packets could be immediately transmitted again. Applications that implement a "send as many packets to transmit as possible" policy can check this specific case and keep invoking the rte_eth_tx_burst() function until a value less than *nb_pkts* is returned.

It is the responsibility of the rte_eth_tx_burst() function to transparently free the memory buffers of packets previously sent. This feature is driven by the *tx_free_thresh* value supplied to the rte_eth_dev_configure() function at device configuration time. When the number of previously sent packets reached the "minimum transmit packets to free" threshold, the rte_eth_tx_burst() function must [attempt to] free the *rte_mbuf* buffers of those packets whose transmission was effectively completed.

Parameters

port_id	The port identifier of the Ethernet device.
queue_id	The index of the transmit queue through which output packets must be sent. The value must be in the range [0, nb_tx_queue - 1] previously supplied to rte_eth_dev_configure().
tx_pkts	The address of an array of ∗nb_pkts∗ pointers to ∗rte_mbuf∗ structures which contain the output packets.
nb_pkts	The maximum number of packets to transmit.

Returns

The number of output packets actually stored in transmit descriptors of the transmit ring. The return value can be less than the value of the ∗tx_pkts∗ parameter when the transmit ring is full or has been filled up.

3.11.5.40 int rte_eth_dev_fdir_add_signature_filter (uint8_t *port_id*, struct rte_fdir_filter ∗ *fdir_filter*, uint8_t *rx_queue*)

Setup a new signature filter rule on an Ethernet device

Parameters

port_id	The port identifier of the Ethernet device.
fdir_filter	The pointer to the fdir filter structure describing the signature filter rule. The ∗rte_fdir_filter∗ structure includes the values of the different fields to match: source and destination IP addresses, vlan id, flexbytes, source and destination ports, and so on.
rx_queue	The index of the RX queue where to store RX packets matching the added signature filter defined in fdir_filter.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support flow director mode.
- (-ENODEV) if ∗port_id∗ invalid.
- (-ENOSYS) if the FDIR mode is not configured in signature mode on ∗port_id∗.
- (-EINVAL) if the fdir_filter information is not correct.

3.11.5.41 int rte_eth_dev_fdir_update_signature_filter (uint8_t *port_id*, struct rte_fdir_filter ∗ *fdir_ftr*, uint8_t *rx_queue*)

Update a signature filter rule on an Ethernet device. If the rule doesn't exits, it is created.

Parameters

port_id	The port identifier of the Ethernet device.
fdir_ftr	The pointer to the structure describing the signature filter rule. The *rte_fdir_filter* structure includes the values of the different fields to match: source and destination IP addresses, vlan id, flexbytes, source and destination ports, and so on.
rx_queue	The index of the RX queue where to store RX packets matching the added signature filter defined in fdir_ftr.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support flow director mode.
- (-ENODEV) if *port_id* invalid.
- (-ENOSYS) if the flow director mode is not configured in signature mode on *port_id*.
- (-EINVAL) if the fdir_filter information is not correct.

3.11.5.42 int rte_eth_dev_fdir_remove_signature_filter (uint8_t *port_id*, struct rte_fdir_filter * *fdir_ftr*)

Remove a signature filter rule on an Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.
fdir_ftr	The pointer to the structure describing the signature filter rule. The *rte_fdir_filter* structure includes the values of the different fields to match: source and destination IP addresses, vlan id, flexbytes, source and destination ports, and so on.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support flow director mode.
- (-ENODEV) if *port_id* invalid.
- (-ENOSYS) if the flow director mode is not configured in signature mode on *port_id*.
- (-EINVAL) if the fdir_filter information is not correct.

3.11.5.43 int rte_eth_dev_fdir_get_infos (uint8_t *port_id*, struct rte_eth_fdir * *fdir*)

Retrieve the flow director information of an Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.
fdir	A pointer to a structure of type *rte_eth_dev_fdir* to be filled with the flow director information of the Ethernet device.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support flow director mode.
- (-ENODEV) if ∗port_id∗ invalid.
- (-ENOSYS) if the flow director mode is not configured on ∗port_id∗.

3.11.5.44 int rte_eth_dev_fdir_add_perfect_filter (uint8_t *port_id*, struct rte_fdir_filter ∗ *fdir_filter*, uint16_t *soft_id*, uint8_t *rx_queue*, uint8_t *drop*)

Add a new perfect filter rule on an Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.
fdir_filter	The pointer to the structure describing the perfect filter rule. The ∗rte_fdir_filter∗ structure includes the values of the different fields to match: source and destination IP addresses, vlan id, flexbytes, source and destination ports, and so on. IPv6 are not supported.
soft_id	The 16-bit value supplied in the field hash.fdir.id of mbuf for RX packets matching the perfect filter.
rx_queue	The index of the RX queue where to store RX packets matching the added perfect filter defined in fdir_filter.
drop	If drop is set to 1, matching RX packets are stored into the RX drop queue defined in the rte_fdir_conf.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support flow director mode.
- (-ENODEV) if ∗port_id∗ invalid.
- (-ENOSYS) if the flow director mode is not configured in perfect mode on ∗port_id∗.
- (-EINVAL) if the fdir_filter information is not correct.

3.11.5.45 int rte_eth_dev_fdir_update_perfect_filter (uint8_t *port_id*, struct rte_fdir_filter ∗ *fdir_filter*, uint16_t *soft_id*, uint8_t *rx_queue*, uint8_t *drop*)

Update a perfect filter rule on an Ethernet device. If the rule doesn't exits, it is created.

Parameters

port_id	The port identifier of the Ethernet device.
fdir_filter	The pointer to the structure describing the perfect filter rule. The ∗rte_fdir_filter∗ structure includes the values of the different fields to match: source and destination IP addresses, vlan id, flexbytes, source and destination ports, and so on. IPv6 are not supported.

soft_id	The 16-bit value supplied in the field hash.fdir.id of mbuf for RX packets matching the perfect filter.
rx_queue	The index of the RX queue where to store RX packets matching the added perfect filter defined in fdir_filter.
drop	If drop is set to 1, matching RX packets are stored into the RX drop queue defined in the rte_fdir_conf.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support flow director mode.
- (-ENODEV) if ∗port_id∗ invalid.
- (-ENOSYS) if the flow director mode is not configured in perfect mode on ∗port_id∗.
- (-EINVAL) if the fdir_filter information is not correct.

3.11.5.46 int rte_eth_dev_fdir_remove_perfect_filter (uint8_t *port_id,* struct rte_fdir_filter ∗ *fdir_filter,* uint16_t *soft_id*)

Remove a perfect filter rule on an Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.
fdir_filter	The pointer to the structure describing the perfect filter rule. The ∗rte_fdir_filter∗ structure includes the values of the different fields to match: source and destination IP addresses, vlan id, flexbytes, source and destination ports, and so on. IPv6 are not supported.
soft_id	The soft_id value provided when adding/updating the removed filter.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support flow director mode.
- (-ENODEV) if ∗port_id∗ invalid.
- (-ENOSYS) if the flow director mode is not configured in perfect mode on ∗port_id∗.
- (-EINVAL) if the fdir_filter information is not correct.

3.11.5.47 int rte_eth_dev_fdir_set_masks (uint8_t *port_id,* struct rte_fdir_masks ∗ *fdir_mask*)

Configure globally the masks for flow director mode for an Ethernet device. For example, the device can match packets with only the first 24 bits of the IPv4 source address.

The following fields can be masked: IPv4 addresses and L4 port numbers. The following fields can be either enabled or disabled completely for the matching functionality: VLAN ID tag; VLAN Priority + CFI bit; Flexible 2-byte tuple. IPv6 masks are not supported.

All filters must comply with the masks previously configured. For example, with a mask equal to 255.255.-255.0 for the source IPv4 address, all IPv4 filters must be created with a source IPv4 address that fits the "X.X.X.0" format.

This function flushes all filters that have been previously added in the device.

Parameters

port_id	The port identifier of the Ethernet device.
fdir_mask	The pointer to the fdir mask structure describing relevant headers fields and relevant bits to use when matching packets addresses and ports. IPv6 masks are not supported.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support flow director mode.
- (-ENODEV) if ∗port_id∗ invalid.
- (-ENOSYS) if the flow director mode is not configured in perfect mode on ∗port_id∗.
- (-EINVAL) if the fdir_filter information is not correct

3.11.5.48 int rte_eth_dev_callback_register (uint8_t *port_id*, enum rte_eth_event_type *event*, rte_eth_dev_cb_fn *cb_fn*, void ∗ *cb_arg*)

Register a callback function for specific port id.

Parameters

port_id	Port id.
event	Event interested.
cb_fn	User supplied callback function to be called.
cb_arg	Pointer to the parameters for the registered callback.

Returns

- On success, zero.
- On failure, a negative value.

3.11.5.49 int rte_eth_dev_callback_unregister (uint8_t *port_id*, enum rte_eth_event_type *event*, rte_eth_dev_cb_fn *cb_fn*, void ∗ *cb_arg*)

Unregister a callback function for specific port id.

Parameters

port_id	Port id.
event	Event interested.
cb_fn	User supplied callback function to be called.
cb_arg	Pointer to the parameters for the registered callback. -1 means to remove all for the same callback address and same event.

Returns

- On success, zero.
- On failure, a negative value.

3.11.5.50 int rte_eth_led_on (uint8_t *port_id*)

Turn on the LED on the Ethernet device. This function turns on the LED on the Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.

Returns

- (0) if successful.
- (-ENOTSUP) if underlying hardware OR driver doesn't support that operation.
- (-ENODEV) if ∗port_id∗ invalid.

3.11.5.51 int rte_eth_led_off (uint8_t *port_id*)

Turn off the LED on the Ethernet device. This function turns off the LED on the Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.

Returns

- (0) if successful.
- (-ENOTSUP) if underlying hardware OR driver doesn't support that operation.
- (-ENODEV) if ∗port_id∗ invalid.

3.11.5.52 int rte_eth_dev_flow_ctrl_set (uint8_t *port_id*, struct rte_eth_fc_conf ∗ *fc_conf*)

Configure the Ethernet link flow control for Ethernet device

Parameters

port_id	The port identifier of the Ethernet device.
fc_conf	The pointer to the structure of the flow control parameters.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support flow control mode.
- (-ENODEV) if ∗port_id∗ invalid.
- (-EINVAL) if bad parameter
- (-EIO) if flow control setup failure

3.11.5.53 int rte_eth_dev_priority_flow_ctrl_set (uint8_t *port_id*, struct rte_eth_pfc_conf ∗ *pfc_conf*)

Configure the Ethernet priority flow control under DCB environment for Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.
pfc_conf	The pointer to the structure of the priority flow control parameters.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support priority flow control mode.
- (-ENODEV) if ∗port_id∗ invalid.
- (-EINVAL) if bad parameter
- (-EIO) if flow control setup failure

3.11.5.54 int rte_eth_dev_mac_addr_add (uint8_t *port*, struct ether_addr ∗ *mac_addr*, uint32_t *pool*)

Add a MAC address to an internal array of addresses used to enable whitelist filtering to accept packets only if the destination MAC address matches.

Parameters

port	The port identifier of the Ethernet device.
mac_addr	The MAC address to add.
pool	VMDq pool index to associate address with (if VMDq is enabled). If VMDq is not enabled, this should be set to 0.

Returns

- (0) if successfully added or ∗mac_addr" was already added.
- (-ENOTSUP) if hardware doesn't support this feature.
- (-ENODEV) if ∗port∗ is invalid.
- (-ENOSPC) if no more MAC addresses can be added.
- (-EINVAL) if MAC address is invalid.

3.11.5.55 int rte_eth_dev_mac_addr_remove (uint8_t *port*, struct ether_addr ∗ *mac_addr*)

Remove a MAC address from the internal array of addresses.

Parameters

port	The port identifier of the Ethernet device.
mac_addr	MAC address to remove.

Returns

- (0) if successful, or ∗mac_addr∗ didn't exist.
- (-ENOTSUP) if hardware doesn't support.
- (-ENODEV) if ∗port∗ invalid.
- (-EADDRINUSE) if attempting to remove the default MAC address

3.11.5.56 int rte_eth_dev_rss_reta_update (uint8_t *port*, struct rte_eth_rss_reta ∗ *reta_conf*)

Update Redirection Table(RETA) of Receive Side Scaling of Ethernet device.

Parameters

port	The port identifier of the Ethernet device.
reta_conf	RETA to update.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support.
- (-EINVAL) if bad parameter.

3.11.5.57 int rte_eth_dev_rss_reta_query (uint8_t *port*, struct rte_eth_rss_reta ∗ *reta_conf*)

Query Redirection Table(RETA) of Receive Side Scaling of Ethernet device.

Parameters

port	The port identifier of the Ethernet device.
reta_conf	RETA to query.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support.
- (-EINVAL) if bad parameter.

3.11.5.58 int rte_eth_dev_uc_hash_table_set (uint8_t *port*, struct ether_addr * *addr*, uint8_t *on*)

Updates unicast hash table for receiving packet with the given destination MAC address, and the packet is routed to all VFs for which the RX mode is accept packets that match the unicast hash table.

Parameters

port	The port identifier of the Ethernet device.
addr	Unicast MAC address.
on	1 - Set an unicast hash bit for receiving packets with the MAC address. 0 - Clear an unicast hash bit.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support.

- (-ENODEV) if *port_id* invalid.
  - (-EINVAL) if bad parameter.

3.11.5.59 int rte_eth_dev_uc_all_hash_table_set (uint8_t *port*, uint8_t *on*)

Updates all unicast hash bitmaps for receiving packet with any Unicast Ethernet MAC addresses,the packet is routed to all VFs for which the RX mode is accept packets that match the unicast hash table.

Parameters

port	The port identifier of the Ethernet device.
on	1 - Set all unicast hash bitmaps for receiving all the Ethernet MAC addresses 0 - Clear all unicast hash bitmaps

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support.

- (-ENODEV) if ∗port_id∗ invalid.
  - (-EINVAL) if bad parameter.

3.11.5.60 int rte_eth_dev_set_vf_rxmode (uint8_t *port*, uint16_t *vf*, uint16_t *rx_mode*, uint8_t *on*)

Set RX L2 Filtering mode of a VF of an Ethernet device.

Parameters

port	The port identifier of the Ethernet device.
vf	VF id.
rx_mode	The RX mode mask, which is one or more of accepting Untagged Packets, packets that match the PFUTA table, Broadcast and Multicast Promiscuous. ETH_VMDQ_ACCEPT_UNTAG,ETH_VMDQ_ACCEPT_HASH_UC, ETH_VMDQ_ACCEPT_BROADCAST and ETH_VMDQ_ACCEPT_MULTICAST will be used in rx_mode.
on	1 - Enable a VF RX mode. 0 - Disable a VF RX mode.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support.
- (-ENOTSUP) if hardware doesn't support.
- (-EINVAL) if bad parameter.

3.11.5.61 int rte_eth_dev_set_vf_tx (uint8_t *port*, uint16_t *vf*, uint8_t *on*)

Enable or disable a VF traffic transmit of the Ethernet device.

Parameters

port	The port identifier of the Ethernet device.
vf	VF id.
on	1 - Enable a VF traffic transmit. 0 - Disable a VF traffic transmit.

Returns

- (0) if successful.
- (-ENODEV) if ∗port_id∗ invalid.
- (-ENOTSUP) if hardware doesn't support.

- (-EINVAL) if bad parameter.

3.11.5.62 int rte_eth_dev_set_vf_rx (uint8_t *port*, uint16_t *vf*, uint8_t *on*)

Enable or disable a VF traffic receive of an Ethernet device.

Parameters

port	The port identifier of the Ethernet device.
vf	VF id.
on	1 - Enable a VF traffic receive. 0 - Disable a VF traffic receive.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support.
- (-ENODEV) if ∗port_id∗ invalid.
- (-EINVAL) if bad parameter.

3.11.5.63 int rte_eth_dev_set_vf_vlan_filter (uint8_t *port*, uint16_t *vlan_id*, uint64_t *vf_mask*, uint8_t *vlan_on*)

Enable/Disable hardware VF VLAN filtering by an Ethernet device of received VLAN packets tagged with a given VLAN Tag Identifier.

Parameters

port	id The port identifier of the Ethernet device.
vlan_id	The VLAN Tag Identifier whose filtering must be enabled or disabled.
vf_mask	Bitmap listing which VFs participate in the VLAN filtering.
vlan_on	1 - Enable VFs VLAN filtering. 0 - Disable VFs VLAN filtering.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support.
- (-ENODEV) if ∗port_id∗ invalid.
- (-EINVAL) if bad parameter.

3.11.5.64 int rte_eth_mirror_rule_set (uint8_t *port_id*, struct rte_eth_vmdq_mirror_conf ∗ *mirror_conf*, uint8_t *rule_id*, uint8_t *on*)

Set a traffic mirroring rule on an Ethernet device

Parameters

port_id	The port identifier of the Ethernet device.
mirror_conf	The pointer to the traffic mirroring structure describing the mirroring rule. The ∗rte_eth-_vm_mirror_conf∗ structure includes the type of mirroring rule, destination pool and the value of rule if enable vlan or pool mirroring.
rule_id	The index of traffic mirroring rule, we support four separated rules.
on	1 - Enable a mirroring rule. 0 - Disable a mirroring rule.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support this feature.
- (-ENODEV) if ∗port_id∗ invalid.
- (-EINVAL) if the mr_conf information is not correct.

3.11.5.65 int rte_eth_mirror_rule_reset (uint8_t *port_id*, uint8_t *rule_id*)

Reset a traffic mirroring rule on an Ethernet device.

Parameters

port_id	The port identifier of the Ethernet device.
rule_id	The index of traffic mirroring rule, we support four separated rules.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support this feature.
- (-ENODEV) if ∗port_id∗ invalid.
- (-EINVAL) if bad parameter.

3.11.5.66 int rte_eth_dev_bypass_init (uint8_t *port*)

Initialize bypass logic. This function needs to be called before executing any other bypass API.

Parameters

port	The port identifier of the Ethernet device.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support.

- (-EINVAL) if bad parameter.

3.11.5.67 int rte_eth_dev_bypass_state_show (uint8_t *port*, uint32_t * *state*)

Return bypass state.

Parameters

port	The port identifier of the Ethernet device.
state	The return bypass state. • (1) Normal mode • (2) Bypass mode • (3) Isolate mode

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support.
- (-EINVAL) if bad parameter.

3.11.5.68 int rte_eth_dev_bypass_state_set (uint8_t *port*, uint32_t * *new_state*)

Set bypass state

Parameters

port	The port identifier of the Ethernet device.
state	The current bypass state. • (1) Normal mode • (2) Bypass mode • (3) Isolate mode

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support.
- (-EINVAL) if bad parameter.

3.11.5.69 int rte_eth_dev_bypass_event_show (uint8_t *port*, uint32_t *event*, uint32_t * *state*)

Return bypass state when given event occurs.

Parameters

port	The port identifier of the Ethernet device.
event	The bypass event • (1) Main power on (power button is pushed) • (2) Auxiliary power on (power supply is being plugged) • (3) Main power off (system shutdown and power supply is left plugged in) • (4) Auxiliary power off (power supply is being unplugged) • (5) Display or set the watchdog timer
state	The bypass state when given event occurred. • (1) Normal mode • (2) Bypass mode • (3) Isolate mode

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support.
- (-EINVAL) if bad parameter.

3.11.5.70 int rte_eth_dev_bypass_event_store (uint8_t *port*, uint32_t *event*, uint32_t *state*)

Set bypass state when given event occurs.

Parameters

port	The port identifier of the Ethernet device.
event	The bypass event • (1) Main power on (power button is pushed) • (2) Auxiliary power on (power supply is being plugged) • (3) Main power off (system shutdown and power supply is left plugged in) • (4) Auxiliary power off (power supply is being unplugged) • (5) Display or set the watchdog timer

state	The assigned state when given event occurs. • (1) Normal mode • (2) Bypass mode • (3) Isolate mode

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support.
- (-EINVAL) if bad parameter.

3.11.5.71 int rte_eth_dev_wd_timeout_store (uint8_t *port*, uint32_t *timeout*)

Set bypass watchdog timeout count.

Parameters

port	The port identifier of the Ethernet device.
state	The timeout to be set. • (0) 0 seconds (timer is off) • (1) 1.5 seconds • (2) 2 seconds • (3) 3 seconds • (4) 4 seconds • (5) 8 seconds • (6) 16 seconds • (7) 32 seconds

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support.
- (-EINVAL) if bad parameter.

3.11.5.72 int rte_eth_dev_bypass_ver_show (uint8_t *port*, uint32_t * *ver*)

Get bypass firmware version.

Parameters

port	The port identifier of the Ethernet device.
ver	The firmware version

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support.
- (-EINVAL) if bad parameter.

3.11.5.73 int rte_eth_dev_bypass_wd_timeout_show (uint8_t *port*, uint32_t * *wd_timeout*)

Return bypass watchdog timeout in seconds

Parameters

port	The port identifier of the Ethernet device.
wd_timeout	The return watchdog timeout. "0" represents timer expired • (0) 0 seconds (timer is off) • (1) 1.5 seconds • (2) 2 seconds • (3) 3 seconds • (4) 4 seconds • (5) 8 seconds • (6) 16 seconds • (7) 32 seconds

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support.
- (-EINVAL) if bad parameter.

3.11.5.74 int rte_eth_dev_bypass_wd_reset (uint8_t *port*)

Reset bypass watchdog timer

Parameters

port	The port identifier of the Ethernet device.

Returns

- (0) if successful.
- (-ENOTSUP) if hardware doesn't support.
- (-EINVAL) if bad parameter.

3.12 rte_ether.h File Reference

Data Structures

- struct ether_addr
- struct ether_hdr
- struct vlan_hdr

Defines

- #define ETHER_ADDR_LEN
- #define ETHER_TYPE_LEN
- #define ETHER_CRC_LEN
- #define ETHER_HDR_LEN
- #define ETHER_MIN_LEN
- #define ETHER_MAX_LEN
- #define ETHER_MTU
- #define ETHER_MAX_VLAN_FRAME_LEN
- #define ETHER_MAX_JUMBO_FRAME_LEN
- #define ETHER_MAX_VLAN_ID
- #define ETHER_LOCAL_ADMIN_ADDR
- #define ETHER_GROUP_ADDR
- #define ETHER_TYPE_IPv4
- #define ETHER_TYPE_IPv6
- #define ETHER_TYPE_ARP
- #define ETHER_TYPE_RARP
- #define ETHER_TYPE_VLAN
- #define ETHER_TYPE_1588

Functions

- static int is_zero_ether_addr (const struct ether_addr *ea)
- static int is_unicast_ether_addr (const struct ether_addr *ea)
- static int is_multicast_ether_addr (const struct ether_addr *ea)
- static int is_broadcast_ether_addr (const struct ether_addr *ea)
- static int is_universal_ether_addr (const struct ether_addr *ea)
- static int is_local_admin_ether_addr (const struct ether_addr *ea)
- static int is_valid_assigned_ether_addr (const struct ether_addr *ea)
- static void eth_random_addr (uint8_t *addr)
- static void ether_addr_copy (const struct ether_addr *ea_from, struct ether_addr *ea_to)

3.12.1 Detailed Description

Ethernet Helpers in RTE

3.12.2 Define Documentation

3.12.2.1 #define ETHER_ADDR_LEN

Length of Ethernet address.

3.12.2.2 #define ETHER_TYPE_LEN

Length of Ethernet type field.

3.12.2.3 #define ETHER_CRC_LEN

Length of Ethernet CRC.

3.12.2.4 #define ETHER_HDR_LEN

Length of Ethernet header.

3.12.2.5 #define ETHER_MIN_LEN

Minimum frame len, including CRC.

3.12.2.6 #define ETHER_MAX_LEN

Maximum frame len, including CRC.

3.12.2.7 #define ETHER_MTU

Ethernet MTU.

3.12.2.8 #define ETHER_MAX_VLAN_FRAME_LEN

Maximum VLAN frame length, including CRC.

3.12.2.9 #define ETHER_MAX_JUMBO_FRAME_LEN

Maximum Jumbo frame length, including CRC.

3.12.2.10 #define ETHER_MAX_VLAN_ID

Maximum VLAN ID.

3.12.2.11 #define ETHER_LOCAL_ADMIN_ADDR

Locally assigned Eth. address.

3.12.2.12 #define ETHER_GROUP_ADDR

Multicast or broadcast Eth. address.

3.12.2.13 #define ETHER_TYPE_IPv4

IPv4 Protocol.

3.12.2.14 #define ETHER_TYPE_IPv6

IPv6 Protocol.

3.12.2.15 #define ETHER_TYPE_ARP

Arp Protocol.

3.12.2.16 #define ETHER_TYPE_RARP

Reverse Arp Protocol.

3.12.2.17 #define ETHER_TYPE_VLAN

IEEE 802.1Q VLAN tagging.

3.12.2.18 #define ETHER_TYPE_1588

IEEE 802.1AS 1588 Precise Time Protocol.

3.12.3 Function Documentation

3.12.3.1 static int is_zero_ether_addr (const struct ether_addr * ea) [static]

Check if an Ethernet address is filled with zeros.

Parameters

ea	A pointer to a ether_addr structure containing the ethernet address to check.

Returns

True (1) if the given ethernet address is filled with zeros; false (0) otherwise.

3.12.3.2 static int is_unicast_ether_addr (const struct ether_addr * ea) [static]

Check if an Ethernet address is a unicast address.

Parameters

ea	A pointer to a ether_addr structure containing the ethernet address to check.

Returns

True (1) if the given ethernet address is a unicast address; false (0) otherwise.

3.12.3.3 static int is_multicast_ether_addr (const struct ether_addr * ea) [static]

Check if an Ethernet address is a multicast address.

Parameters

ea	A pointer to a ether_addr structure containing the ethernet address to check.

Returns

True (1) if the given ethernet address is a multicast address; false (0) otherwise.

3.12.3.4 static int is_broadcast_ether_addr (const struct ether_addr * ea) [static]

Check if an Ethernet address is a broadcast address.

Parameters

ea	A pointer to a ether_addr structure containing the ethernet address to check.

Returns

True (1) if the given ethernet address is a broadcast address; false (0) otherwise.

3.12.3.5 static int is_universal_ether_addr (const struct ether_addr * *ea*) [static]

Check if an Ethernet address is a universally assigned address.

Parameters

ea	A pointer to a ether_addr structure containing the ethernet address to check.

Returns

True (1) if the given ethernet address is a universally assigned address; false (0) otherwise.

3.12.3.6 static int is_local_admin_ether_addr (const struct ether_addr * *ea*) [static]

Check if an Ethernet address is a locally assigned address.

Parameters

ea	A pointer to a ether_addr structure containing the ethernet address to check.

Returns

True (1) if the given ethernet address is a locally assigned address; false (0) otherwise.

3.12.3.7 static int is_valid_assigned_ether_addr (const struct ether_addr * *ea*) [static]

Check if an Ethernet address is a valid address. Checks that the address is a unicast address and is not filled with zeros.

Parameters

ea	A pointer to a ether_addr structure containing the ethernet address to check.

Returns

True (1) if the given ethernet address is valid; false (0) otherwise.

3.12.3.8 static void eth_random_addr (uint8_t * *addr*) [static]

Generate a random Ethernet address that is locally administered and not multicast.

Parameters

addr	A pointer to Ethernet address.

3.12.3.9 static void ether_addr_copy (const struct ether_addr * *ea_from*, struct ether_addr * *ea_to*) [static]

Fast copy an Ethernet address.

Parameters

ea_from	A pointer to a ether_addr structure holding the Ethernet address to copy.
ea_to	A pointer to a ether_addr structure where to copy the Ethernet address.

3.13 rte_fbk_hash.h File Reference

Data Structures

- struct rte_fbk_hash_params
- union rte_fbk_hash_entry
- struct rte_fbk_hash_table

Defines

- #define RTE_FBK_HASH_INIT_VAL_DEFAULT
- #define RTE_FBK_HASH_ENTRIES_MAX
- #define RTE_FBK_HASH_ENTRIES_PER_BUCKET_MAX
- #define RTE_FBK_HASH_NAMESIZE

Typedefs

- typedef uint32_t(* rte_fbk_hash_fn)(uint32_t key, uint32_t init_val)

Functions

- static uint32_t rte_fbk_hash_get_bucket (const struct rte_fbk_hash_table *ht, uint32_t key)
- static int rte_fbk_hash_add_key_with_bucket (struct rte_fbk_hash_table *ht, uint32_t key, uint16_t value, uint32_t bucket)
- static int rte_fbk_hash_add_key (struct rte_fbk_hash_table *ht, uint32_t key, uint16_t value)
- static int rte_fbk_hash_delete_key_with_bucket (struct rte_fbk_hash_table *ht, uint32_t key, uint32_t bucket)
- static int rte_fbk_hash_delete_key (struct rte_fbk_hash_table *ht, uint32_t key)
- static int rte_fbk_hash_lookup_with_bucket (const struct rte_fbk_hash_table *ht, uint32_t key, uint32_t bucket)

- static int rte_fbk_hash_lookup (const struct rte_fbk_hash_table *ht, uint32_t key)
- static void rte_fbk_hash_clear_all (struct rte_fbk_hash_table *ht)
- static double rte_fbk_hash_get_load_factor (struct rte_fbk_hash_table *ht)
- struct rte_fbk_hash_table * rte_fbk_hash_find_existing (const char *name)
- struct rte_fbk_hash_table * rte_fbk_hash_create (const struct rte_fbk_hash_params *params)
- void rte_fbk_hash_free (struct rte_fbk_hash_table *ht)

3.13.1 Detailed Description

This is a hash table implementation for four byte keys (fbk).

Note that the return value of the add function should always be checked as, if a bucket is full, the key is not added even if there is space in other buckets. This keeps the lookup function very simple and therefore fast.

3.13.2 Define Documentation

3.13.2.1 #define RTE_FBK_HASH_INIT_VAL_DEFAULT

Initialising value used when calculating hash.

3.13.2.2 #define RTE_FBK_HASH_ENTRIES_MAX

The maximum number of entries in the hash table that is supported.

3.13.2.3 #define RTE_FBK_HASH_ENTRIES_PER_BUCKET_MAX

The maximum number of entries in each bucket that is supported.

3.13.2.4 #define RTE_FBK_HASH_NAMESIZE

Maximum size of string for naming the hash.

3.13.3 Typedef Documentation

3.13.3.1 typedef uint32_t(* rte_fbk_hash_fn)(uint32_t key, uint32_t init_val)

Type of function that can be used for calculating the hash value.

3.13.4 Function Documentation

3.13.4.1 static uint32_t rte_fbk_hash_get_bucket (const struct rte_fbk_hash_table * *ht*, uint32_t *key*) [static]

Find the offset into hash table of the bucket containing a particular key.

Parameters

ht	Pointer to hash table.
key	Key to calculate bucket for.

Returns

Offset into hash table.

3.13.4.2 static int rte_fbk_hash_add_key_with_bucket (struct rte_fbk_hash_table * *ht*, uint32_t *key*, uint16_t *value*, uint32_t *bucket*) [static]

Add a key to an existing hash table with bucket id. This operation is not multi-thread safe and should only be called from one thread.

Parameters

ht	Hash table to add the key to.
key	Key to add to the hash table.
value	Value to associate with key.
bucket	Bucket to associate with key.

Returns

0 if ok, or negative value on error.

3.13.4.3 static int rte_fbk_hash_add_key (struct rte_fbk_hash_table * *ht*, uint32_t *key*, uint16_t *value*) [static]

Add a key to an existing hash table. This operation is not multi-thread safe and should only be called from one thread.

Parameters

ht	Hash table to add the key to.
key	Key to add to the hash table.
value	Value to associate with key.

Returns

0 if ok, or negative value on error.

3.13.4.4 static int rte_fbk_hash_delete_key_with_bucket (struct rte_fbk_hash_table * *ht*, uint32_t *key*, uint32_t *bucket*) [static]

Remove a key with a given bucket id from an existing hash table. This operation is not multi-thread safe and should only be called from one thread.

Parameters

ht	Hash table to remove the key from.
key	Key to remove from the hash table.
bucket	Bucket id associate with key.

Returns

0 if ok, or negative value on error.

3.13.4.5 static int rte_fbk_hash_delete_key (struct rte_fbk_hash_table * *ht*, uint32_t *key*) [static]

Remove a key from an existing hash table. This operation is not multi-thread safe and should only be called from one thread.

Parameters

ht	Hash table to remove the key from.
key	Key to remove from the hash table.

Returns

0 if ok, or negative value on error.

3.13.4.6 static int rte_fbk_hash_lookup_with_bucket (const struct rte_fbk_hash_table * *ht*, uint32_t *key*, uint32_t *bucket*) [static]

Find a key in the hash table with a given bucketid. This operation is multi-thread safe.

Parameters

ht	Hash table to look in.
key	Key to find.
bucket	Bucket associate to the key.

Returns

The value that was associated with the key, or negative value on error.

3.13.4.7 static int rte_fbk_hash_lookup (const struct rte_fbk_hash_table * *ht*, uint32_t *key*) [static]

Find a key in the hash table. This operation is multi-thread safe.

Parameters

ht	Hash table to look in.
key	Key to find.

Returns

The value that was associated with the key, or negative value on error.

3.13.4.8 static void rte_fbk_hash_clear_all (struct rte_fbk_hash_table * *ht*) [static]

Delete all entries in a hash table. This operation is not multi-thread safe and should only be called from one thread.

Parameters

ht	Hash table to delete entries in.

3.13.4.9 static double rte_fbk_hash_get_load_factor (struct rte_fbk_hash_table * *ht*) [static]

Find what fraction of entries are being used.

Parameters

ht	Hash table to find how many entries are being used in.

Returns

Load factor of the hash table, or negative value on error.

3.13.4.10 struct rte_fbk_hash_table* rte_fbk_hash_find_existing (const char * *name*) [read]

Performs a lookup for an existing hash table, and returns a pointer to the table if found.

Parameters

name	Name of the hash table to find

Returns

pointer to hash table structure or NULL on error with rte_errno set appropriately. Possible rte_errno values include:

- ENOENT - required entry not available to return.

3.13.4.11 struct rte_fbk_hash_table* rte_fbk_hash_create (const struct rte_fbk_hash_params * *params*) [read]

Create a new hash table for use with four byte keys.

Parameters

params	Parameters used in creation of hash table.

Returns

Pointer to hash table structure that is used in future hash table operations, or NULL on error with rte_errno set appropriately. Possible rte_errno error 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
- E_RTE_NO_TAILQ - no tailq list could be got for the fbk hash table list
- EINVAL - invalid parameter value passed to function
- 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

3.13.4.12 void rte_fbk_hash_free (struct rte_fbk_hash_table * *ht*)

Free all memory used by a hash table. Has no effect on hash tables allocated in memory zones

Parameters

ht	Hash table to deallocate.

3.14 rte_hash.h File Reference

Data Structures

- struct rte_hash_parameters
- struct rte_hash

Defines

- #define RTE_HASH_ENTRIES_MAX
- #define RTE_HASH_BUCKET_ENTRIES_MAX
- #define RTE_HASH_KEY_LENGTH_MAX
- #define RTE_HASH_LOOKUP_BULK_MAX
- #define RTE_HASH_NAMESIZE

Typedefs

- typedef uint32_t hash_sig_t
- typedef uint32_t(∗ rte_hash_function)(const void ∗key, uint32_t key_len, uint32_t init_val)

Functions

- struct rte_hash ∗ rte_hash_create (const struct rte_hash_parameters ∗params)
- struct rte_hash ∗ rte_hash_find_existing (const char ∗name)
- void rte_hash_free (struct rte_hash ∗h)
- int32_t rte_hash_add_key (const struct rte_hash ∗h, const void ∗key)
- int32_t rte_hash_add_key_with_hash (const struct rte_hash ∗h, const void ∗key, hash_sig_t sig)
- int32_t rte_hash_del_key (const struct rte_hash ∗h, const void ∗key)
- int32_t rte_hash_del_key_with_hash (const struct rte_hash ∗h, const void ∗key, hash_sig_t sig)
- int32_t rte_hash_lookup (const struct rte_hash ∗h, const void ∗key)
- int32_t rte_hash_lookup_with_hash (const struct rte_hash ∗h, const void ∗key, hash_sig_t sig)
- static hash_sig_t rte_hash_hash (const struct rte_hash ∗h, const void ∗key)
- int rte_hash_lookup_bulk (const struct rte_hash ∗h, const void ∗∗keys, uint32_t num_keys, int32_t ∗positions)

3.14.1 Detailed Description

RTE Hash Table

3.14.2 Define Documentation

3.14.2.1 #define RTE_HASH_ENTRIES_MAX

Maximum size of hash table that can be created.

3.14.2.2 #define RTE_HASH_BUCKET_ENTRIES_MAX

Maximum bucket size that can be created.

3.14.2.3 #define RTE_HASH_KEY_LENGTH_MAX

Maximum length of key that can be used.

3.14.2.4 #define RTE_HASH_LOOKUP_BULK_MAX

Max number of keys that can be searched for using rte_hash_lookup_multi.

3.14.2.5 #define RTE_HASH_NAMESIZE

Max number of characters in hash name.

3.14.3 Typedef Documentation

3.14.3.1 typedef uint32_t hash_sig_t

Signature of key that is stored internally.

3.14.3.2 typedef uint32_t(* rte_hash_function)(const void *key, uint32_t key_len, uint32_t init_val)

Type of function that can be used for calculating the hash value.

3.14.4 Function Documentation

3.14.4.1 struct rte_hash* rte_hash_create (const struct rte_hash_parameters * *params*) [read]

Create a new hash table.

Parameters

params	Parameters used to create and initialise the hash table.

Returns

Pointer to hash table structure that is used in future hash table operations, or NULL on error, with error code set in rte_errno. Possible rte_errno errors 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

- E_RTE_NO_TAILQ - no tailq list could be got for the hash table list
- ENOENT - missing entry
- EINVAL - invalid parameter passed to function
- 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

3.14.4.2 struct rte_hash* rte_hash_find_existing (const char * *name*) [read]

Find an existing hash table object and return a pointer to it.

Parameters

name	Name of the hash table as passed to rte_hash_create()

Returns

Pointer to hash table or NULL if object not found with rte_errno set appropriately. Possible rte_errno values include:

- ENOENT - value not available for return

3.14.4.3 void rte_hash_free (struct rte_hash * *h*)

De-allocate all memory used by hash table.

Parameters

h	Hash table to free

3.14.4.4 int32_t rte_hash_add_key (const struct rte_hash * *h*, const void * *key*)

Add a key to an existing hash table. This operation is not multi-thread safe and should only be called from one thread.

Parameters

h	Hash table to add the key to.
key	Key to add to the hash table.

Returns

- -EINVAL if the parameters are invalid.
- -ENOSPC if there is no space in the hash for this key.

- A positive value that can be used by the caller as an offset into an array of user data. This value is unique for this key.

3.14.4.5 int32_t rte_hash_add_key_with_hash (const struct rte_hash * h, const void * key, hash_sig_t sig)

Add a key to an existing hash table. This operation is not multi-thread safe and should only be called from one thread.

Parameters

h	Hash table to add the key to.
key	Key to add to the hash table.
sig	Hash value to add to the hash table.

Returns

- -EINVAL if the parameters are invalid.
- -ENOSPC if there is no space in the hash for this key.
- A positive value that can be used by the caller as an offset into an array of user data. This value is unique for this key.

3.14.4.6 int32_t rte_hash_del_key (const struct rte_hash * h, const void * key)

Remove a key from an existing hash table. This operation is not multi-thread safe and should only be called from one thread.

Parameters

h	Hash table to remove the key from.
key	Key to remove from the hash table.

Returns

- -EINVAL if the parameters are invalid.
- -ENOENT if the key is not found.
- A positive value that can be used by the caller as an offset into an array of user data. This value is unique for this key, and is the same value that was returned when the key was added.

3.14.4.7 int32_t rte_hash_del_key_with_hash (const struct rte_hash * h, const void * key, hash_sig_t sig)

Remove a key from an existing hash table. This operation is not multi-thread safe and should only be called from one thread.

Parameters

h	Hash table to remove the key from.
key	Key to remove from the hash table.
sig	Hash value to remove from the hash table.

Returns

- -EINVAL if the parameters are invalid.
- -ENOENT if the key is not found.
- A positive value that can be used by the caller as an offset into an array of user data. This value is unique for this key, and is the same value that was returned when the key was added.

3.14.4.8 int32_t rte_hash_lookup (const struct rte_hash * *h*, const void * *key*)

Find a key in the hash table. This operation is multi-thread safe.

Parameters

h	Hash table to look in.
key	Key to find.

Returns

- -EINVAL if the parameters are invalid.
- -ENOENT if the key is not found.
- A positive value that can be used by the caller as an offset into an array of user data. This value is unique for this key, and is the same value that was returned when the key was added.

3.14.4.9 int32_t rte_hash_lookup_with_hash (const struct rte_hash * *h*, const void * *key*, hash_sig_t *sig*)

Find a key in the hash table. This operation is multi-thread safe.

Parameters

h	Hash table to look in.
key	Key to find.
sig	Hash value to find.

Returns

- -EINVAL if the parameters are invalid.
- -ENOENT if the key is not found.
- A positive value that can be used by the caller as an offset into an array of user data. This value is unique for this key, and is the same value that was returned when the key was added.

3.14.4.10 static hash_sig_t rte_hash_hash (const struct rte_hash * h, const void * key) [static]

Calc a hash value by key. This operation is not multi-process safe.

Parameters

h	Hash table to look in.
key	Key to find.

Returns

- hash value

3.14.4.11 int rte_hash_lookup_bulk (const struct rte_hash * h, const void ** keys, uint32_t num_keys, int32_t * positions)

Find multiple keys in the hash table. This operation is multi-thread safe.

Parameters

h	Hash table to look in.
keys	A pointer to a list of keys to look for.
num_keys	How many keys are in the keys list (less than RTE_HASH_LOOKUP_BULK_MAX).
positions	Output containing a list of values, corresponding to the list of keys that can be used by the caller as an offset into an array of user data. These values are unique for each key, and are the same values that were returned when each key was added. If a key in the list was not found, then -ENOENT will be the value.

Returns

-EINVAL if there's an error, otherwise 0.

3.15 rte_hash_crc.h File Reference

Functions

- static uint32_t rte_hash_crc_4byte (uint32_t data, uint32_t init_val)
- static uint32_t rte_hash_crc (const void *data, uint32_t data_len, uint32_t init_val)

3.15.1 Detailed Description

RTE CRC Hash

3.15.2 Function Documentation

3.15.2.1 static uint32_t rte_hash_crc_4byte (uint32_t *data*, uint32_t *init_val*) [static]

Use single crc32 instruction to perform a hash on a 4 byte value.

Parameters

data	Data to perform hash on.
init_val	Value to initialise hash generator.

Returns

32bit calculated hash value.

3.15.2.2 static uint32_t rte_hash_crc (const void * *data*, uint32_t *data_len*, uint32_t *init_val*) [static]

Use crc32 instruction to perform a hash.

Parameters

data	Data to perform hash on.
data_len	How many bytes to use to calculate hash value.
init_val	Value to initialise hash generator.

Returns

32bit calculated hash value.

3.16 rte_hexdump.h File Reference

Functions

- void rte_hexdump (const char *title, const void *buf, unsigned int len)
- void rte_memdump (const char *title, const void *buf, unsigned int len)

3.16.1 Detailed Description

Simple API to dump out memory in a special hex format.

3.16.2 Function Documentation

3.16.2.1 void rte_hexdump (const char * *title*, const void * *buf*, unsigned int *len*)

Dump out memory in a special hex dump format.

Parameters

title	If not NULL this string is printed as a header to the output.
buf	This is the buffer address to print out.
len	The number of bytes to dump out

Returns

None.

3.16.2.2 void rte_memdump (const char * *title*, const void * *buf*, unsigned int *len*)

Dump out memory in a hex format with colons between bytes.

Parameters

title	If not NULL this string is printed as a header to the output.
buf	This is the buffer address to print out.
len	The number of bytes to dump out

Returns

None.

3.17 rte_interrupts.h File Reference

Typedefs

- typedef void(* rte_intr_callback_fn)(struct rte_intr_handle *intr_handle, void *cb_arg)

Functions

- int rte_intr_callback_register (struct rte_intr_handle *intr_handle, rte_intr_callback_fn cb, void *cb_arg)
- int rte_intr_callback_unregister (struct rte_intr_handle *intr_handle, rte_intr_callback_fn cb, void *cb_arg)
- int rte_intr_enable (struct rte_intr_handle *intr_handle)
- int rte_intr_disable (struct rte_intr_handle *intr_handle)

3.17.1 Detailed Description

The RTE interrupt interface provides functions to register/unregister callbacks for a specific interrupt.

3.17.2 Typedef Documentation

3.17.2.1 typedef void(* rte_intr_callback_fn)(struct rte_intr_handle *intr_handle, void *cb_arg)

Function to be registered for the specific interrupt

3.17.3 Function Documentation

3.17.3.1 int rte_intr_callback_register (struct rte_intr_handle * *intr_handle*, rte_intr_callback_fn *cb*, void * *cb_arg*)

It registers the callback for the specific interrupt. Multiple callbacks cal be registered at the same time.

Parameters

intr_handle	Pointer to the interrupt handle.
cb	callback address.
cb_arg	address of parameter for callback.

Returns

- On success, zero.
- On failure, a negative value.

3.17.3.2 int rte_intr_callback_unregister (struct rte_intr_handle * *intr_handle*, rte_intr_callback_fn *cb*, void * *cb_arg*)

It unregisters the callback according to the specified interrupt handle.

Parameters

intr_handle	pointer to the interrupt handle.
cb	callback address.
cb_arg	address of parameter for callback, (void *)-1 means to remove all registered which has the same callback address.

Returns

- On success, return the number of callback entities removed.
- On failure, a negative value.

3.17.3.3 int rte_intr_enable (struct rte_intr_handle * *intr_handle*)

It enables the interrupt for the specified handle.

Parameters

intr_handle	pointer to the interrupt handle.

Returns

- On success, zero.
- On failure, a negative value.

3.17.3.4 int rte_intr_disable (struct rte_intr_handle * *intr_handle*)

It disables the interrupt for the specified handle.

Parameters

intr_handle	pointer to the interrupt handle.

Returns

- On success, zero.
- On failure, a negative value.

3.18 rte_ip.h File Reference

Data Structures

- struct ipv4_hdr
- struct ipv6_hdr

Defines

- #define IPv4(a, b, c, d)
- #define IPPROTO_IP
- #define IPPROTO_HOPOPTS
- #define IPPROTO_ICMP
- #define IPPROTO_IGMP
- #define IPPROTO_GGP
- #define IPPROTO_IPV4
- #define IPPROTO_TCP

- #define IPPROTO_ST
- #define IPPROTO_EGP
- #define IPPROTO_PIGP
- #define IPPROTO_RCCMON
- #define IPPROTO_NVPII
- #define IPPROTO_PUP
- #define IPPROTO_ARGUS
- #define IPPROTO_EMCON
- #define IPPROTO_XNET
- #define IPPROTO_CHAOS
- #define IPPROTO_UDP
- #define IPPROTO_MUX
- #define IPPROTO_MEAS
- #define IPPROTO_HMP
- #define IPPROTO_PRM
- #define IPPROTO_IDP
- #define IPPROTO_TRUNK1
- #define IPPROTO_TRUNK2
- #define IPPROTO_LEAF1
- #define IPPROTO_LEAF2
- #define IPPROTO_RDP
- #define IPPROTO_IRTP
- #define IPPROTO_TP
- #define IPPROTO_BLT
- #define IPPROTO_NSP
- #define IPPROTO_INP
- #define IPPROTO_SEP
- #define IPPROTO_3PC
- #define IPPROTO_IDPR
- #define IPPROTO_XTP
- #define IPPROTO_DDP
- #define IPPROTO_CMTP
- #define IPPROTO_TPXX
- #define IPPROTO_IL
- #define IPPROTO_IPV6
- #define IPPROTO_SDRP
- #define IPPROTO_ROUTING
- #define IPPROTO_FRAGMENT
- #define IPPROTO_IDRP
- #define IPPROTO_RSVP
- #define IPPROTO_GRE
- #define IPPROTO_MHRP
- #define IPPROTO_BHA
- #define IPPROTO_ESP
- #define IPPROTO_AH
- #define IPPROTO_INLSP

- #define IPPROTO_SWIPE
- #define IPPROTO_NHRP
- #define IPPROTO_ICMPV6
- #define IPPROTO_NONE
- #define IPPROTO_DSTOPTS
- #define IPPROTO_AHIP
- #define IPPROTO_CFTP
- #define IPPROTO_HELLO
- #define IPPROTO_SATEXPAK
- #define IPPROTO_KRYPTOLAN
- #define IPPROTO_RVD
- #define IPPROTO_IPPC
- #define IPPROTO_ADFS
- #define IPPROTO_SATMON
- #define IPPROTO_VISA
- #define IPPROTO_IPCV
- #define IPPROTO_CPNX
- #define IPPROTO_CPHB
- #define IPPROTO_WSN
- #define IPPROTO_PVP
- #define IPPROTO_BRSATMON
- #define IPPROTO_ND
- #define IPPROTO_WBMON
- #define IPPROTO_WBEXPAK
- #define IPPROTO_EON
- #define IPPROTO_VMTP
- #define IPPROTO_SVMTP
- #define IPPROTO_VINES
- #define IPPROTO_TTP
- #define IPPROTO_IGP
- #define IPPROTO_DGP
- #define IPPROTO_TCF
- #define IPPROTO_IGRP
- #define IPPROTO_OSPFIGP
- #define IPPROTO_SRPC
- #define IPPROTO_LARP
- #define IPPROTO_MTP
- #define IPPROTO_AX25
- #define IPPROTO_IPEIP
- #define IPPROTO_MICP
- #define IPPROTO_SCCSP
- #define IPPROTO_ETHERIP
- #define IPPROTO_ENCAP
- #define IPPROTO_APES
- #define IPPROTO_GMTP
- #define IPPROTO_IPCOMP

- #define IPPROTO_PIM
- #define IPPROTO_PGM
- #define IPPROTO_SCTP
- #define IPPROTO_DIVERT
- #define IPPROTO_RAW
- #define IPPROTO_MAX
- #define IPV4_ANY
- #define IPV4_LOOPBACK
- #define IPV4_BROADCAST
- #define IPV4_ALLHOSTS_GROUP
- #define IPV4_ALLRTRS_GROUP
- #define IPV4_MAX_LOCAL_GROUP
- #define IPV4_MIN_MCAST
- #define IPV4_MAX_MCAST
- #define IS_IPV4_MCAST(x)

3.18.1 Detailed Description

IP-related defines

3.18.2 Define Documentation

3.18.2.1 #define IPv4(*a, b, c, d*)

Create IPv4 address

3.18.2.2 #define IPPROTO_IP

dummy for IP

3.18.2.3 #define IPPROTO_HOPOPTS

IP6 hop-by-hop options

3.18.2.4 #define IPPROTO_ICMP

control message protocol

3.18.2.5 #define IPPROTO_IGMP

group mgmt protocol

3.18.2.6 #define IPPROTO_GGP

gateway^2 (deprecated)

3.18.2.7 #define IPPROTO_IPV4

IPv4 encapsulation

3.18.2.8 #define IPPROTO_TCP

tcp

3.18.2.9 #define IPPROTO_ST

Stream protocol II

3.18.2.10 #define IPPROTO_EGP

exterior gateway protocol

3.18.2.11 #define IPPROTO_PIGP

private interior gateway

3.18.2.12 #define IPPROTO_RCCMON

BBN RCC Monitoring

3.18.2.13 #define IPPROTO_NVPII

network voice protocol

3.18.2.14 #define IPPROTO_PUP

pup

3.18.2.15 #define IPPROTO_ARGUS

Argus

3.18.2.16 #define IPPROTO_EMCON

EMCON

3.18.2.17 #define IPPROTO_XNET

Cross Net Debugger

3.18.2.18 #define IPPROTO_CHAOS

Chaos

3.18.2.19 #define IPPROTO_UDP

user datagram protocol

3.18.2.20 #define IPPROTO_MUX

Multiplexing

3.18.2.21 #define IPPROTO_MEAS

DCN Measurement Subsystems

3.18.2.22 #define IPPROTO_HMP

Host Monitoring

3.18.2.23 #define IPPROTO_PRM

Packet Radio Measurement

3.18.2.24 #define IPPROTO_IDP

xns idp

3.18.2.25 #define IPPROTO_TRUNK1

Trunk-1

3.18.2.26 #define IPPROTO_TRUNK2

Trunk-2

3.18.2.27 #define IPPROTO_LEAF1

Leaf-1

3.18.2.28 #define IPPROTO_LEAF2

Leaf-2

3.18.2.29 #define IPPROTO_RDP

Reliable Data

3.18.2.30 #define IPPROTO_IRTP

Reliable Transaction

3.18.2.31 #define IPPROTO_TP

tp-4 w/ class negotiation

3.18.2.32 #define IPPROTO_BLT

Bulk Data Transfer

3.18.2.33 #define IPPROTO_NSP

Network Services

3.18.2.34 #define IPPROTO_INP

Merit Internodal

3.18.2.35 #define IPPROTO_SEP

Sequential Exchange

3.18.2.36 #define IPPROTO_3PC

Third Party Connect

3.18.2.37 #define IPPROTO_IDPR

InterDomain Policy Routing

3.18.2.38 #define IPPROTO_XTP

XTP

3.18.2.39 #define IPPROTO_DDP

Datagram Delivery

3.18.2.40 #define IPPROTO_CMTP

Control Message Transport

3.18.2.41 #define IPPROTO_TPXX

TP++ Transport

3.18.2.42 #define IPPROTO_IL

IL transport protocol

3.18.2.43 #define IPPROTO_IPV6

IP6 header

3.18.2.44 #define IPPROTO_SDRP

Source Demand Routing

3.18.2.45 #define IPPROTO_ROUTING

IP6 routing header

3.18.2.46 #define IPPROTO_FRAGMENT

IP6 fragmentation header

3.18.2.47 #define IPPROTO_IDRP

InterDomain Routing

3.18.2.48 #define IPPROTO_RSVP

resource reservation

3.18.2.49 #define IPPROTO_GRE

General Routing Encap.

3.18.2.50 #define IPPROTO_MHRP

Mobile Host Routing

3.18.2.51 #define IPPROTO_BHA

BHA

3.18.2.52 #define IPPROTO_ESP

IP6 Encap Sec. Payload

3.18.2.53 #define IPPROTO_AH

IP6 Auth Header

3.18.2.54 #define IPPROTO_INLSP

Integ. Net Layer Security

3.18.2.55 #define IPPROTO_SWIPE

IP with encryption

3.18.2.56 #define IPPROTO_NHRP

Next Hop Resolution

3.18.2.57 #define IPPROTO_ICMPV6

ICMP6

3.18.2.58 #define IPPROTO_NONE

IP6 no next header

3.18.2.59 #define IPPROTO_DSTOPTS

IP6 destination option

3.18.2.60 #define IPPROTO_AHIP

any host internal protocol

3.18.2.61 #define IPPROTO_CFTP

CFTP

3.18.2.62 #define IPPROTO_HELLO

"hello" routing protocol

3.18.2.63 #define IPPROTO_SATEXPAK

SATNET/Backroom EXPAK

3.18.2.64 #define IPPROTO_KRYPTOLAN

Kryptolan

3.18.2.65 #define IPPROTO_RVD

Remote Virtual Disk

3.18.2.66 #define IPPROTO_IPPC

Pluribus Packet Core

3.18.2.67 #define IPPROTO_ADFS

Any distributed FS

3.18.2.68 #define IPPROTO_SATMON

Satnet Monitoring

3.18.2.69 #define IPPROTO_VISA

VISA Protocol

3.18.2.70 #define IPPROTO_IPCV

Packet Core Utility

3.18.2.71 #define IPPROTO_CPNX

Comp. Prot. Net. Executive

3.18.2.72 #define IPPROTO_CPHB

Comp. Prot. HeartBeat

3.18.2.73 #define IPPROTO_WSN

Wang Span Network

3.18.2.74 #define IPPROTO_PVP

Packet Video Protocol

3.18.2.75 #define IPPROTO_BRSATMON

BackRoom SATNET Monitoring

3.18.2.76 #define IPPROTO_ND

Sun net disk proto (temp.)

3.18.2.77 #define IPPROTO_WBMON

WIDEBAND Monitoring

3.18.2.78 #define IPPROTO_WBEXPAK

WIDEBAND EXPAK

3.18.2.79 #define IPPROTO_EON

ISO cnlp

3.18.2.80 #define IPPROTO_VMTP

VMTP

3.18.2.81 #define IPPROTO_SVMTP

Secure VMTP

3.18.2.82 #define IPPROTO_VINES

Banyon VINES

3.18.2.83 #define IPPROTO_TTP

TTP

3.18.2.84 #define IPPROTO_IGP

NSFNET-IGP

3.18.2.85 #define IPPROTO_DGP

dissimilar gateway prot.

3.18.2.86 #define IPPROTO_TCF

TCF

3.18.2.87 #define IPPROTO_IGRP

Cisco/GXS IGRP

3.18.2.88 #define IPPROTO_OSPFIGP

OSPFIGP

3.18.2.89 #define IPPROTO_SRPC

Strite RPC protocol

3.18.2.90 #define IPPROTO_LARP

Locus Address Resoloution

3.18.2.91 #define IPPROTO_MTP

Multicast Transport

3.18.2.92 #define IPPROTO_AX25

AX.25 Frames

3.18.2.93 #define IPPROTO_IPEIP

IP encapsulated in IP

3.18.2.94 #define IPPROTO_MICP

Mobile Int.ing control

3.18.2.95 #define IPPROTO_SCCSP

Semaphore Comm. security

3.18.2.96 #define IPPROTO_ETHERIP

Ethernet IP encapsulation

3.18.2.97 #define IPPROTO_ENCAP

encapsulation header

3.18.2.98 #define IPPROTO_APES

any private encr. scheme

3.18.2.99 #define IPPROTO_GMTP

GMTP

3.18.2.100 #define IPPROTO_IPCOMP

payload compression (IPComp)

3.18.2.101 #define IPPROTO_PIM

Protocol Independent Mcast

3.18.2.102 #define IPPROTO_PGM

PGM

3.18.2.103 #define IPPROTO_SCTP

Stream Control Transport Protocol

3.18.2.104 #define IPPROTO_DIVERT

divert pseudo-protocol

3.18.2.105 #define IPPROTO_RAW

raw IP packet

3.18.2.106 #define IPPROTO_MAX

maximum protocol number

3.18.2.107 #define IPV4_ANY

0.0.0.0

3.18.2.108 #define IPV4_LOOPBACK

127.0.0.1

3.18.2.109 #define IPV4_BROADCAST

224.0.0.0

3.18.2.110 #define IPV4_ALLHOSTS_GROUP

224.0.0.1

3.18.2.111 #define IPV4_ALLRTRS_GROUP

224.0.0.2

3.18.2.112 #define IPV4_MAX_LOCAL_GROUP

224.0.0.255

3.18.2.113 #define IPV4_MIN_MCAST

Minimal IPv4-multicast address

3.18.2.114 #define IPV4_MAX_MCAST

Maximum IPv4 multicast address

3.18.2.115 #define IS_IPV4_MCAST(*x*)

check if IPv4 address is multicast

3.19 rte_ivshmem.h File Reference

Data Structures

- struct rte_ivshmem_metadata_entry
- struct rte_ivshmem_metadata

Functions

- int rte_ivshmem_metadata_create (const char *name)
- int rte_ivshmem_metadata_add_memzone (const struct rte_memzone *mz, const char *md_name)
- int rte_ivshmem_metadata_add_ring (const struct rte_ring *r, const char *md_name)
- int rte_ivshmem_metadata_add_mempool (const struct rte_mempool *mp, const char *md_name)
- int rte_ivshmem_metadata_cmdline_generate (char *buffer, unsigned size, const char *name)

Dump all metadata entries from a given metadata file to the console.

Name of the metadata file to be dumped to console.

3.19.1 Detailed Description

The RTE IVSHMEM interface provides functions to create metadata files describing memory segments to be shared via QEMU IVSHMEM.

3.19.2 Function Documentation

3.19.2.1 int rte_ivshmem_metadata_create (const char * *name*)

Creates metadata file with a given name

Parameters

name	Name of metadata file to be created

Returns

- On success, zero
- On failure, a negative value

3.19.2.2 int rte_ivshmem_metadata_add_memzone (const struct rte_memzone * *mz*, const char * *md_name*)

Adds memzone to a specific metadata file

Parameters

mz	Memzone to be added
md_name	Name of metadata file for the memzone to be added to

Returns

- On success, zero
- On failure, a negative value

3.19.2.3 int rte_ivshmem_metadata_add_ring (const struct rte_ring * r, const char * md_name)

Adds a ring descriptor to a specific metadata file

Parameters

r	Ring descriptor to be added
md_name	Name of metadata file for the ring to be added to

Returns

- On success, zero
- On failure, a negative value

3.19.2.4 int rte_ivshmem_metadata_add_mempool (const struct rte_mempool * mp, const char * md_name)

Adds a mempool to a specific metadata file

Parameters

mp	Mempool to be added
md_name	Name of metadata file for the mempool to be added to

Returns

- On success, zero
- On failure, a negative value

3.19.2.5 int rte_ivshmem_metadata_cmdline_generate (char * buffer, unsigned size, const char * name)

Generates the QEMU command-line for IVSHMEM device for a given metadata file. This function is to be called after all the objects were added.

Parameters

buffer	Buffer to be filled with the command line arguments.
size	Size of the buffer.
name	Name of metadata file to generate QEMU command-line parameters for

Returns

- On success, zero
- On failure, a negative value

3.20 rte_jhash.h File Reference

Defines

- #define RTE_JHASH_GOLDEN_RATIO

Functions

- static uint32_t rte_jhash (const void *key, uint32_t length, uint32_t initval)
- static uint32_t rte_jhash2 (uint32_t *k, uint32_t length, uint32_t initval)
- static uint32_t rte_jhash_3words (uint32_t a, uint32_t b, uint32_t c, uint32_t initval)
- static uint32_t rte_jhash_2words (uint32_t a, uint32_t b, uint32_t initval)
- static uint32_t rte_jhash_1word (uint32_t a, uint32_t initval)

3.20.1 Detailed Description

jhash functions.

3.20.2 Define Documentation

3.20.2.1 #define RTE_JHASH_GOLDEN_RATIO

The golden ratio: an arbitrary value.

3.20.3 Function Documentation

3.20.3.1 static uint32_t rte_jhash (const void * *key*, uint32_t *length*, uint32_t *initval*) [static]

The most generic version, hashes an arbitrary sequence of bytes. No alignment or length assumptions are made about the input key.

Parameters

key	Key to calculate hash of.
length	Length of key in bytes.
initval	Initialising value of hash.

Returns

Calculated hash value.

3.20.3.2 static uint32_t rte_jhash2 (uint32_t * *k*, uint32_t *length*, uint32_t *initval*) [static]

A special optimized version that handles 1 or more of uint32_ts. The length parameter here is the number of uint32_ts in the key.

Parameters

k	Key to calculate hash of.
length	Length of key in units of 4 bytes.
initval	Initialising value of hash.

Returns

Calculated hash value.

3.20.3.3 static uint32_t rte_jhash_3words (uint32_t *a*, uint32_t *b*, uint32_t *c*, uint32_t *initval*) [static]

A special ultra-optimized versions that knows it is hashing exactly 3 words.

Parameters

a	First word to calcuate hash of.
b	Second word to calcuate hash of.
c	Third word to calcuate hash of.
initval	Initialising value of hash.

Returns

Calculated hash value.

3.20.3.4 static uint32_t rte_jhash_2words (uint32_t *a*, uint32_t *b*, uint32_t *initval*) [static]

A special ultra-optimized versions that knows it is hashing exactly 2 words.

Parameters

a	First word to calcuate hash of.
b	Second word to calcuate hash of.
initval	Initialising value of hash.

Returns

Calculated hash value.

3.20.3.5 static uint32_t rte_jhash_1word (uint32_t *a*, uint32_t *initval*) [static]

A special ultra-optimized versions that knows it is hashing exactly 1 word.

Parameters

a	Word to calcuate hash of.
initval	Initialising value of hash.

Returns

Calculated hash value.

3.21 rte_kni.h File Reference

Data Structures

- struct rte_kni_ops
- struct rte_kni_conf

Functions

- struct rte_kni * rte_kni_alloc (struct rte_mempool *pktmbuf_pool, const struct rte_kni_conf *conf, struct rte_kni_ops *ops)
- struct rte_kni * rte_kni_create (uint8_t port_id, unsigned mbuf_size, struct rte_mempool *pktmbuf_pool, struct rte_kni_ops *ops)
- int rte_kni_release (struct rte_kni *kni)
- int rte_kni_handle_request (struct rte_kni *kni)
- unsigned rte_kni_rx_burst (struct rte_kni *kni, struct rte_mbuf **mbufs, unsigned num)
- unsigned rte_kni_tx_burst (struct rte_kni *kni, struct rte_mbuf **mbufs, unsigned num)
- uint8_t rte_kni_get_port_id (struct rte_kni *kni)
- struct rte_kni * rte_kni_get (const char *name)
- struct rte_kni * rte_kni_info_get (uint8_t port_id)
- int rte_kni_register_handlers (struct rte_kni *kni, struct rte_kni_ops *ops)

- int rte_kni_unregister_handlers (struct rte_kni ∗kni)
- void rte_kni_close (void)

3.21.1 Detailed Description

RTE KNI

The KNI library provides the ability to create and destroy kernel NIC interfaces that may be used by the RTE application to receive/transmit packets from/to Linux kernel net interfaces.

This library provide two APIs to burst receive packets from KNI interfaces, and burst transmit packets to KNI interfaces.

3.21.2 Function Documentation

3.21.2.1 struct rte_kni∗ rte_kni_alloc (struct rte_mempool ∗ *pktmbuf_pool,* const struct rte_kni_conf ∗ *conf,* struct rte_kni_ops ∗ *ops*) [read]

Allocate KNI interface according to the port id, mbuf size, mbuf pool, configurations and callbacks for kernel requests.The KNI interface created in the kernel space is the net interface the traditional Linux application talking to.

Parameters

pktmbuf_pool	The mempool for allocting mbufs for packets.
conf	The pointer to the configurations of the KNI device.
ops	The pointer to the callbacks for the KNI kernel requests.

Returns

- The pointer to the context of a KNI interface.
- NULL indicate error.

3.21.2.2 struct rte_kni∗ rte_kni_create (uint8_t *port_id,* unsigned *mbuf_size,* struct rte_mempool ∗ *pktmbuf_pool,* struct rte_kni_ops ∗ *ops*) [read]

It create a KNI device for specific port.

Note: It is deprecated and just for backward compatibility.

Parameters

port_id	Port ID.
mbuf_size	mbuf size.
pktmbuf_pool	The mempool for allocting mbufs for packets.
ops	The pointer to the callbacks for the KNI kernel requests.

Returns

- The pointer to the context of a KNI interface.
- NULL indicate error.

3.21.2.3 int rte_kni_release (struct rte_kni * *kni*)

Release KNI interface according to the context. It will also release the paired KNI interface in kernel space. All processing on the specific KNI context need to be stopped before calling this interface.

Parameters

kni	The pointer to the context of an existant KNI interface.

Returns

- 0 indicates success.
- negative value indicates failure.

3.21.2.4 int rte_kni_handle_request (struct rte_kni * *kni*)

It is used to handle the request mbufs sent from kernel space. Then analyzes it and calls the specific actions for the specific requests. Finally constructs the response mbuf and puts it back to the resp_q.

Parameters

kni	The pointer to the context of an existant KNI interface.

Returns

- 0
- negative value indicates failure.

3.21.2.5 unsigned rte_kni_rx_burst (struct rte_kni * *kni,* struct **rte_mbuf** ** *mbufs,* unsigned *num*)

Retrieve a burst of packets from a KNI interface. The retrieved packets are stored in rte_mbuf structures whose pointers are supplied in the array of mbufs, and the maximum number is indicated by num. It handles the freeing of the mbufs in the free queue of KNI interface.

Parameters

kni	The KNI interface context.
mbufs	The array to store the pointers of mbufs.
num	The maximum number per burst.

Returns

The actual number of packets retrieved.

3.21.2.6 unsigned rte_kni_tx_burst (struct rte_kni * *kni*, struct rte_mbuf ** *mbufs*, unsigned *num*)

Send a burst of packets to a KNI interface. The packets to be sent out are stored in rte_mbuf structures whose pointers are supplied in the array of mbufs, and the maximum number is indicated by num. It handles allocating the mbufs for KNI interface alloc queue.

Parameters

kni	The KNI interface context.
mbufs	The array to store the pointers of mbufs.
num	The maximum number per burst.

Returns

The actual number of packets sent.

3.21.2.7 uint8_t rte_kni_get_port_id (struct rte_kni * *kni*)

Get the port id from KNI interface.

Note: It is deprecated and just for backward compatibility.

Parameters

kni	The KNI interface context.

Returns

On success: The port id. On failure: ~0x0

3.21.2.8 struct rte_kni* rte_kni_get (const char * *name*) [read]

Get the KNI context of its name.

Parameters

name	pointer to the KNI device name.

Returns

On success: Pointer to KNI interface. On failure: NULL.

3.21.2.9 struct rte_kni* rte_kni_info_get (uint8_t *port_id*) [read]

Get the KNI context of the specific port.

Note: It is deprecated and just for backward compatibility.

Parameters

port_id	the port id.

Returns

On success: Pointer to KNI interface. On failure: NULL

3.21.2.10 int rte_kni_register_handlers (struct rte_kni * *kni*, struct rte_kni_ops * *ops*)

Register KNI request handling for a specified port,and it can be called by master process or slave process.

Parameters

kni	pointer to struct rte_kni.
ops	ponter to struct rte_kni_ops.

Returns

On success: 0 On failure: -1

3.21.2.11 int rte_kni_unregister_handlers (struct rte_kni * *kni*)

Unregister KNI request handling for a specified port.

Parameters

kni	pointer to struct rte_kni.

Returns

On success: 0 On failure: -1

3.21.2.12 void rte_kni_close (void)

close KNI device.

Parameters

void	

Returns

void

3.22 rte_launch.h File Reference

Typedefs

- typedef int(lcore_function_t)(void ∗)

Enumerations

- enum rte_lcore_state_t { WAIT, RUNNING, FINISHED }
- enum rte_rmt_call_master_t { SKIP_MASTER, CALL_MASTER }

Functions

- int rte_eal_remote_launch (lcore_function_t ∗f, void ∗arg, unsigned slave_id)
- int rte_eal_mp_remote_launch (lcore_function_t ∗f, void ∗arg, enum rte_rmt_call_master_t call_-master)
- enum rte_lcore_state_t rte_eal_get_lcore_state (unsigned slave_id)
- int rte_eal_wait_lcore (unsigned slave_id)
- void rte_eal_mp_wait_lcore (void)

3.22.1 Detailed Description

Launch tasks on other lcores

3.22.2 Typedef Documentation

3.22.2.1 typedef int(lcore_function_t)(void ∗)

Definition of a remote launch function.

3.22.3 Enumeration Type Documentation

3.22.3.1 enum rte_lcore_state_t

State of an lcore.

Enumerator:

WAIT waiting a new command

RUNNING executing command

FINISHED command executed

3.22.3.2 enum rte_rmt_call_master_t

This enum indicates whether the master core must execute the handler launched on all logical cores.

Enumerator:

SKIP_MASTER lcore handler not executed by master core.

CALL_MASTER lcore handler executed by master core.

3.22.4 Function Documentation

3.22.4.1 int rte_eal_remote_launch (lcore_function_t * *f*, void * *arg*, unsigned *slave_id*)

Launch a function on another lcore.

To be executed on the MASTER lcore only.

Sends a message to a slave lcore (identified by the slave_id) that is in the WAIT state (this is true after the first call to rte_eal_init()). This can be checked by first calling rte_eal_wait_lcore(slave_id).

When the remote lcore receives the message, it switches to the RUNNING state, then calls the function f with argument arg. Once the execution is done, the remote lcore switches to a FINISHED state and the return value of f is stored in a local variable to be read using rte_eal_wait_lcore().

The MASTER lcore returns as soon as the message is sent and knows nothing about the completion of f.

Note: This function is not designed to offer optimum performance. It is just a practical way to launch a function on another lcore at initialization time.

Parameters

f	The function to be called.
arg	The argument for the function.
slave_id	The identifier of the lcore on which the function should be executed.

Returns

- 0: Success. Execution of function f started on the remote lcore.
- (-EBUSY): The remote lcore is not in a WAIT state.

3.22.4.2 int rte_eal_mp_remote_launch (lcore_function_t * f, void * arg, enum rte_rmt_call_master_t call_master)

Launch a function on all lcores.

Check that each SLAVE lcore is in a WAIT state, then call rte_eal_remote_launch() for each lcore.

Parameters

f	The function to be called.
arg	The argument for the function.
call_master	If call_master set to SKIP_MASTER, the MASTER lcore does not call the function. - If call_master is set to CALL_MASTER, the function is also called on master before returning. In any case, the master lcore returns as soon as it finished its job and knows nothing about the completion of f on the other lcores.

Returns

- 0: Success. Execution of function f started on all remote lcores.
- (-EBUSY): At least one remote lcore is not in a WAIT state. In this case, no message is sent to any of the lcores.

3.22.4.3 enum rte_lcore_state_t rte_eal_get_lcore_state (unsigned slave_id)

Get the state of the lcore identified by slave_id.

To be executed on the MASTER lcore only.

Parameters

slave_id	The identifier of the lcore.

Returns

The state of the lcore.

3.22.4.4 int rte_eal_wait_lcore (unsigned slave_id)

Wait until an lcore finishes its job.

To be executed on the MASTER lcore only.

If the slave lcore identified by the slave_id is in a FINISHED state, switch to the WAIT state. If the lcore is in RUNNING state, wait until the lcore finishes its job and moves to the FINISHED state.

Parameters

slave_id	The identifier of the lcore.

Returns

- 0: If the lcore identified by the slave_id is in a WAIT state.
- The value that was returned by the previous remote launch function call if the lcore identified by the slave_id was in a FINISHED or RUNNING state. In this case, it changes the state of the lcore to WAIT.

3.22.4.5 void rte_eal_mp_wait_lcore (void)

Wait until all lcores finish their jobs.

To be executed on the MASTER lcore only. Issue an rte_eal_wait_lcore() for every lcore. The return values are ignored.

After a call to rte_eal_mp_wait_lcore(), the caller can assume that all slave lcores are in a WAIT state.

3.23 rte_lcore.h File Reference

Defines

- #define LCORE_ID_ANY
- #define RTE_LCORE_FOREACH(i)
- #define RTE_LCORE_FOREACH_SLAVE(i)

Functions

- RTE_DECLARE_PER_LCORE (unsigned, _lcore_id)
- static unsigned rte_lcore_id (void)
- static unsigned rte_get_master_lcore (void)
- static unsigned rte_lcore_count (void)
- static unsigned rte_socket_id (void)
- static unsigned rte_lcore_to_socket_id (unsigned lcore_id)
- static int rte_lcore_is_enabled (unsigned lcore_id)
- static unsigned rte_get_next_lcore (unsigned i, int skip_master, int wrap)

3.23.1 Detailed Description

API for lcore and Socket Manipulation. Parts of this are execution environment specific.

3.23.2 Define Documentation

3.23.2.1 #define LCORE_ID_ANY

Any lcore.

3.23.2.2 #define RTE_LCORE_FOREACH(*i*)

Macro to browse all running lcores.

3.23.2.3 #define RTE_LCORE_FOREACH_SLAVE(*i*)

Macro to browse all running lcores except the master lcore.

3.23.3 Function Documentation

3.23.3.1 RTE_DECLARE_PER_LCORE (unsigned , _lcore_id)

Per core "core id".

3.23.3.2 static unsigned rte_lcore_id (void) [static]

Return the ID of the execution unit we are running on.

Returns

Logical core ID

3.23.3.3 static unsigned rte_get_master_lcore (void) [static]

Get the id of the master lcore

Returns

the id of the master lcore

3.23.3.4 static unsigned rte_lcore_count (void) [static]

Return the number of execution units (lcores) on the system.

Returns

the number of execution units (lcores) on the system.

3.23.3.5 static unsigned rte_socket_id (void) [static]

Return the ID of the physical socket of the logical core we are running on.

Returns

the ID of current lcoreid's physical socket

3.23.3.6 static unsigned rte_lcore_to_socket_id (unsigned *lcore_id*) [static]

Get the ID of the physical socket of the specified lcore

Parameters

lcore_id	the targeted lcore, which MUST be between 0 and RTE_MAX_LCORE-1.

Returns

the ID of lcoreid's physical socket

3.23.3.7 static int rte_lcore_is_enabled (unsigned *lcore_id*) [static]

Test if an lcore is enabled.

Parameters

lcore_id	The identifier of the lcore, which MUST be between 0 and RTE_MAX_LCORE-1.

Returns

True if the given lcore is enabled; false otherwise.

3.23.3.8 static unsigned rte_get_next_lcore (unsigned *i,* int *skip_master,* int *wrap*) [static]

Get the next enabled lcore ID.

Parameters

i	The current lcore (reference).
skip_master	If true, do not return the ID of the master lcore.
wrap	If true, go back to 0 when RTE_MAX_LCORE is reached; otherwise, return RTE_MAX-_LCORE.

Returns

The next lcore_id or RTE_MAX_LCORE if not found.

3.24 rte_log.h File Reference

Data Structures

- struct rte_logs

Defines

- #define RTE_LOGTYPE_EAL
- #define RTE_LOGTYPE_MALLOC
- #define RTE_LOGTYPE_RING
- #define RTE_LOGTYPE_MEMPOOL
- #define RTE_LOGTYPE_TIMER
- #define RTE_LOGTYPE_PMD
- #define RTE_LOGTYPE_HASH
- #define RTE_LOGTYPE_LPM
- #define RTE_LOGTYPE_KNI
- #define RTE_LOGTYPE_ACL
- #define RTE_LOGTYPE_POWER
- #define RTE_LOGTYPE_METER
- #define RTE_LOGTYPE_SCHED
- #define RTE_LOGTYPE_USER1
- #define RTE_LOGTYPE_USER2
- #define RTE_LOGTYPE_USER3
- #define RTE_LOGTYPE_USER4
- #define RTE_LOGTYPE_USER5
- #define RTE_LOGTYPE_USER6
- #define RTE_LOGTYPE_USER7
- #define RTE_LOGTYPE_USER8
- #define RTE_LOG_EMERG
- #define RTE_LOG_ALERT
- #define RTE_LOG_CRIT
- #define RTE_LOG_ERR
- #define RTE_LOG_WARNING
- #define RTE_LOG_NOTICE
- #define RTE_LOG_INFO
- #define RTE_LOG_DEBUG
- #define RTE_LOG(l, t,...)

Functions

- int rte_openlog_stream (FILE *f)
- void rte_set_log_level (uint32_t level)
- void rte_set_log_type (uint32_t type, int enable)
- int rte_log_cur_msg_loglevel (void)
- int rte_log_cur_msg_logtype (void)
- void rte_log_set_history (int enable)
- void rte_log_dump_history (void)
- int rte_log_add_in_history (const char *buf, size_t size)
- int rte_log (uint32_t level, uint32_t logtype, const char *format,...)
- int rte_vlog (uint32_t level, uint32_t logtype, const char *format, va_list ap)

Variables

- struct rte_logs rte_logs
- FILE * eal_default_log_stream

3.24.1 Detailed Description

RTE Logs API

This file provides a log API to RTE applications.

3.24.2 Define Documentation

3.24.2.1 #define RTE_LOGTYPE_EAL

Log related to eal.

3.24.2.2 #define RTE_LOGTYPE_MALLOC

Log related to malloc.

3.24.2.3 #define RTE_LOGTYPE_RING

Log related to ring.

3.24.2.4 #define RTE_LOGTYPE_MEMPOOL

Log related to mempool.

3.24.2.5 #define RTE_LOGTYPE_TIMER

Log related to timers.

3.24.2.6 #define RTE_LOGTYPE_PMD

Log related to poll mode driver.

3.24.2.7 #define RTE_LOGTYPE_HASH

Log related to hash table.

3.24.2.8 #define RTE_LOGTYPE_LPM

Log related to LPM.

3.24.2.9 #define RTE_LOGTYPE_KNI

Log related to KNI.

3.24.2.10 #define RTE_LOGTYPE_ACL

Log related to ACL.

3.24.2.11 #define RTE_LOGTYPE_POWER

Log related to power.

3.24.2.12 #define RTE_LOGTYPE_METER

Log related to QoS meter.

3.24.2.13 #define RTE_LOGTYPE_SCHED

Log related to QoS port scheduler.

3.24.2.14 #define RTE_LOGTYPE_USER1

User-defined log type 1.

3.24.2.15 #define RTE_LOGTYPE_USER2

User-defined log type 2.

3.24.2.16 #define RTE_LOGTYPE_USER3

User-defined log type 3.

3.24.2.17 #define RTE_LOGTYPE_USER4

User-defined log type 4.

3.24.2.18 #define RTE_LOGTYPE_USER5

User-defined log type 5.

3.24.2.19 #define RTE_LOGTYPE_USER6

User-defined log type 6.

3.24.2.20 #define RTE_LOGTYPE_USER7

User-defined log type 7.

3.24.2.21 #define RTE_LOGTYPE_USER8

User-defined log type 8.

3.24.2.22 #define RTE_LOG_EMERG

System is unusable.

3.24.2.23 #define RTE_LOG_ALERT

Action must be taken immediately.

3.24.2.24 #define RTE_LOG_CRIT

Critical conditions.

3.24.2.25 #define RTE_LOG_ERR

Error conditions.

3.24.2.26 #define RTE_LOG_WARNING

Warning conditions.

3.24.2.27 #define RTE_LOG_NOTICE

Normal but significant condition.

3.24.2.28 #define RTE_LOG_INFO

Informational.

3.24.2.29 #define RTE_LOG_DEBUG

Debug-level messages.

3.24.2.30 #define RTE_LOG(*l*, *t*, ...)

Generates a log message.

The RTE_LOG() is equivalent to rte_log() with two differences:

- RTE_LOG() can be used to remove debug logs at compilation time, depending on RTE_LOG_LEVEL configuration option, and compilation optimization level. If optimization is enabled, the tests involving constants only are pre-computed. If compilation is done with -O0, these tests will be done at run time.
- The log level and log type names are smaller, for example: RTE_LOG(INFO, EAL, "this is a %s", "log");

Parameters

l	Log level. A value between EMERG (1) and DEBUG (8). The short name is expanded by the macro, so it cannot be an integer value.
t	The log type, for example, EAL. The short name is expanded by the macro, so it cannot be an integer value.
fmt	The fmt string, as in printf(3), followed by the variable arguments required by the format.
args	The variable list of arguments according to the format string.

Returns

- 0: Success.
- Negative on error.

3.24.3 Function Documentation

3.24.3.1 int rte_openlog_stream (FILE * f)

Change the stream that will be used by the logging system.

This can be done at any time. The f argument represents the stream to be used to send the logs. If f is NULL, the default output is used, which is the serial line in case of bare metal, or directly sent to syslog in case of linux application.

Parameters

f	Pointer to the stream.

Returns

- 0 on success.
- Negative on error.

3.24.3.2 void rte_set_log_level (uint32_t level)

Set the global log level.

After this call, all logs that are lower or equal than level and lower or equal than the RTE_LOG_LEVEL configuration option will be displayed.

Parameters

level	Log level. A value between RTE_LOG_EMERG (1) and RTE_LOG_DEBUG (8).

3.24.3.3 void rte_set_log_type (uint32_t type, int enable)

Enable or disable the log type.

Parameters

type	Log type, for example, RTE_LOGTYPE_EAL.
enable	True for enable; false for disable.

3.24.3.4 int rte_log_cur_msg_loglevel (void)

Get the current loglevel for the message being processed.

Before calling the user-defined stream for logging, the log subsystem sets a per-lcore variable containing the loglevel and the logtype of the message being processed. This information can be accessed by the user-defined log output function through this function.

Returns

The loglevel of the message being processed.

3.24.3.5 int rte_log_cur_msg_logtype (void)

Get the current logtype for the message being processed.

Before calling the user-defined stream for logging, the log subsystem sets a per-lcore variable containing the loglevel and the logtype of the message being processed. This information can be accessed by the user-defined log output function through this function.

Returns

The logtype of the message being processed.

3.24.3.6 void rte_log_set_history (int *enable*)

Enable or disable the history (enabled by default)

Parameters

enable	true to enable, or 0 to disable history.

3.24.3.7 void rte_log_dump_history (void)

Dump the log history to the console.

3.24.3.8 int rte_log_add_in_history (const char * *buf*, size_t *size*)

Add a log message to the history.

This function can be called from a user-defined log stream. It adds the given message in the history that can be dumped using rte_log_dump_history().

Parameters

buf	A data buffer containing the message to be saved in the history.
size	The length of the data buffer.

Returns

- 0: Success.
- (-ENOBUFS) if there is no room to store the message.

3.24.3.9 int rte_log (uint32_t *level*, uint32_t *logtype*, const char * *format*, ...)

Generates a log message.

The message will be sent in the stream defined by the previous call to rte_openlog_stream().

The level argument determines if the log should be displayed or not, depending on the global rte_logs variable.

The preferred alternative is the RTE_LOG() function because debug logs may be removed at compilation time if optimization is enabled. Moreover, logs are automatically prefixed by type when using the macro.

Parameters

level	Log level. A value between RTE_LOG_EMERG (1) and RTE_LOG_DEBUG (8).
logtype	The log type, for example, RTE_LOGTYPE_EAL.
format	The format string, as in printf(3), followed by the variable arguments required by the format.

Returns

- 0: Success.
- Negative on error.

3.24.3.10 int rte_vlog (uint32_t *level*, uint32_t *logtype*, const char * *format*, va_list *ap*)

Generates a log message.

The message will be sent in the stream defined by the previous call to rte_openlog_stream().

The level argument determines if the log should be displayed or not, depending on the global rte_logs variable. A trailing newline may be added if needed.

The preferred alternative is the RTE_LOG() because debug logs may be removed at compilation time.

Parameters

level	Log level. A value between RTE_LOG_EMERG (1) and RTE_LOG_DEBUG (8).
logtype	The log type, for example, RTE_LOGTYPE_EAL.

format	The format string, as in printf(3), followed by the variable arguments required by the format.
ap	The va_list of the variable arguments required by the format.

Returns

- 0: Success.
- Negative on error.

3.24.4 Variable Documentation

3.24.4.1 struct rte_logs rte_logs

Global log informations

3.24.4.2 FILE* eal_default_log_stream

The default log stream.

3.25 rte_lpm.h File Reference

Data Structures

- struct rte_lpm_tbl24_entry
- struct rte_lpm_tbl8_entry
- struct rte_lpm_rule
- struct rte_lpm_rule_info
- struct rte_lpm

Defines

- #define RTE_LPM_NAMESIZE
- #define RTE_LPM_HEAP
- #define RTE_LPM_MEMZONE
- #define RTE_LPM_MAX_DEPTH
- #define RTE_LPM_LOOKUP_SUCCESS
- #define rte_lpm_lookup_bulk(lpm, ips, next_hops, n)

Functions

- struct rte_lpm ∗ rte_lpm_create (const char ∗name, int socket_id, int max_rules, int flags)
- struct rte_lpm ∗ rte_lpm_find_existing (const char ∗name)
- void rte_lpm_free (struct rte_lpm ∗lpm)
- int rte_lpm_add (struct rte_lpm ∗lpm, uint32_t ip, uint8_t depth, uint8_t next_hop)
- int rte_lpm_delete (struct rte_lpm ∗lpm, uint32_t ip, uint8_t depth)
- void rte_lpm_delete_all (struct rte_lpm ∗lpm)
- static int rte_lpm_lookup (struct rte_lpm ∗lpm, uint32_t ip, uint8_t ∗next_hop)

3.25.1 Detailed Description

RTE Longest Prefix Match (LPM)

3.25.2 Define Documentation

3.25.2.1 #define RTE_LPM_NAMESIZE

Max number of characters in LPM name.

3.25.2.2 #define RTE_LPM_HEAP

Deprecated Possible location to allocate memory. This was for last parameter of rte_lpm_create(), but is now redundant. The LPM table is always allocated in memory using librte_malloc which uses a memzone.

3.25.2.3 #define RTE_LPM_MEMZONE

Deprecated Possible location to allocate memory. This was for last parameter of rte_lpm_create(), but is now redundant. The LPM table is always allocated in memory using librte_malloc which uses a memzone.

3.25.2.4 #define RTE_LPM_MAX_DEPTH

Maximum depth value possible for IPv4 LPM.

3.25.2.5 #define RTE_LPM_LOOKUP_SUCCESS

Bitmask used to indicate successful lookup

3.25.2.6 #define rte_lpm_lookup_bulk(*lpm, ips, next_hops, n*)

Lookup multiple IP addresses in an LPM table. This may be implemented as a macro, so the address of the function should not be used.

Parameters

lpm	LPM object handle
ips	Array of IPs to be looked up in the LPM table
next_hops	Next hop of the most specific rule found for IP (valid on lookup hit only). This is an array of two byte values. The most significant byte in each value says whether the lookup was successful (bitmask RTE_LPM_LOOKUP_SUCCESS is set). The least significant byte is the actual next hop.
n	Number of elements in ips (and next_hops) array to lookup. This should be a compile time constant, and divisible by 8 for best performance.

Returns

-EINVAL for incorrect arguments, otherwise 0

3.25.3 Function Documentation

3.25.3.1 struct rte_lpm* rte_lpm_create (const char * *name*, int *socket_id*, int *max_rules*, int *flags*) [read]

Create an LPM object.

Parameters

name	LPM object name
socket_id	NUMA socket ID for LPM table memory allocation
max_rules	Maximum number of LPM rules that can be added
flags	This parameter is currently unused

Returns

Handle to LPM object on success, NULL otherwise with rte_errno set to an appropriate values. 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
- E_RTE_NO_TAILQ - no tailq list could be got for the lpm object list
- EINVAL - invalid parameter passed to function
- 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

3.25.3.2 struct rte_lpm* rte_lpm_find_existing (const char * *name*) [read]

Find an existing LPM object and return a pointer to it.

Parameters

name	Name of the lpm object as passed to rte_lpm_create()

Returns

Pointer to lpm object or NULL if object not found with rte_errno set appropriately. Possible rte_errno values include:

- ENOENT - required entry not available to return.

3.25.3.3 void rte_lpm_free (struct rte_lpm * *lpm*)

Free an LPM object.

Parameters

lpm	LPM object handle

Returns

None

3.25.3.4 int rte_lpm_add (struct rte_lpm * *lpm*, uint32_t *ip*, uint8_t *depth*, uint8_t *next_hop*)

Add a rule to the LPM table.

Parameters

lpm	LPM object handle
ip	IP of the rule to be added to the LPM table
depth	Depth of the rule to be added to the LPM table
next_hop	Next hop of the rule to be added to the LPM table

Returns

0 on success, negative value otherwise

3.25.3.5 int rte_lpm_delete (struct rte_lpm * *lpm*, uint32_t *ip*, uint8_t *depth*)

Delete a rule from the LPM table.

Parameters

lpm	LPM object handle
ip	IP of the rule to be deleted from the LPM table
depth	Depth of the rule to be deleted from the LPM table

Returns

0 on success, negative value otherwise

3.25.3.6 void rte_lpm_delete_all (struct rte_lpm * *lpm*)

Delete all rules from the LPM table.

Parameters

lpm	LPM object handle

3.25.3.7 static int rte_lpm_lookup (struct rte_lpm * *lpm*, uint32_t *ip*, uint8_t * *next_hop*) [static]

Lookup an IP into the LPM table.

Parameters

lpm	LPM object handle
ip	IP to be looked up in the LPM table
next_hop	Next hop of the most specific rule found for IP (valid on lookup hit only)

Returns

-EINVAL for incorrect arguments, -ENOENT on lookup miss, 0 on lookup hit

3.26 rte_lpm6.h File Reference

Data Structures

- struct rte_lpm6_config

Defines

- #define RTE_LPM6_NAMESIZE

Functions

- struct rte_lpm6 * rte_lpm6_create (const char *name, int socket_id, const struct rte_lpm6_config *config)
- struct rte_lpm6 * rte_lpm6_find_existing (const char *name)
- void rte_lpm6_free (struct rte_lpm6 *lpm)
- int rte_lpm6_add (struct rte_lpm6 *lpm, uint8_t *ip, uint8_t depth, uint8_t next_hop)
- int rte_lpm6_delete (struct rte_lpm6 *lpm, uint8_t *ip, uint8_t depth)
- int rte_lpm6_delete_bulk_func (struct rte_lpm6 *lpm, uint8_t ips[][RTE_LPM6_IPV6_ADDR_SIZE], uint8_t *depths, unsigned n)
- void rte_lpm6_delete_all (struct rte_lpm6 *lpm)
- int rte_lpm6_lookup (const struct rte_lpm6 *lpm, uint8_t *ip, uint8_t *next_hop)
- int rte_lpm6_lookup_bulk_func (const struct rte_lpm6 *lpm, uint8_t ips[][RTE_LPM6_IPV6_ADDR_SIZE], int16_t *next_hops, unsigned n)

3.26.1 Detailed Description

RTE Longest Prefix Match for IPv6 (LPM6)

3.26.2 Define Documentation

3.26.2.1 #define RTE_LPM6_NAMESIZE

Max number of characters in LPM name.

3.26.3 Function Documentation

3.26.3.1 struct rte_lpm6* rte_lpm6_create (const char * *name*, int *socket_id*, const struct rte_lpm6_config * *config*) [read]

Create an LPM object.

Parameters

name	LPM object name
socket_id	NUMA socket ID for LPM table memory allocation
config	Structure containing the configuration

Returns

Handle to LPM object on success, NULL otherwise with rte_errno set to an appropriate values. 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

- E_RTE_NO_TAILQ - no tailq list could be got for the lpm object list
- EINVAL - invalid parameter passed to function
- 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

3.26.3.2 struct rte_lpm6* rte_lpm6_find_existing (const char * *name*) [read]

Find an existing LPM object and return a pointer to it.

Parameters

name	Name of the lpm object as passed to rte_lpm6_create()

Returns

Pointer to lpm object or NULL if object not found with rte_errno set appropriately. Possible rte_errno values include:

- ENOENT - required entry not available to return.

3.26.3.3 void rte_lpm6_free (struct rte_lpm6 * *lpm*)

Free an LPM object.

Parameters

lpm	LPM object handle

Returns

None

3.26.3.4 int rte_lpm6_add (struct rte_lpm6 * *lpm*, uint8_t * *ip*, uint8_t *depth*, uint8_t *next_hop*)

Add a rule to the LPM table.

Parameters

lpm	LPM object handle
ip	IP of the rule to be added to the LPM table
depth	Depth of the rule to be added to the LPM table
next_hop	Next hop of the rule to be added to the LPM table

Returns

0 on success, negative value otherwise

3.26.3.5 int rte_lpm6_delete (struct rte_lpm6 * *lpm*, uint8_t * *ip*, uint8_t *depth*)

Delete a rule from the LPM table.

Parameters

lpm	LPM object handle
ip	IP of the rule to be deleted from the LPM table
depth	Depth of the rule to be deleted from the LPM table

Returns

0 on success, negative value otherwise

3.26.3.6 int rte_lpm6_delete_bulk_func (struct rte_lpm6 * *lpm*, uint8_t *ips[][RTE_LPM6_IPV6_ADDR_SIZE]*, uint8_t * *depths*, unsigned *n*)

Delete a rule from the LPM table.

Parameters

lpm	LPM object handle
ips	Array of IPs to be deleted from the LPM table
depths	Array of depths of the rules to be deleted from the LPM table
n	Number of rules to be deleted from the LPM table

Returns

0 on success, negative value otherwise.

3.26.3.7 void rte_lpm6_delete_all (struct rte_lpm6 * *lpm*)

Delete all rules from the LPM table.

Parameters

lpm	LPM object handle

3.26.3.8 int rte_lpm6_lookup (const struct rte_lpm6 * *lpm*, uint8_t * *ip*, uint8_t * *next_hop*)

Lookup an IP into the LPM table.

Parameters

lpm	LPM object handle
ip	IP to be looked up in the LPM table
next_hop	Next hop of the most specific rule found for IP (valid on lookup hit only)

Returns

-EINVAL for incorrect arguments, -ENOENT on lookup miss, 0 on lookup hit

3.26.3.9 int rte_lpm6_lookup_bulk_func (const struct rte_lpm6 * *lpm*, uint8_t *ips[][RTE_LPM6_IPV6_ADDR_SIZE]*, int16_t * *next_hops*, unsigned *n*)

Lookup multiple IP addresses in an LPM table.

Parameters

lpm	LPM object handle
ips	Array of IPs to be looked up in the LPM table
next_hops	Next hop of the most specific rule found for IP (valid on lookup hit only). This is an array of two byte values. The next hop will be stored on each position on success; otherwise the position will be set to -1.
n	Number of elements in ips (and next_hops) array to lookup.

Returns

-EINVAL for incorrect arguments, otherwise 0

3.27 rte_malloc.h File Reference

Data Structures

- struct rte_malloc_socket_stats

Functions

- void * rte_malloc (const char *type, size_t size, unsigned align)
- void * rte_zmalloc (const char *type, size_t size, unsigned align)
- void * rte_calloc (const char *type, size_t num, size_t size, unsigned align)
- void * rte_realloc (void *ptr, size_t size, unsigned align)

- void * rte_malloc_socket (const char *type, size_t size, unsigned align, int socket)
- void * rte_zmalloc_socket (const char *type, size_t size, unsigned align, int socket)
- void * rte_calloc_socket (const char *type, size_t num, size_t size, unsigned align, int socket)
- void rte_free (void *ptr)
- int rte_malloc_validate (void *ptr, size_t *size)
- int rte_malloc_get_socket_stats (int socket, struct rte_malloc_socket_stats *socket_stats)
- void rte_malloc_dump_stats (const char *type)
- int rte_malloc_set_limit (const char *type, size_t max)

3.27.1 Detailed Description

RTE Malloc. This library provides methods for dynamically allocating memory from hugepages.

3.27.2 Function Documentation

3.27.2.1 void* rte_malloc (const char * *type*, size_t *size*, unsigned *align*)

This function allocates memory from the huge-page area of memory. The memory is not cleared. In NUMA systems, the memory allocated resides on the same NUMA socket as the core that calls this function.

Parameters

type	A string identifying the type of allocated objects (useful for debug purposes, such as identifying the cause of a memory leak). Can be NULL.
size	Size (in bytes) to be allocated.
align	If 0, the return is a pointer that is suitably aligned for any kind of variable (in the same manner as malloc()). Otherwise, the return is a pointer that is a multiple of *align*. - In this case, it must be a power of two. (Minimum alignment is the cacheline size, i.e. 64-bytes)

Returns

- NULL on error. Not enough memory, or invalid arguments (size is 0, align is not a power of two).
- Otherwise, the pointer to the allocated object.

3.27.2.2 void* rte_zmalloc (const char * *type*, size_t *size*, unsigned *align*)

Allocate zero'ed memory from the heap.

Equivalent to rte_malloc() except that the memory zone is initialised with zeros. In NUMA systems, the memory allocated resides on the same NUMA socket as the core that calls this function.

Parameters

type	A string identifying the type of allocated objects (useful for debug purposes, such as identifying the cause of a memory leak). Can be NULL.
size	Size (in bytes) to be allocated.
align	If 0, the return is a pointer that is suitably aligned for any kind of variable (in the same manner as malloc()). Otherwise, the return is a pointer that is a multiple of *align*. In this case, it must obviously be a power of two. (Minimum alignment is the cacheline size, i.e. 64-bytes)

Returns

- NULL on error. Not enough memory, or invalid arguments (size is 0, align is not a power of two).
- Otherwise, the pointer to the allocated object.

3.27.2.3 void* rte_calloc (const char * *type*, size_t *num*, size_t *size*, unsigned *align*)

Replacement function for calloc(), using huge-page memory. Memory area is initialised with zeros. In NUMA systems, the memory allocated resides on the same NUMA socket as the core that calls this function.

Parameters

type	A string identifying the type of allocated objects (useful for debug purposes, such as identifying the cause of a memory leak). Can be NULL.
num	Number of elements to be allocated.
size	Size (in bytes) of a single element.
align	If 0, the return is a pointer that is suitably aligned for any kind of variable (in the same manner as malloc()). Otherwise, the return is a pointer that is a multiple of *align*. In this case, it must obviously be a power of two. (Minimum alignment is the cacheline size, i.e. 64-bytes)

Returns

- NULL on error. Not enough memory, or invalid arguments (size is 0, align is not a power of two).
- Otherwise, the pointer to the allocated object.

3.27.2.4 void* rte_realloc (void * *ptr*, size_t *size*, unsigned *align*)

Replacement function for realloc(), using huge-page memory. Reserved area memory is resized, preserving contents. In NUMA systems, the new area resides on the same NUMA socket as the old area.

Parameters

ptr	Pointer to already allocated memory
size	Size (in bytes) of new area. If this is 0, memory is freed.

align	If 0, the return is a pointer that is suitably aligned for any kind of variable (in the same manner as malloc()). Otherwise, the return is a pointer that is a multiple of *align*. In this case, it must obviously be a power of two. (Minimum alignment is the cacheline size, i.e. 64-bytes)

Returns

- NULL on error. Not enough memory, or invalid arguments (size is 0, align is not a power of two).
- Otherwise, the pointer to the reallocated memory.

3.27.2.5 void* rte_malloc_socket (const char * *type*, size_t *size*, unsigned *align*, int *socket*)

This function allocates memory from the huge-page area of memory. The memory is not cleared.

Parameters

type	A string identifying the type of allocated objects (useful for debug purposes, such as identifying the cause of a memory leak). Can be NULL.
size	Size (in bytes) to be allocated.
align	If 0, the return is a pointer that is suitably aligned for any kind of variable (in the same manner as malloc()). Otherwise, the return is a pointer that is a multiple of *align*. - In this case, it must be a power of two. (Minimum alignment is the cacheline size, i.e. 64-bytes)
socket	NUMA socket to allocate memory on. If SOCKET_ID_ANY is used, this function will behave the same as rte_malloc().

Returns

- NULL on error. Not enough memory, or invalid arguments (size is 0, align is not a power of two).
- Otherwise, the pointer to the allocated object.

3.27.2.6 void* rte_zmalloc_socket (const char * *type*, size_t *size*, unsigned *align*, int *socket*)

Allocate zero'ed memory from the heap.

Equivalent to rte_malloc() except that the memory zone is initialised with zeros.

Parameters

type	A string identifying the type of allocated objects (useful for debug purposes, such as identifying the cause of a memory leak). Can be NULL.
size	Size (in bytes) to be allocated.
align	If 0, the return is a pointer that is suitably aligned for any kind of variable (in the same manner as malloc()). Otherwise, the return is a pointer that is a multiple of *align*. In this case, it must obviously be a power of two. (Minimum alignment is the cacheline size, i.e. 64-bytes)

socket	NUMA socket to allocate memory on. If SOCKET_ID_ANY is used, this function will behave the same as rte_zmalloc().

Returns

- NULL on error. Not enough memory, or invalid arguments (size is 0, align is not a power of two).
- Otherwise, the pointer to the allocated object.

3.27.2.7 void* rte_calloc_socket (const char * *type*, size_t *num*, size_t *size*, unsigned *align*, int *socket*)

Replacement function for calloc(), using huge-page memory. Memory area is initialised with zeros.

Parameters

type	A string identifying the type of allocated objects (useful for debug purposes, such as identifying the cause of a memory leak). Can be NULL.
num	Number of elements to be allocated.
size	Size (in bytes) of a single element.
align	If 0, the return is a pointer that is suitably aligned for any kind of variable (in the same manner as malloc()). Otherwise, the return is a pointer that is a multiple of *align*. In this case, it must obviously be a power of two. (Minimum alignment is the cacheline size, i.e. 64-bytes)
socket	NUMA socket to allocate memory on. If SOCKET_ID_ANY is used, this function will behave the same as rte_calloc().

Returns

- NULL on error. Not enough memory, or invalid arguments (size is 0, align is not a power of two).
- Otherwise, the pointer to the allocated object.

3.27.2.8 void rte_free (void * *ptr*)

Frees the memory space pointed to by the provided pointer.

This pointer must have been returned by a previous call to rte_malloc(), rte_zmalloc(), rte_calloc() or rte_realloc(). The behaviour of rte_free() is undefined if the pointer does not match this requirement.

If the pointer is NULL, the function does nothing.

Parameters

ptr	The pointer to memory to be freed.

3.27.2.9 int rte_malloc_validate (void * *ptr*, size_t * *size*)

If malloc debug is enabled, check a memory block for header and trailer markers to indicate that all is well with the block. If size is non-null, also return the size of the block.

Parameters

ptr	pointer to the start of a data block, must have been returned by a previous call to rte_malloc(), rte_zmalloc(), rte_calloc() or rte_realloc()
size	if non-null, and memory block pointer is valid, returns the size of the memory block

Returns

-1 on error, invalid pointer passed or header and trailer markers are missing or corrupted 0 on success

3.27.2.10 int rte_malloc_get_socket_stats (int *socket*, struct rte_malloc_socket_stats * *socket_stats*)

Get heap statistics for the specified heap.

Parameters

socket	An unsigned integer specifying the socket to get heap statistics for
socket_stats	A structure which provides memory to store statistics

Returns

Null on error Pointer to structure storing statistics on success

3.27.2.11 void rte_malloc_dump_stats (const char * *type*)

Dump statistics.

Dump for the specified type to the console. If the type argument is NULL, all memory types will be dumped.

Parameters

type	A string identifying the type of objects to dump, or NULL to dump all objects.

3.27.2.12 int rte_malloc_set_limit (const char * *type*, size_t *max*)

Set the maximum amount of allocated memory for this type.

This is not yet implemented

Parameters

type	A string identifying the type of allocated objects.
max	The maximum amount of allocated bytes for this type.

Returns

- 0: Success.
- (-1): Error.

3.28 rte_mbuf.h File Reference

Data Structures

- struct rte_ctrlmbuf
- union rte_vlan_macip
- struct rte_pktmbuf
- struct rte_mbuf
- struct rte_pktmbuf_pool_private

Defines

- #define PKT_RX_VLAN_PKT
- #define PKT_RX_RSS_HASH
- #define PKT_RX_FDIR
- #define PKT_RX_L4_CKSUM_BAD
- #define PKT_RX_IP_CKSUM_BAD
- #define PKT_RX_IPV4_HDR
- #define PKT_RX_IPV4_HDR_EXT
- #define PKT_RX_IPV6_HDR
- #define PKT_RX_IPV6_HDR_EXT
- #define PKT_RX_IEEE1588_PTP
- #define PKT_RX_IEEE1588_TMST
- #define PKT_TX_VLAN_PKT
- #define PKT_TX_IP_CKSUM
- #define PKT_TX_L4_MASK
- #define PKT_TX_L4_NO_CKSUM
- #define PKT_TX_TCP_CKSUM
- #define PKT_TX_SCTP_CKSUM
- #define PKT_TX_UDP_CKSUM
- #define PKT_TX_IEEE1588_TMST
- #define PKT_TX_OFFLOAD_MASK
- #define TX_VLAN_CMP_MASK
- #define TX_MAC_LEN_CMP_MASK

- #define TX_IP_LEN_CMP_MASK
- #define RTE_MBUF_FROM_BADDR(ba)
- #define RTE_MBUF_TO_BADDR(mb)
- #define RTE_MBUF_INDIRECT(mb)
- #define RTE_MBUF_DIRECT(mb)
- #define __rte_mbuf_sanity_check(m, t, is_h)
- #define __rte_mbuf_sanity_check_raw(m, t, is_h)
- #define RTE_MBUF_ASSERT(exp)
- #define RTE_MBUF_PREFETCH_TO_FREE(m)
- #define rte_ctrlmbuf_data(m)
- #define rte_ctrlmbuf_len(m)
- #define rte_pktmbuf_mtod(m, t)
- #define rte_pktmbuf_pkt_len(m)
- #define rte_pktmbuf_data_len(m)

Enumerations

- enum rte_mbuf_type { RTE_MBUF_CTRL, RTE_MBUF_PKT }

Functions

- static uint16_t rte_mbuf_refcnt_update (struct rte_mbuf *m, int16_t value)
- static uint16_t rte_mbuf_refcnt_read (const struct rte_mbuf *m)
- static void rte_mbuf_refcnt_set (struct rte_mbuf *m, uint16_t new_value)
- void rte_mbuf_sanity_check (const struct rte_mbuf *m, enum rte_mbuf_type t, int is_header)
- void rte_ctrlmbuf_init (struct rte_mempool *mp, void *opaque_arg, void *m, unsigned i)
- static struct rte_mbuf * rte_ctrlmbuf_alloc (struct rte_mempool *mp)
- static void rte_ctrlmbuf_free (struct rte_mbuf *m)
- void rte_pktmbuf_init (struct rte_mempool *mp, void *opaque_arg, void *m, unsigned i)
- void rte_pktmbuf_pool_init (struct rte_mempool *mp, void *opaque_arg)
- static void rte_pktmbuf_reset (struct rte_mbuf *m)
- static struct rte_mbuf * rte_pktmbuf_alloc (struct rte_mempool *mp)
- static void rte_pktmbuf_attach (struct rte_mbuf *mi, struct rte_mbuf *md)
- static void rte_pktmbuf_detach (struct rte_mbuf *m)
- static void rte_pktmbuf_free_seg (struct rte_mbuf *m)
- static void rte_pktmbuf_free (struct rte_mbuf *m)
- static struct rte_mbuf * rte_pktmbuf_clone (struct rte_mbuf *md, struct rte_mempool *mp)
- static void rte_pktmbuf_refcnt_update (struct rte_mbuf *m, int16_t v)
- static uint16_t rte_pktmbuf_headroom (const struct rte_mbuf *m)
- static uint16_t rte_pktmbuf_tailroom (const struct rte_mbuf *m)
- static struct rte_mbuf * rte_pktmbuf_lastseg (struct rte_mbuf *m)
- static char * rte_pktmbuf_prepend (struct rte_mbuf *m, uint16_t len)
- static char * rte_pktmbuf_append (struct rte_mbuf *m, uint16_t len)
- static char * rte_pktmbuf_adj (struct rte_mbuf *m, uint16_t len)
- static int rte_pktmbuf_trim (struct rte_mbuf *m, uint16_t len)
- static int rte_pktmbuf_is_contiguous (const struct rte_mbuf *m)
- void rte_pktmbuf_dump (const struct rte_mbuf *m, unsigned dump_len)

3.28.1 Detailed Description

RTE Mbuf

The mbuf library provides the ability to create and destroy buffers that may be used by the RTE application to store message buffers. The message buffers are stored in a mempool, using the RTE mempool library.

This library provide an API to allocate/free mbufs, manipulate control message buffer (ctrlmbuf), which are generic message buffers, and packet buffers (pktmbuf), which are used to carry network packets.

To understand the concepts of packet buffers or mbufs, you should read "TCP/IP Illustrated, Volume 2: The Implementation, Addison-Wesley, 1995, ISBN 0-201-63354-X from Richard Stevens" http://www.-kohala.com/start/tcpipiv2.html

The main modification of this implementation is the use of mbuf for transports other than packets. mbufs can have other types.

3.28.2 Define Documentation

3.28.2.1 #define PKT_RX_VLAN_PKT

RX packet is a 802.1q VLAN packet.

3.28.2.2 #define PKT_RX_RSS_HASH

RX packet with RSS hash result.

3.28.2.3 #define PKT_RX_FDIR

RX packet with FDIR infos.

3.28.2.4 #define PKT_RX_L4_CKSUM_BAD

L4 cksum of RX pkt. is not OK.

3.28.2.5 #define PKT_RX_IP_CKSUM_BAD

IP cksum of RX pkt. is not OK.

3.28.2.6 #define PKT_RX_IPV4_HDR

RX packet with IPv4 header.

3.28.2.7 #define PKT_RX_IPV4_HDR_EXT

RX packet with extended IPv4 header.

3.28.2.8 #define PKT_RX_IPV6_HDR

RX packet with IPv6 header.

3.28.2.9 #define PKT_RX_IPV6_HDR_EXT

RX packet with extended IPv6 header.

3.28.2.10 #define PKT_RX_IEEE1588_PTP

RX IEEE1588 L2 Ethernet PT Packet.

3.28.2.11 #define PKT_RX_IEEE1588_TMST

RX IEEE1588 L2/L4 timestamped packet.

3.28.2.12 #define PKT_TX_VLAN_PKT

TX packet is a 802.1q VLAN packet.

3.28.2.13 #define PKT_TX_IP_CKSUM

IP cksum of TX pkt. computed by NIC.

3.28.2.14 #define PKT_TX_L4_MASK

Mask bits for L4 checksum offload request.

3.28.2.15 #define PKT_TX_L4_NO_CKSUM

Disable L4 cksum of TX pkt.

3.28.2.16 #define PKT_TX_TCP_CKSUM

TCP cksum of TX pkt. computed by NIC.

3.28.2.17 #define PKT_TX_SCTP_CKSUM

SCTP cksum of TX pkt. computed by NIC.

3.28.2.18 #define PKT_TX_UDP_CKSUM

UDP cksum of TX pkt. computed by NIC.

3.28.2.19 #define PKT_TX_IEEE1588_TMST

TX IEEE1588 packet to timestamp.

3.28.2.20 #define PKT_TX_OFFLOAD_MASK

Bit Mask to indicate what bits required for building TX context

3.28.2.21 #define TX_VLAN_CMP_MASK

VLAN length - 16-bits.

3.28.2.22 #define TX_MAC_LEN_CMP_MASK

MAC length - 7-bits.

3.28.2.23 #define TX_IP_LEN_CMP_MASK

IP length - 9-bits. MAC+IP length.

3.28.2.24 #define RTE_MBUF_FROM_BADDR(*ba*)

Given the buf_addr returns the pointer to corresponding mbuf.

3.28.2.25 #define RTE_MBUF_TO_BADDR(*mb*)

Given the pointer to mbuf returns an address where it's buf_addr should point to.

3.28.2.26 #define RTE_MBUF_INDIRECT(*mb*)

Returns TRUE if given mbuf is indirect, or FALSE otherwise.

3.28.2.27 #define RTE_MBUF_DIRECT(*mb*)

Returns TRUE if given mbuf is direct, or FALSE otherwise.

3.28.2.28 #define __rte_mbuf_sanity_check(*m*, *t*, *is_h*)

check mbuf type in debug mode

3.28.2.29 #define __rte_mbuf_sanity_check_raw(*m*, *t*, *is_h*)

check mbuf type in debug mode if mbuf pointer is not null

3.28.2.30 #define RTE_MBUF_ASSERT(*exp*)

MBUF asserts in debug mode

3.28.2.31 #define RTE_MBUF_PREFETCH_TO_FREE(*m*)

Mbuf prefetch

3.28.2.32 #define rte_ctrlmbuf_data(*m*)

A macro that returns the pointer to the carried data.

The value that can be read or assigned.

Parameters

m	The control mbuf.

3.28.2.33 #define rte_ctrlmbuf_len(*m*)

A macro that returns the length of the carried data.

The value that can be read or assigned.

Parameters

m	The control mbuf.

3.28.2.34 #define rte_pktmbuf_mtod(m, t)

A macro that points to the start of the data in the mbuf.

The returned pointer is cast to type t. Before using this function, the user must ensure that m_headlen(m) is large enough to read its data.

Parameters

m	The packet mbuf.
t	The type to cast the result into.

3.28.2.35 #define rte_pktmbuf_pkt_len(m)

A macro that returns the length of the packet.

The value can be read or assigned.

Parameters

m	The packet mbuf.

3.28.2.36 #define rte_pktmbuf_data_len(m)

A macro that returns the length of the segment.

The value can be read or assigned.

Parameters

m	The packet mbuf.

3.28.3 Enumeration Type Documentation

3.28.3.1 enum rte_mbuf_type

This enum indicates the mbuf type.

Enumerator:

RTE_MBUF_CTRL Control mbuf.

RTE_MBUF_PKT Packet mbuf.

3.28.4 Function Documentation

3.28.4.1 static uint16_t rte_mbuf_refcnt_update (struct rte_mbuf * m, int16_t value) [static]

Adds given value to an mbuf's refcnt and returns its new value.

Parameters

m	Mbuf to update
value	Value to add/subtract

Returns

Updated value

3.28.4.2 static uint16_t rte_mbuf_refcnt_read (const struct rte_mbuf * m) [static]

Reads the value of an mbuf's refcnt.

Parameters

m	Mbuf to read

Returns

Reference count number.

3.28.4.3 static void rte_mbuf_refcnt_set (struct rte_mbuf * m, uint16_t new_value) [static]

Sets an mbuf's refcnt to a defined value.

Parameters

m	Mbuf to update
new_value	Value set

3.28.4.4 void rte_mbuf_sanity_check (const struct rte_mbuf * m, enum rte_mbuf_type t, int is_header)

Sanity checks on an mbuf.

Check the consistency of the given mbuf. The function will cause a panic if corruption is detected.

Parameters

m	The mbuf to be checked.
t	The expected type of the mbuf.
is_header	True if the mbuf is a packet header, false if it is a sub-segment of a packet (in this case, some fields like nb_segs are not checked)

3.28.4.5 void rte_ctrlmbuf_init (struct rte_mempool * mp, void * opaque_arg, void * m, unsigned i)

The control mbuf constructor.

This function initializes some fields in an mbuf structure that are not modified by the user once created (mbuf type, origin pool, buffer start address, and so on). This function is given as a callback function to rte_mempool_create() at pool creation time.

Parameters

mp	The mempool from which the mbuf is allocated.
opaque_arg	A pointer that can be used by the user to retrieve useful information for mbuf initialization. This pointer comes from the “init_arg“ parameter of rte_mempool_create().
m	The mbuf to initialize.
i	The index of the mbuf in the pool table.

3.28.4.6 static struct rte_mbuf* rte_ctrlmbuf_alloc (struct rte_mempool * mp) [static, read]

Allocate a new mbuf (type is ctrl) from mempool *mp*.

This new mbuf is initialized with data pointing to the beginning of buffer, and with a length of zero.

Parameters

mp	The mempool from which the mbuf is allocated.

Returns

- The pointer to the new mbuf on success.
- NULL if allocation failed.

3.28.4.7 static void rte_ctrlmbuf_free (struct rte_mbuf * m) [static]

Free a control mbuf back into its original mempool.

Parameters

m	The control mbuf to be freed.

3.28.4.8 void rte_pktmbuf_init (struct rte_mempool * mp, void * opaque_arg, void * m, unsigned i)

The packet mbuf constructor.

This function initializes some fields in the mbuf structure that are not modified by the user once created (mbuf type, origin pool, buffer start address, and so on). This function is given as a callback function to rte_mempool_create() at pool creation time.

Parameters

mp	The mempool from which mbufs originate.
opaque_arg	A pointer that can be used by the user to retrieve useful information for mbuf initialization. This pointer comes from the “init_arg“ parameter of rte_mempool_create().
m	The mbuf to initialize.
i	The index of the mbuf in the pool table.

3.28.4.9 void rte_pktmbuf_pool_init (struct rte_mempool * *mp*, void * *opaque_arg*)

A packet mbuf pool constructor.

This function initializes the mempool private data in the case of a pktmbuf pool. This private data is needed by the driver. The function is given as a callback function to rte_mempool_create() at pool creation. It can be extended by the user, for example, to provide another packet size.

Parameters

mp	The mempool from which mbufs originate.
opaque_arg	A pointer that can be used by the user to retrieve useful information for mbuf initialization. This pointer comes from the “init_arg“ parameter of rte_mempool_create().

3.28.4.10 static void rte_pktmbuf_reset (struct rte_mbuf * *m*) [static]

Reset the fields of a packet mbuf to their default values.

The given mbuf must have only one segment.

Parameters

m	The packet mbuf to be resetted.

3.28.4.11 static struct rte_mbuf* rte_pktmbuf_alloc (struct rte_mempool * *mp*) [static, read]

Allocate a new mbuf (type is pkt) from a mempool.

This new mbuf contains one segment, which has a length of 0. The pointer to data is initialized to have some bytes of headroom in the buffer (if buffer size allows).

Parameters

mp	The mempool from which the mbuf is allocated.

Returns

- The pointer to the new mbuf on success.
- NULL if allocation failed.

3.28.4.12 static void rte_pktmbuf_attach (struct rte_mbuf * mi, struct rte_mbuf * md) [static]

Attach packet mbuf to another packet mbuf. After attachment we refer the mbuf we attached as 'indirect', while mbuf we attached to as 'direct'. Right now, not supported:

- attachment to indirect mbuf (e.g. - md has to be direct).
- attachment for already indirect mbuf (e.g. - mi has to be direct).
- mbuf we trying to attach (mi) is used by someone else e.g. it's reference counter is greater then 1.

Parameters

mi	The indirect packet mbuf.
md	The direct packet mbuf.

3.28.4.13 static void rte_pktmbuf_detach (struct rte_mbuf * m) [static]

Detach an indirect packet mbuf -

- restore original mbuf address and length values.
- reset pktmbuf data and data_len to their default values. All other fields of the given packet mbuf will be left intact.

Parameters

m	The indirect attached packet mbuf.

3.28.4.14 static void rte_pktmbuf_free_seg (struct rte_mbuf * m) [static]

Free a segment of a packet mbuf into its original mempool.

Free an mbuf, without parsing other segments in case of chained buffers.

Parameters

m	The packet mbuf segment to be freed.

3.28.4.15 static void rte_pktmbuf_free (struct rte_mbuf * m) [static]

Free a packet mbuf back into its original mempool.

Free an mbuf, and all its segments in case of chained buffers. Each segment is added back into its original mempool.

Parameters

m	The packet mbuf to be freed.

3.28.4.16 static struct rte_mbuf* rte_pktmbuf_clone (struct rte_mbuf * *md*, struct rte_mempool * *mp*) [static, read]

Creates a "clone" of the given packet mbuf.

Walks through all segments of the given packet mbuf, and for each of them:

- Creates a new packet mbuf from the given pool.
- Attaches newly created mbuf to the segment. Then updates pkt_len and nb_segs of the "clone" packet mbuf to match values from the original packet mbuf.

Parameters

md	The packet mbuf to be cloned.
mp	The mempool from which the "clone" mbufs are allocated.

Returns

- The pointer to the new "clone" mbuf on success.
- NULL if allocation fails.

3.28.4.17 static void rte_pktmbuf_refcnt_update (struct rte_mbuf * *m*, int16_t *v*) [static]

Adds given value to the refcnt of all packet mbuf segments.

Walks through all segments of given packet mbuf and for each of them invokes rte_mbuf_refcnt_update().

Parameters

m	The packet mbuf whose refcnt to be updated.
v	The value to add to the mbuf's segments refcnt.

3.28.4.18 static uint16_t rte_pktmbuf_headroom (const struct rte_mbuf * *m*) [static]

Get the headroom in a packet mbuf.

Parameters

m	The packet mbuf.

Returns

The length of the headroom.

3.28.4.19 static uint16_t rte_pktmbuf_tailroom (const struct rte_mbuf * m) [static]

Get the tailroom of a packet mbuf.

Parameters

m	The packet mbuf.

Returns

The length of the tailroom.

3.28.4.20 static struct rte_mbuf* rte_pktmbuf_lastseg (struct rte_mbuf * m) [static, read]

Get the last segment of the packet.

Parameters

m	The packet mbuf.

Returns

The last segment of the given mbuf.

3.28.4.21 static char* rte_pktmbuf_prepend (struct rte_mbuf * m, uint16_t len) [static]

Prepend len bytes to an mbuf data area.

Returns a pointer to the new data start address. If there is not enough headroom in the first segment, the function will return NULL, without modifying the mbuf.

Parameters

m	The pkt mbuf.
len	The amount of data to prepend (in bytes).

Returns

A pointer to the start of the newly prepended data, or NULL if there is not enough headroom space in the first segment

3.28.4.22 static char* rte_pktmbuf_append (struct rte_mbuf * m, uint16_t len) [static]

Append len bytes to an mbuf.

Append len bytes to an mbuf and return a pointer to the start address of the added data. If there is not enough tailroom in the last segment, the function will return NULL, without modifying the mbuf.

Parameters

m	The packet mbuf.
len	The amount of data to append (in bytes).

Returns

A pointer to the start of the newly appended data, or NULL if there is not enough tailroom space in the last segment

3.28.4.23 static char* rte_pktmbuf_adj (struct rte_mbuf * m, uint16_t len) [static]

Remove len bytes at the beginning of an mbuf.

Returns a pointer to the start address of the new data area. If the length is greater than the length of the first segment, then the function will fail and return NULL, without modifying the mbuf.

Parameters

m	The packet mbuf.
len	The amount of data to remove (in bytes).

Returns

A pointer to the new start of the data.

3.28.4.24 static int rte_pktmbuf_trim (struct rte_mbuf * m, uint16_t len) [static]

Remove len bytes of data at the end of the mbuf.

If the length is greater than the length of the last segment, the function will fail and return -1 without modifying the mbuf.

Parameters

m	The packet mbuf.
len	The amount of data to remove (in bytes).

Returns

- 0: On success.
- -1: On error.

3.28.4.25 static int rte_pktmbuf_is_contiguous (const struct rte_mbuf * *m*) [static]

Test if mbuf data is contiguous.

Parameters

m	The packet mbuf.

Returns

- 1, if all data is contiguous (one segment).
- 0, if there is several segments.

3.28.4.26 void rte_pktmbuf_dump (const struct rte_mbuf * *m*, unsigned *dump_len*)

Dump an mbuf structure to the console.

Dump all fields for the given packet mbuf and all its associated segments (in the case of a chained buffer).

Parameters

m	The packet mbuf.
dump_len	If dump_len != 0, also dump the "dump_len" first data bytes of the packet.

3.29 rte_memcpy.h File Reference

Defines

- #define rte_memcpy(dst, src, n)

Functions

- static void rte_mov16 (uint8_t *dst, const uint8_t *src)
- static void rte_mov32 (uint8_t *dst, const uint8_t *src)
- static void rte_mov48 (uint8_t *dst, const uint8_t *src)
- static void rte_mov64 (uint8_t *dst, const uint8_t *src)
- static void rte_mov128 (uint8_t *dst, const uint8_t *src)
- static void rte_mov256 (uint8_t *dst, const uint8_t *src)

3.29.1 Detailed Description

Functions for SSE implementation of memcpy().

3.29.2 Define Documentation

3.29.2.1 #define rte_memcpy(*dst*, *src*, *n*)

Copy bytes from one location to another. The locations must not overlap.

Note

This is implemented as a macro, so it's address should not be taken and care is needed as parameter expressions may be evaluated multiple times.

Parameters

dst	Pointer to the destination of the data.
src	Pointer to the source data.
n	Number of bytes to copy.

Returns

Pointer to the destination data.

3.29.3 Function Documentation

3.29.3.1 static void rte_mov16 (uint8_t * *dst*, const uint8_t * *src*) [static]

Copy 16 bytes from one location to another using optimised SSE instructions. The locations should not overlap.

Parameters

dst	Pointer to the destination of the data.
src	Pointer to the source data.

3.29.3.2 static void rte_mov32 (uint8_t * *dst*, const uint8_t * *src*) [static]

Copy 32 bytes from one location to another using optimised SSE instructions. The locations should not overlap.

Parameters

dst	Pointer to the destination of the data.
src	Pointer to the source data.

3.29.3.3 static void rte_mov48 (uint8_t * *dst*, const uint8_t * *src*) [static]

Copy 48 bytes from one location to another using optimised SSE instructions. The locations should not overlap.

Parameters

dst	Pointer to the destination of the data.
src	Pointer to the source data.

3.29.3.4 static void rte_mov64 (uint8_t * *dst*, const uint8_t * *src*) [static]

Copy 64 bytes from one location to another using optimised SSE instructions. The locations should not overlap.

Parameters

dst	Pointer to the destination of the data.
src	Pointer to the source data.

3.29.3.5 static void rte_mov128 (uint8_t * *dst*, const uint8_t * *src*) [static]

Copy 128 bytes from one location to another using optimised SSE instructions. The locations should not overlap.

Parameters

dst	Pointer to the destination of the data.
src	Pointer to the source data.

3.29.3.6 static void rte_mov256 (uint8_t * *dst*, const uint8_t * *src*) [static]

Copy 256 bytes from one location to another using optimised SSE instructions. The locations should not overlap.

Parameters

dst	Pointer to the destination of the data.
src	Pointer to the source data.

3.30 rte_memory.h File Reference

Data Structures

- struct rte_memseg

Defines

- #define SOCKET_ID_ANY
- #define CACHE_LINE_SIZE
- #define CACHE_LINE_MASK
- #define CACHE_LINE_ROUNDUP(size)
- #define __rte_cache_aligned

Typedefs

- typedef uint64_t phys_addr_t

Functions

- struct rte_memseg * rte_eal_get_physmem_layout (void)
- void rte_dump_physmem_layout (void)
- uint64_t rte_eal_get_physmem_size (void)
- unsigned rte_memory_get_nchannel (void)
- unsigned rte_memory_get_nrank (void)

3.30.1 Detailed Description

Memory-related RTE API.

3.30.2 Define Documentation

3.30.2.1 #define SOCKET_ID_ANY

Any NUMA socket.

3.30.2.2 #define CACHE_LINE_SIZE

Cache line size.

3.30.2.3 #define CACHE_LINE_MASK

Cache line mask.

3.30.2.4 #define CACHE_LINE_ROUNDUP(*size*)

Return the first cache-aligned value greater or equal to size.

3.30.2.5 #define __rte_cache_aligned

Force alignment to cache line.

3.30.3 Typedef Documentation

3.30.3.1 typedef uint64_t phys_addr_t

Physical address definition.

3.30.4 Function Documentation

3.30.4.1 struct rte_memseg* rte_eal_get_physmem_layout (void) [read]

Get the layout of the available physical memory.

It can be useful for an application to have the full physical memory layout to decide the size of a memory zone to reserve. This table is stored in rte_config (see rte_eal_get_configuration()).

Returns

- On success, return a pointer to a read-only table of struct rte_physmem_desc elements, containing the layout of all addressable physical memory. The last element of the table contains a NULL address.
- On error, return NULL. This should not happen since it is a fatal error that will probably cause the entire system to panic.

3.30.4.2 void rte_dump_physmem_layout (void)

Dump the physical memory layout to the console.

3.30.4.3 uint64_t rte_eal_get_physmem_size (void)

Get the total amount of available physical memory.

Returns

The total amount of available physical memory in bytes.

3.30.4.4 unsigned rte_memory_get_nchannel (void)

Get the number of memory channels.

Returns

The number of memory channels on the system. The value is 0 if unknown or not the same on all devices.

3.30.4.5 unsigned rte_memory_get_nrank (void)

Get the number of memory ranks.

Returns

The number of memory ranks on the system. The value is 0 if unknown or not the same on all devices.

3.31 rte_mempool.h File Reference

Data Structures

- struct rte_mempool_cache
- struct rte_mempool_objsz
- struct rte_mempool

Defines

- #define RTE_MEMPOOL_HEADER_COOKIE1
- #define RTE_MEMPOOL_HEADER_COOKIE2
- #define RTE_MEMPOOL_TRAILER_COOKIE
- #define RTE_MEMPOOL_NAMESIZE
- #define MEMPOOL_PG_NUM_DEFAULT
- #define MEMPOOL_F_NO_SPREAD
- #define MEMPOOL_F_NO_CACHE_ALIGN
- #define MEMPOOL_F_SP_PUT
- #define MEMPOOL_F_SC_GET
- #define MEMPOOL_HEADER_SIZE(mp, pgn)
- #define MEMPOOL_IS_CONTIG(mp)

Typedefs

- typedef void(* rte_mempool_obj_iter_t)(void *, void *, void *, uint32_t)
- typedef void(rte_mempool_obj_ctor_t)(struct rte_mempool *, void *, void *, unsigned)
- typedef void(rte_mempool_ctor_t)(struct rte_mempool *, void *)

Functions

- static struct rte_mempool * rte_mempool_from_obj (void *obj)
- struct rte_mempool * rte_mempool_create (const char *name, unsigned n, unsigned elt_size, unsigned cache_size, unsigned private_data_size, rte_mempool_ctor_t *mp_init, void *mp_init_arg, rte_mempool_obj_ctor_t *obj_init, void *obj_init_arg, int socket_id, unsigned flags)
- struct rte_mempool * rte_mempool_xmem_create (const char *name, unsigned n, unsigned elt_size, unsigned cache_size, unsigned private_data_size, rte_mempool_ctor_t *mp_init, void *mp_init_arg, rte_mempool_obj_ctor_t *obj_init, void *obj_init_arg, int socket_id, unsigned flags, void *vaddr, const phys_addr_t paddr[], uint32_t pg_num, uint32_t pg_shift)
- void rte_mempool_dump (const struct rte_mempool *mp)
- static void rte_mempool_mp_put_bulk (struct rte_mempool *mp, void *const *obj_table, unsigned n)
- static void rte_mempool_sp_put_bulk (struct rte_mempool *mp, void *const *obj_table, unsigned n)
- static void rte_mempool_put_bulk (struct rte_mempool *mp, void *const *obj_table, unsigned n)
- static void rte_mempool_mp_put (struct rte_mempool *mp, void *obj)
- static void rte_mempool_sp_put (struct rte_mempool *mp, void *obj)
- static void rte_mempool_put (struct rte_mempool *mp, void *obj)
- static int rte_mempool_mc_get_bulk (struct rte_mempool *mp, void **obj_table, unsigned n)
- static int rte_mempool_sc_get_bulk (struct rte_mempool *mp, void **obj_table, unsigned n)
- static int rte_mempool_get_bulk (struct rte_mempool *mp, void **obj_table, unsigned n)
- static int rte_mempool_mc_get (struct rte_mempool *mp, void **obj_p)
- static int rte_mempool_sc_get (struct rte_mempool *mp, void **obj_p)
- static int rte_mempool_get (struct rte_mempool *mp, void **obj_p)
- unsigned rte_mempool_count (const struct rte_mempool *mp)
- static unsigned rte_mempool_free_count (const struct rte_mempool *mp)
- static int rte_mempool_full (const struct rte_mempool *mp)
- static int rte_mempool_empty (const struct rte_mempool *mp)
- static phys_addr_t rte_mempool_virt2phy (const struct rte_mempool *mp, const void *elt)
- void rte_mempool_audit (const struct rte_mempool *mp)
- static void * rte_mempool_get_priv (struct rte_mempool *mp)
- void rte_mempool_list_dump (void)
- struct rte_mempool * rte_mempool_lookup (const char *name)
- uint32_t rte_mempool_calc_obj_size (uint32_t elt_size, uint32_t flags, struct rte_mempool_objsz *sz)
- size_t rte_mempool_xmem_size (uint32_t elt_num, size_t elt_sz, uint32_t pg_shift)
- ssize_t rte_mempool_xmem_usage (void *vaddr, uint32_t elt_num, size_t elt_sz, const phys_addr_t paddr[], uint32_t pg_num, uint32_t pg_shift)

3.31.1 Detailed Description

RTE Mempool.

A memory pool is an allocator of fixed-size object. It is identified by its name, and uses a ring to store free objects. It provides some other optional services, like a per-core object cache, and an alignment helper to ensure that objects are padded to spread them equally on all RAM channels, ranks, and so on.

Objects owned by a mempool should never be added in another mempool. When an object is freed using rte_mempool_put() or equivalent, the object data is not modified; the user can save some meta-data in the object data and retrieve them when allocating a new object.

Note: the mempool implementation is not preemptable. A lcore must not be interrupted by another task that uses the same mempool (because it uses a ring which is not preemptable). Also, mempool functions must not be used outside the DPDK environment: for example, in linuxapp environment, a thread that is not created by the EAL must not use mempools. This is due to the per-lcore cache that won't work as rte_lcore_id() will not return a correct value.

3.31.2 Define Documentation

3.31.2.1 #define RTE_MEMPOOL_HEADER_COOKIE1

Header cookie.

3.31.2.2 #define RTE_MEMPOOL_HEADER_COOKIE2

Header cookie.

3.31.2.3 #define RTE_MEMPOOL_TRAILER_COOKIE

Trailer cookie.

3.31.2.4 #define RTE_MEMPOOL_NAMESIZE

Maximum length of a memory pool.

3.31.2.5 #define MEMPOOL_PG_NUM_DEFAULT

Mempool over one chunk of physically continuous memory

3.31.2.6 #define MEMPOOL_F_NO_SPREAD

Do not spread in memory.

3.31.2.7 #define MEMPOOL_F_NO_CACHE_ALIGN

Do not align objs on cache lines.

3.31.2.8 #define MEMPOOL_F_SP_PUT

Default put is "single-producer".

3.31.2.9 #define MEMPOOL_F_SC_GET

Default get is "single-consumer".

3.31.2.10 #define MEMPOOL_HEADER_SIZE(*mp, pgn*)

Calculates size of the mempool header.

Parameters

mp	Pointer to the memory pool.
pgn	Number of page used to store mempool objects.

3.31.2.11 #define MEMPOOL_IS_CONTIG(*mp*)

Returns TRUE if whole mempool is allocated in one contiguous block of memory.

3.31.3 Typedef Documentation

3.31.3.1 typedef void(* rte_mempool_obj_iter_t)(void *, void *, void *, uint32_t)

An mempool's object iterator callback function.

3.31.3.2 typedef void(rte_mempool_obj_ctor_t)(struct rte_mempool *, void *, void *, unsigned)

An object constructor callback function for mempool.

Arguments are the mempool, the opaque pointer given by the user in rte_mempool_create(), the pointer to the element and the index of the element in the pool.

3.31.3.3 typedef void(rte_mempool_ctor_t)(struct rte_mempool *, void *)

A mempool constructor callback function.

Arguments are the mempool and the opaque pointer given by the user in rte_mempool_create().

3.31.4 Function Documentation

3.31.4.1 static struct rte_mempool* rte_mempool_from_obj (void * obj) [static, read]

Return a pointer to the mempool owning this object.

Parameters

obj	An object that is owned by a pool. If this is not the case, the behavior is undefined.

Returns

A pointer to the mempool structure.

3.31.4.2 struct rte_mempool* rte_mempool_create (const char * name, unsigned n, unsigned elt_size, unsigned cache_size, unsigned private_data_size, rte_mempool_ctor_t * mp_init, void * mp_init_arg, rte_mempool_obj_ctor_t * obj_init, void * obj_init_arg, int socket_id, unsigned flags) [read]

Creates a new mempool named *name* in memory.

This function uses “memzone_reserve()“ to allocate memory. The pool contains n elements of elt_size. - Its size is set to n. All elements of the mempool are allocated together with the mempool header, in one physically continuous chunk of memory.

Parameters

name	The name of the mempool.
n	The number of elements in the mempool. 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).
elt_size	The size of each element.
cache_size	If cache_size is non-zero, the rte_mempool library will try to limit the accesses to the common lockless pool, by maintaining a per-lcore object cache. This argument must be lower or equal to CONFIG_RTE_MEMPOOL_CACHE_MAX_SIZE. It is advised to choose cache_size to have "n modulo cache_size == 0": if this is not the case, some elements will always stay in the pool and will never be used. The access to the per-lcore table is of course faster than the multi-producer/consumer pool. The cache can be disabled if the cache_size argument is set to 0; it can be useful to avoid loosing objects in cache. Note that even if not used, the memory space for cache is always reserved in a mempool structure, except if CONFIG_RTE_MEMPOOL_CACHE_MAX_SIZE is set to 0.
private_data_size	The size of the private data appended after the mempool structure. This is useful for storing some private data after the mempool structure, as is done for rte_mbuf_pool for example.
mp_init	A function pointer that is called for initialization of the pool, before object initialization. The user can initialize the private data in this function if needed. This parameter can be NULL if not needed.
mp_init_arg	An opaque pointer to data that can be used in the mempool constructor function.

obj_init	A function pointer that is called for each object at initialization of the pool. The user can set some meta data in objects if needed. This parameter can be NULL if not needed. The obj_init() function takes the mempool pointer, the init_arg, the object pointer and the object number as parameters.
obj_init_arg	An opaque pointer to data that can be used as an argument for each call to the object constructor function.
socket_id	The ∗socket_id∗ argument is the socket identifier in the case of NUMA. The value can be ∗SOCKET_ID_ANY∗ if there is no NUMA constraint for the reserved zone.
flags	The ∗flags∗ arguments is an OR of following flags: • MEMPOOL_F_NO_SPREAD: By default, objects addresses are spread between channels in RAM: the pool allocator will add padding between objects depending on the hardware configuration. See Memory alignment constraints for details. If this flag is set, the allocator will just align them to a cache line. • MEMPOOL_F_NO_CACHE_ALIGN: By default, the returned objects are cache-aligned. This flag removes this constraint, and no padding will be present between objects. This flag implies MEMPOOL_F_NO_SPREAD. • MEMPOOL_F_SP_PUT: If this flag is set, the default behavior when using rte_mempool_put() or rte_mempool_put_bulk() is "single-producer". Otherwise, it is "multi-producers". • MEMPOOL_F_SC_GET: If this flag is set, the default behavior when using rte_mempool_get() or rte_mempool_get_bulk() is "single-consumer". Otherwise, it is "multi-consumers".

Returns

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
- E_RTE_NO_TAILQ - no tailq list could be got for the ring or mempool list
- EINVAL - cache size provided is too large
- 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

3.31.4.3 struct rte_mempool∗ rte_mempool_xmem_create (const char ∗ *name*, unsigned *n*, unsigned *elt_size*, unsigned *cache_size*, unsigned *private_data_size*, rte_mempool_ctor_t ∗ *mp_init*, void ∗ *mp_init_arg*, rte_mempool_obj_ctor_t ∗ *obj_init*, void ∗ *obj_init_arg*, int *socket_id*, unsigned *flags*, void ∗ *vaddr*, const phys_addr_t *paddr[]*, uint32_t *pg_num*, uint32_t *pg_shift*) [read]

Creates a new mempool named ∗name∗ in memory.

This function uses “memzone_reserve()“ to allocate memory. The pool contains n elements of elt_size. Its size is set to n. Depending on the input parameters, mempool elements can be either allocated together with the mempool header, or an externally provided memory buffer could be used to store mempool objects. In later case, that external memory buffer can consist of set of disjoint phyiscal pages.

Parameters

name	The name of the mempool.
n	The number of elements in the mempool. 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).
elt_size	The size of each element.
cache_size	If cache_size is non-zero, the rte_mempool library will try to limit the accesses to the common lockless pool, by maintaining a per-lcore object cache. This argument must be lower or equal to CONFIG_RTE_MEMPOOL_CACHE_MAX_SIZE. It is advised to choose cache_size to have "n modulo cache_size == 0": if this is not the case, some elements will always stay in the pool and will never be used. The access to the per-lcore table is of course faster than the multi-producer/consumer pool. The cache can be disabled if the cache_size argument is set to 0; it can be useful to avoid loosing objects in cache. Note that even if not used, the memory space for cache is always reserved in a mempool structure, except if CONFIG_RTE_MEMPOOL_CACHE_MAX_SIZE is set to 0.
private_data_size	The size of the private data appended after the mempool structure. This is useful for storing some private data after the mempool structure, as is done for rte_mbuf_pool for example.
mp_init	A function pointer that is called for initialization of the pool, before object initialization. The user can initialize the private data in this function if needed. This parameter can be NULL if not needed.
mp_init_arg	An opaque pointer to data that can be used in the mempool constructor function.
obj_init	A function pointer that is called for each object at initialization of the pool. The user can set some meta data in objects if needed. This parameter can be NULL if not needed. The obj_init() function takes the mempool pointer, the init_arg, the object pointer and the object number as parameters.
obj_init_arg	An opaque pointer to data that can be used as an argument for each call to the object constructor function.
socket_id	The ∗socket_id∗ argument is the socket identifier in the case of NUMA. The value can be ∗SOCKET_ID_ANY∗ if there is no NUMA constraint for the reserved zone.

flags	The *flags* arguments is an OR of following flags: • MEMPOOL_F_NO_SPREAD: By default, objects addresses are spread between channels in RAM: the pool allocator will add padding between objects depending on the hardware configuration. See Memory alignment constraints for details. If this flag is set, the allocator will just align them to a cache line. • MEMPOOL_F_NO_CACHE_ALIGN: By default, the returned objects are cache-aligned. This flag removes this constraint, and no padding will be present between objects. This flag implies MEMPOOL_F_NO_SPREAD. • MEMPOOL_F_SP_PUT: If this flag is set, the default behavior when using rte_mempool_put() or rte_mempool_put_bulk() is "single-producer". Otherwise, it is "multi-producers". • MEMPOOL_F_SC_GET: If this flag is set, the default behavior when using rte_mempool_get() or rte_mempool_get_bulk() is "single-consumer". Otherwise, it is "multi-consumers".
vaddr	Virtual address of the externally allocated memory buffer. Will be used to store mempool objects.
paddr	Array of phyiscall addresses of the pages that comprises given memory buffer.
pg_num	Number of elements in the paddr array.
pg_shift	LOG2 of the physical pages size.

Returns

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
- E_RTE_NO_TAILQ - no tailq list could be got for the ring or mempool list
- EINVAL - cache size provided is too large
- 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

3.31.4.4 void rte_mempool_dump (const struct rte_mempool * *mp*)

Dump the status of the mempool to the console.

Parameters

mp	A pointer to the mempool structure.

3.31.4.5 static void rte_mempool_mp_put_bulk (struct rte_mempool * mp, void *const * obj_table, unsigned n) [static]

Put several objects back in the mempool (multi-producers safe).

Parameters

mp	A pointer to the mempool structure.
obj_table	A pointer to a table of void * pointers (objects).
n	The number of objects to add in the mempool from the obj_table.

3.31.4.6 static void rte_mempool_sp_put_bulk (struct rte_mempool * mp, void *const * obj_table, unsigned n) [static]

Put several objects back in the mempool (NOT multi-producers safe).

Parameters

mp	A pointer to the mempool structure.
obj_table	A pointer to a table of void * pointers (objects).
n	The number of objects to add in the mempool from obj_table.

3.31.4.7 static void rte_mempool_put_bulk (struct rte_mempool * mp, void *const * obj_table, unsigned n) [static]

Put several objects back in the mempool.

This function calls the multi-producer or the single-producer version depending on the default behavior that was specified at mempool creation time (see flags).

Parameters

mp	A pointer to the mempool structure.
obj_table	A pointer to a table of void * pointers (objects).
n	The number of objects to add in the mempool from obj_table.

3.31.4.8 static void rte_mempool_mp_put (struct rte_mempool * mp, void * obj) [static]

Put one object in the mempool (multi-producers safe).

Parameters

mp	A pointer to the mempool structure.
obj	A pointer to the object to be added.

3.31.4.9 static void rte_mempool_sp_put (struct rte_mempool * *mp*, void * *obj*) [static]

Put one object back in the mempool (NOT multi-producers safe).

Parameters

mp	A pointer to the mempool structure.
obj	A pointer to the object to be added.

3.31.4.10 static void rte_mempool_put (struct rte_mempool * *mp*, void * *obj*) [static]

Put one object back in the mempool.

This function calls the multi-producer or the single-producer version depending on the default behavior that was specified at mempool creation time (see flags).

Parameters

mp	A pointer to the mempool structure.
obj	A pointer to the object to be added.

3.31.4.11 static int rte_mempool_mc_get_bulk (struct rte_mempool * *mp*, void ** *obj_table*, unsigned *n*) [static]

Get several objects from the mempool (multi-consumers safe).

If cache is enabled, objects will be retrieved first from cache, subsequently from the common pool. Note that it can return -ENOENT when the local cache and common pool are empty, even if cache from other lcores are full.

Parameters

mp	A pointer to the mempool structure.
obj_table	A pointer to a table of void * pointers (objects) that will be filled.
n	The number of objects to get from mempool to obj_table.

Returns

- 0: Success; objects taken.
- -ENOENT: Not enough entries in the mempool; no object is retrieved.

3.31.4.12 static int rte_mempool_sc_get_bulk (struct rte_mempool * *mp*, void ** *obj_table*, unsigned *n*) [static]

Get several objects from the mempool (NOT multi-consumers safe).

If cache is enabled, objects will be retrieved first from cache, subsequently from the common pool. Note that it can return -ENOENT when the local cache and common pool are empty, even if cache from other lcores are full.

Parameters

mp	A pointer to the mempool structure.
obj_table	A pointer to a table of void * pointers (objects) that will be filled.
n	The number of objects to get from the mempool to obj_table.

Returns

- 0: Success; objects taken.
- -ENOENT: Not enough entries in the mempool; no object is retrieved.

3.31.4.13 static int rte_mempool_get_bulk (struct rte_mempool * *mp*, void ** *obj_table*, unsigned *n*) [static]

Get several objects from the mempool.

This function calls the multi-consumers or the single-consumer version, depending on the default behaviour that was specified at mempool creation time (see flags).

If cache is enabled, objects will be retrieved first from cache, subsequently from the common pool. Note that it can return -ENOENT when the local cache and common pool are empty, even if cache from other lcores are full.

Parameters

mp	A pointer to the mempool structure.
obj_table	A pointer to a table of void * pointers (objects) that will be filled.
n	The number of objects to get from the mempool to obj_table.

Returns

- 0: Success; objects taken
- -ENOENT: Not enough entries in the mempool; no object is retrieved.

3.31.4.14 static int rte_mempool_mc_get (struct rte_mempool * *mp*, void ** *obj_p*) [static]

Get one object from the mempool (multi-consumers safe).

If cache is enabled, objects will be retrieved first from cache, subsequently from the common pool. Note that it can return -ENOENT when the local cache and common pool are empty, even if cache from other lcores are full.

Parameters

mp	A pointer to the mempool structure.
obj_p	A pointer to a void * pointer (object) that will be filled.

Returns

- 0: Success; objects taken.
- -ENOENT: Not enough entries in the mempool; no object is retrieved.

3.31.4.15 static int rte_mempool_sc_get (struct rte_mempool * *mp*, void ** *obj_p*) [static]

Get one object from the mempool (NOT multi-consumers safe).

If cache is enabled, objects will be retrieved first from cache, subsequently from the common pool. Note that it can return -ENOENT when the local cache and common pool are empty, even if cache from other lcores are full.

Parameters

mp	A pointer to the mempool structure.
obj_p	A pointer to a void * pointer (object) that will be filled.

Returns

- 0: Success; objects taken.
- -ENOENT: Not enough entries in the mempool; no object is retrieved.

3.31.4.16 static int rte_mempool_get (struct rte_mempool * *mp*, void ** *obj_p*) [static]

Get one object from the mempool.

This function calls the multi-consumers or the single-consumer version, depending on the default behavior that was specified at mempool creation (see flags).

If cache is enabled, objects will be retrieved first from cache, subsequently from the common pool. Note that it can return -ENOENT when the local cache and common pool are empty, even if cache from other lcores are full.

Parameters

mp	A pointer to the mempool structure.
obj_p	A pointer to a void * pointer (object) that will be filled.

Returns

- 0: Success; objects taken.
- -ENOENT: Not enough entries in the mempool; no object is retrieved.

3.31.4.17 unsigned rte_mempool_count (const struct rte_mempool * *mp*)

Return the number of entries in the mempool.

When cache is enabled, this function has to browse the length of all lcores, so it should not be used in a data path, but only for debug purposes.

Parameters

mp	A pointer to the mempool structure.

Returns

The number of entries in the mempool.

3.31.4.18 static unsigned rte_mempool_free_count (const struct rte_mempool * *mp*) [static]

Return the number of free entries in the mempool ring. i.e. how many entries can be freed back to the mempool.

NOTE: This corresponds to the number of elements ∗allocated∗ from the memory pool, not the number of elements in the pool itself. To count the number elements currently available in the pool, use "rte_mempool_count"

When cache is enabled, this function has to browse the length of all lcores, so it should not be used in a data path, but only for debug purposes.

Parameters

mp	A pointer to the mempool structure.

Returns

The number of free entries in the mempool.

3.31.4.19 static int rte_mempool_full (const struct rte_mempool * *mp*) [static]

Test if the mempool is full.

When cache is enabled, this function has to browse the length of all lcores, so it should not be used in a data path, but only for debug purposes.

Parameters

mp	A pointer to the mempool structure.

Returns

- 1: The mempool is full.
- 0: The mempool is not full.

3.31.4.20 static int rte_mempool_empty (const struct rte_mempool * *mp*) [static]

Test if the mempool is empty.

When cache is enabled, this function has to browse the length of all lcores, so it should not be used in a data path, but only for debug purposes.

Parameters

mp	A pointer to the mempool structure.

Returns

- 1: The mempool is empty.
- 0: The mempool is not empty.

3.31.4.21 static phys_addr_t rte_mempool_virt2phy (const struct rte_mempool * *mp,* const void * *elt*) [static]

Return the physical address of elt, which is an element of the pool mp.

Parameters

mp	A pointer to the mempool structure.
elt	A pointer (virtual address) to the element of the pool.

Returns

The physical address of the elt element.

3.31.4.22 void rte_mempool_audit (const struct rte_mempool * *mp*)

Check the consistency of mempool objects.

Verify the coherency of fields in the mempool structure. Also check that the cookies of mempool objects (even the ones that are not present in pool) have a correct value. If not, a panic will occur.

Parameters

mp	A pointer to the mempool structure.

3.31.4.23 static void* rte_mempool_get_priv (struct rte_mempool * *mp*) [static]

Return a pointer to the private data in an mempool structure.

Parameters

mp	A pointer to the mempool structure.

Returns

A pointer to the private data.

3.31.4.24 void rte_mempool_list_dump (void)

Dump the status of all mempools on the console

3.31.4.25 struct rte_mempool* rte_mempool_lookup (const char * *name*) [read]

Search a mempool from its name

Parameters

name	The name of the mempool.

Returns

The pointer to the mempool matching the name, or NULL if not found. NULL on error with rte_errno set appropriately. Possible rte_errno values include:

- ENOENT - required entry not available to return.

3.31.4.26 uint32_t rte_mempool_calc_obj_size (uint32_t *elt_size,* uint32_t *flags,* struct rte_mempool_objsz * *sz*)

Given a desired size of the mempool element and mempool flags, caluclates header, trailer, body and total sizes of the mempool object.

Parameters

elt_size	The size of each element.
flags	The flags used for the mempool creation. Consult rte_mempool_create() for more information about possible values. The size of each element.

Returns

Total size of the mempool object.

3.31.4.27 size_t rte_mempool_xmem_size (uint32_t *elt_num*, size_t *elt_sz*, uint32_t *pg_shift*)

Calculate maximum amount of memory required to store given number of objects. Assumes that the memory buffer will be alligned at page boundary. Note, that if object size is bigger then page size, then it assumes that we have a subsets of physically continuous pages big enough to store at least one object.

Parameters

elt_num	Number of elements.
elt_sz	The size of each element.
pg_shift	LOG2 of the physical pages size.

Returns

Required memory size aligned at page boundary.

3.31.4.28 ssize_t rte_mempool_xmem_usage (void * *vaddr*, uint32_t *elt_num*, size_t *elt_sz*, const phys_addr_t *paddr[]*, uint32_t *pg_num*, uint32_t *pg_shift*)

Calculate how much memory would be actually required with the given memory footprint to store required number of objects.

Parameters

vaddr	Virtual address of the externally allocated memory buffer. Will be used to store mempool objects.
elt_num	Number of elements.
elt_sz	The size of each element.
paddr	Array of phyiscall addresses of the pages that comprises given memory buffer.
pg_num	Number of elements in the paddr array.
pg_shift	LOG2 of the physical pages size.

Returns

Number of bytes needed to store given number of objects, aligned to the given page size. If provided memory buffer is not big enough: (-1) * actual number of elemnts that can be stored in that buffer.

3.32 rte_memzone.h File Reference

Data Structures

- struct rte_memzone

Defines

- #define RTE_MEMZONE_2MB
- #define RTE_MEMZONE_1GB
- #define RTE_MEMZONE_SIZE_HINT_ONLY
- #define RTE_MEMZONE_NAMESIZE

Functions

- struct rte_memzone * rte_memzone_reserve (const char *name, size_t len, int socket_id, unsigned flags)
- struct rte_memzone * rte_memzone_reserve_aligned (const char *name, size_t len, int socket_id, unsigned flags, unsigned align)
- struct rte_memzone * rte_memzone_reserve_bounded (const char *name, size_t len, int socket_id, unsigned flags, unsigned align, unsigned bound)
- struct rte_memzone * rte_memzone_lookup (const char *name)
- void rte_memzone_dump (void)

3.32.1 Detailed Description

RTE Memzone

The goal of the memzone allocator is to reserve contiguous portions of physical memory. These zones are identified by a name.

The memzone descriptors are shared by all partitions and are located in a known place of physical memory. This zone is accessed using rte_eal_get_configuration(). The lookup (by name) of a memory zone can be done in any partition and returns the same physical address.

A reserved memory zone cannot be unreserved. The reservation shall be done at initialization time only.

3.32.2 Define Documentation

3.32.2.1 #define RTE_MEMZONE_2MB

Use 2MB pages.

3.32.2.2 #define RTE_MEMZONE_1GB

Use 1GB pages.

3.32.2.3 #define RTE_MEMZONE_SIZE_HINT_ONLY

Use available page size

3.32.2.4 #define RTE_MEMZONE_NAMESIZE

Maximum length of memory zone name.

3.32.3 Function Documentation

3.32.3.1 struct rte_memzone* rte_memzone_reserve (const char * *name*, size_t *len*, int *socket_id*, unsigned *flags*) [read]

Reserve a portion of physical memory.

This function reserves some memory and returns a pointer to a correctly filled memzone descriptor. If the allocation cannot be done, return NULL. Note: A reserved zone cannot be freed.

Parameters

name	The name of the memzone. If it already exists, the function will fail and return NULL.
len	The size of the memory to be reserved. If it is 0, the biggest contiguous zone will be reserved.
socket_id	The socket identifier in the case of NUMA. The value can be SOCKET_ID_ANY if there is no NUMA constraint for the reserved zone.
flags	The flags parameter is used to request memzones to be taken from 1GB or 2MB hugepages. • RTE_MEMZONE_2MB - Reserve from 2MB pages • RTE_MEMZONE_1GB - Reserve from 1GB pages • RTE_MEMZONE_SIZE_HINT_ONLY - Allow alternative page size to be used if the requested page size is unavailable. If this flag is not set, the function will return error on an unavailable size request.

Returns

A pointer to a correctly-filled read-only memzone descriptor, or NULL on error. On error case, rte_errno will be set appropriately:

- E_RTE_NO_CONFIG - function could not get pointer to rte_config structure
- E_RTE_SECONDARY - function was called from a secondary process instance
- 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
- EINVAL - invalid parameters

3.32.3.2 struct rte_memzone* rte_memzone_reserve_aligned (const char * *name*, size_t *len*, int *socket_id*, unsigned *flags*, unsigned *align*) [read]

Reserve a portion of physical memory with alignment on a specified boundary.

This function reserves some memory with alignment on a specified boundary, and returns a pointer to a correctly filled memzone descriptor. If the allocation cannot be done or if the alignment is not a power of 2, returns NULL. Note: A reserved zone cannot be freed.

Parameters

name	The name of the memzone. If it already exists, the function will fail and return NULL.
len	The size of the memory to be reserved. If it is 0, the biggest contiguous zone will be reserved.
socket_id	The socket identifier in the case of NUMA. The value can be SOCKET_ID_ANY if there is no NUMA constraint for the reserved zone.
flags	The flags parameter is used to request memzones to be taken from 1GB or 2MB hugepages. • RTE_MEMZONE_2MB - Reserve from 2MB pages • RTE_MEMZONE_1GB - Reserve from 1GB pages • RTE_MEMZONE_SIZE_HINT_ONLY - Allow alternative page size to be used if the requested page size is unavailable. If this flag is not set, the function will return error on an unavailable size request.
align	Alignment for resulting memzone. Must be a power of 2.

Returns

A pointer to a correctly-filled read-only memzone descriptor, or NULL on error. On error case, rte_errno will be set appropriately:

- E_RTE_NO_CONFIG - function could not get pointer to rte_config structure
- E_RTE_SECONDARY - function was called from a secondary process instance
- 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
- EINVAL - invalid parameters

3.32.3.3 struct rte_memzone* rte_memzone_reserve_bounded (const char * *name*, size_t *len*, int *socket_id*, unsigned *flags*, unsigned *align*, unsigned *bound*) [read]

Reserve a portion of physical memory with specified alignment and boundary.

This function reserves some memory with specified alignment and boundary, and returns a pointer to a correctly filled memzone descriptor. If the allocation cannot be done or if the alignment or boundary are not a power of 2, returns NULL. Memory buffer is reserved in a way, that it wouldn't cross specified boundary. That implies that requested length should be less or equal then boundary. Note: A reserved zone cannot be freed.

Parameters

name	The name of the memzone. If it already exists, the function will fail and return NULL.
len	The size of the memory to be reserved. If it is 0, the biggest contiguous zone will be reserved.
socket_id	The socket identifier in the case of NUMA. The value can be SOCKET_ID_ANY if there is no NUMA constraint for the reserved zone.
flags	The flags parameter is used to request memzones to be taken from 1GB or 2MB hugepages. • RTE_MEMZONE_2MB - Reserve from 2MB pages • RTE_MEMZONE_1GB - Reserve from 1GB pages • RTE_MEMZONE_SIZE_HINT_ONLY - Allow alternative page size to be used if the requested page size is unavailable. If this flag is not set, the function will return error on an unavailable size request.
align	Alignment for resulting memzone. Must be a power of 2.
bound	Boundary for resulting memzone. Must be a power of 2 or zero. Zero value implies no boundary condition.

Returns

A pointer to a correctly-filled read-only memzone descriptor, or NULL on error. On error case, rte_errno will be set appropriately:

- E_RTE_NO_CONFIG - function could not get pointer to rte_config structure
- E_RTE_SECONDARY - function was called from a secondary process instance
- 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
- EINVAL - invalid parameters

3.32.3.4 struct rte_memzone* rte_memzone_lookup (const char * *name*) [read]

Lookup for a memzone.

Get a pointer to a descriptor of an already reserved memory zone identified by the name given as an argument.

Parameters

name	The name of the memzone.

Returns

A pointer to a read-only memzone descriptor.

3.32.3.5 void rte_memzone_dump (void)

Dump all reserved memzones to the console.

3.33 rte_meter.h File Reference

Data Structures

- struct rte_meter_srtcm_params
- struct rte_meter_trtcm_params
- struct rte_meter_srtcm
- struct rte_meter_trtcm

Enumerations

- enum rte_meter_color { e_RTE_METER_GREEN, e_RTE_METER_YELLOW, e_RTE_METER_RED, e_RTE_METER_COLORS }

Functions

- int rte_meter_srtcm_config (struct rte_meter_srtcm *m, struct rte_meter_srtcm_params *params)
- int rte_meter_trtcm_config (struct rte_meter_trtcm *m, struct rte_meter_trtcm_params *params)
- static enum rte_meter_color rte_meter_srtcm_color_blind_check (struct rte_meter_srtcm *m, uint64_t time, uint32_t pkt_len)
- static enum rte_meter_color rte_meter_srtcm_color_aware_check (struct rte_meter_srtcm *m, uint64_t time, uint32_t pkt_len, enum rte_meter_color pkt_color)
- static enum rte_meter_color rte_meter_trtcm_color_blind_check (struct rte_meter_trtcm *m, uint64_t time, uint32_t pkt_len)
- static enum rte_meter_color rte_meter_trtcm_color_aware_check (struct rte_meter_trtcm *m, uint64_t time, uint32_t pkt_len, enum rte_meter_color pkt_color)

3.33.1 Detailed Description

RTE Traffic Metering

Traffic metering algorithms: 1. Single Rate Three Color Marker (srTCM): defined by IETF RFC 2697 2. Two Rate Three Color Marker (trTCM): defined by IETF RFC 2698

3.33.2 Enumeration Type Documentation

3.33.2.1 enum rte_meter_color

Packet Color Set

Enumerator:

e_RTE_METER_GREEN Green

e_RTE_METER_YELLOW Yellow

e_RTE_METER_RED Red

e_RTE_METER_COLORS Number of available colors

3.33.3 Function Documentation

3.33.3.1 int rte_meter_srtcm_config (struct rte_meter_srtcm * *m*, struct rte_meter_srtcm_params * *params*)

srTCM configuration per metered traffic flow

Parameters

m	Pointer to pre-allocated srTCM data structure
params	User parameters per srTCM metered traffic flow

Returns

0 upon success, error code otherwise

3.33.3.2 int rte_meter_trtcm_config (struct rte_meter_trtcm * *m*, struct rte_meter_trtcm_params * *params*)

trTCM configuration per metered traffic flow

Parameters

m	Pointer to pre-allocated trTCM data structure
params	User parameters per trTCM metered traffic flow

Returns

0 upon success, error code otherwise

3.33.3.3 static enum rte_meter_color rte_meter_srtcm_color_blind_check (struct rte_meter_srtcm * *m*, uint64_t *time*, uint32_t *pkt_len*) [static]

srTCM color blind traffic metering

Parameters

m	Handle to srTCM instance
time	Current CPU time stamp (measured in CPU cycles)
pkt_length	Length of the current IP packet (measured in bytes)

Returns

Color assigned to the current IP packet

3.33.3.4 static enum rte_meter_color rte_meter_srtcm_color_aware_check (struct rte_meter_srtcm * *m*, uint64_t *time*, uint32_t *pkt_len*, enum rte_meter_color *pkt_color*) [static]

srTCM color aware traffic metering

Parameters

m	Handle to srTCM instance
time	Current CPU time stamp (measured in CPU cycles)
pkt_length	Length of the current IP packet (measured in bytes)
pkt_color	Input color of the current IP packet

Returns

Color assigned to the current IP packet

3.33.3.5 static enum rte_meter_color rte_meter_trtcm_color_blind_check (struct rte_meter_trtcm * *m*, uint64_t *time*, uint32_t *pkt_len*) [static]

trTCM color blind traffic metering

Parameters

m	Handle to trTCM instance
time	Current CPU time stamp (measured in CPU cycles)
pkt_length	Length of the current IP packet (measured in bytes)

Returns

Color assigned to the current IP packet

3.33.3.6 static enum rte_meter_color rte_meter_trtcm_color_aware_check (struct rte_meter_trtcm * *m*, uint64_t *time*, uint32_t *pkt_len*, enum rte_meter_color *pkt_color*) [static]

trTCM color aware traffic metering

Parameters

m	Handle to trTCM instance
time	Current CPU time stamp (measured in CPU cycles)
pkt_length	Length of the current IP packet (measured in bytes)
pkt_color	Input color of the current IP packet

Returns

Color assigned to the current IP packet

3.34 rte_pci.h File Reference

Data Structures

- struct rte_pci_resource
- struct rte_pci_id
- struct rte_pci_addr
- struct rte_pci_device
- struct rte_pci_driver

Defines

- #define SYSFS_PCI_DEVICES
- #define PCI_PRI_FMT
- #define PCI_SHORT_PRI_FMT
- #define PCI_FMT_NVAL
- #define PCI_RESOURCE_FMT_NVAL
- #define PCI_MAX_RESOURCE
- #define PCI_ANY_ID
- #define RTE_PCI_DEVICE(vend, dev)
- #define RTE_PCI_DRV_MULTIPLE

Typedefs

- typedef int(pci_devinit_t)(struct rte_pci_driver *, struct rte_pci_device *)

Functions

- TAILQ_HEAD (pci_device_list, rte_pci_device)
- TAILQ_HEAD (pci_driver_list, rte_pci_driver)
- static int eal_parse_pci_BDF (const char *input, struct rte_pci_addr *dev_addr)
- static int eal_parse_pci_DomBDF (const char *input, struct rte_pci_addr *dev_addr)
- int rte_eal_pci_probe (void)
- void rte_eal_pci_dump (void)
- void rte_eal_pci_register (struct rte_pci_driver *driver)
- void rte_eal_pci_unregister (struct rte_pci_driver *driver)
- void rte_eal_pci_set_blacklist (struct rte_pci_addr *blacklist, unsigned size)

Variables

- struct pci_driver_list driver_list
- struct pci_device_list device_list

3.34.1 Detailed Description

RTE PCI Interface

3.34.2 Define Documentation

3.34.2.1 #define SYSFS_PCI_DEVICES

Pathname of PCI devices directory.

3.34.2.2 #define PCI_PRI_FMT

Formatting string for PCI device identifier: Ex: 0000:00:01.0

3.34.2.3 #define PCI_SHORT_PRI_FMT

Short formatting string, without domain, for PCI device: Ex: 00:01.0

3.34.2.4 #define PCI_FMT_NVAL

Nb. of values in PCI device identifier format string.

3.34.2.5 #define PCI_RESOURCE_FMT_NVAL

Nb. of values in PCI resource format.

3.34.2.6 #define PCI_MAX_RESOURCE

Maximum number of PCI resources.

3.34.2.7 #define PCI_ANY_ID

Any PCI device identifier (vendor, device, ...)

3.34.2.8 #define RTE_PCI_DEVICE(vend, dev)

Macro used to help building up tables of device IDs

3.34.2.9 #define RTE_PCI_DRV_MULTIPLE

Device driver must be registered several times until failure Internal use only - Macro used by pci addr parsing functions

3.34.3 Typedef Documentation

3.34.3.1 typedef int(pci_devinit_t)(struct rte_pci_driver *, struct rte_pci_device *)

Initialisation function for the driver called during PCI probing.

3.34.4 Function Documentation

3.34.4.1 TAILQ_HEAD (pci_device_list , rte_pci_device)

PCI devices in D-linked Q.

3.34.4.2 TAILQ_HEAD (pci_driver_list , rte_pci_driver)

PCI drivers in D-linked Q.

3.34.4.3 static int eal_parse_pci_BDF (const char * input, struct rte_pci_addr * dev_addr) [static]

Utility function to produce a PCI Bus-Device-Function value given a string representation. Assumes that the BDF is provided without a domain prefix (i.e. domain returned is always 0)

Parameters

input	The input string to be parsed. Should have the format XX:XX.X
dev_addr	The PCI Bus-Device-Function address to be returned. Domain will always be returned as 0

Returns

0 on success, negative on error.

3.34.4.4 static int eal_parse_pci_DomBDF (const char * *input*, struct rte_pci_addr * *dev_addr*) [static]

Utility function to produce a PCI Bus-Device-Function value given a string representation. Assumes that the BDF is provided including a domain prefix.

Parameters

input	The input string to be parsed. Should have the format XXXX:XX:XX.X
dev_addr	The PCI Bus-Device-Function address to be returned

Returns

0 on success, negative on error.

3.34.4.5 int rte_eal_pci_probe (void)

Probe the PCI bus for registered drivers.

Scan the content of the PCI bus, and call the probe() function for all registered drivers that have a matching entry in its id_table for discovered devices.

Returns

- 0 on success.
- Negative on error.

3.34.4.6 void rte_eal_pci_dump (void)

Dump the content of the PCI bus.

3.34.4.7 void rte_eal_pci_register (struct rte_pci_driver * *driver*)

Register a PCI driver.

Parameters

driver	A pointer to a rte_pci_driver structure describing the driver to be registered.

3.34.4.8 void rte_eal_pci_unregister (struct rte_pci_driver * *driver*)

Unregister a PCI driver.

Parameters

driver	A pointer to a rte_pci_driver structure describing the driver to be unregistered.

3.34.4.9 void rte_eal_pci_set_blacklist (struct rte_pci_addr * *blacklist*, unsigned *size*)

Register a list of PCI locations that will be blacklisted (not used by DPDK).

Parameters

blacklist	List of PCI device addresses that will not be used by DPDK.
size	Number of items in the list.

3.34.5 Variable Documentation

3.34.5.1 struct pci_driver_list driver_list

Global list of PCI drivers.

3.34.5.2 struct pci_device_list device_list

Global list of PCI devices.

3.35 rte_pci_dev_ids.h File Reference

Defines

- #define PCI_VENDOR_ID_INTEL
- #define PCI_VENDOR_ID_QUMRANET
- #define PCI_VENDOR_ID_VMWARE

3.35.1 Detailed Description

This file contains a list of the PCI device IDs recognised by DPDK, which can be used to fill out an array of structures describing the devices.

Currently four families of devices are recognised: those supported by the IGB driver, by EM driver, those supported by the IXGBE driver, and by virtio driver which is a para virtualization driver running in guest

virtual machine. The inclusion of these in an array built using this file depends on the definition of RTE_-PCI_DEV_ID_DECL_EM RTE_PCI_DEV_ID_DECL_IGB RTE_PCI_DEV_ID_DECL_IGBVF RTE_PCI_D-EV_ID_DECL_IXGBE RTE_PCI_DEV_ID_DECL_IXGBEVF RTE_PCI_DEV_ID_DECL_VIRTIO at the time when this file is included.

In order to populate an array, the user of this file must define this macro: RTE_PCI_DEV_ID_DECL_IXGB-E(vendorID, deviceID). For example:

```
struct device {
    int vend;
    int dev;
};

struct device devices[] = {
#define RTE_PCI_DEV_ID_DECL_IXGBE(vendorID, deviceID) {vend, dev},
#include <rte_pci_dev_ids.h>
};
```

Note that this file can be included multiple times within the same file.

3.35.2 Define Documentation

3.35.2.1 #define PCI_VENDOR_ID_INTEL

Vendor ID used by Intel devices

3.35.2.2 #define PCI_VENDOR_ID_QUMRANET

Vendor ID used by virtio devices

3.35.2.3 #define PCI_VENDOR_ID_VMWARE

Vendor ID used by VMware devices

3.36 rte_per_lcore.h File Reference

Defines

- #define RTE_DEFINE_PER_LCORE(type, name)
- #define RTE_DECLARE_PER_LCORE(type, name)
- #define RTE_PER_LCORE(name)

3.36.1 Detailed Description

Per-lcore variables in RTE

This file defines an API for instantiating per-lcore "global variables" that are environment-specific. Note that in all environments, a "shared variable" is the default when you use a global variable.

Parts of this are execution environment specific.

3.36.2 Define Documentation

3.36.2.1 #define RTE_DEFINE_PER_LCORE(*type, name*)

Macro to define a per lcore variable "var" of type "type", don't use keywords like "static" or "volatile" in type, just prefix the whole macro.

3.36.2.2 #define RTE_DECLARE_PER_LCORE(*type, name*)

Macro to declare an extern per lcore variable "var" of type "type"

3.36.2.3 #define RTE_PER_LCORE(*name*)

Read/write the per-lcore variable value

3.37 rte_power.h File Reference

Functions

- int rte_power_init (unsigned lcore_id)
- int rte_power_exit (unsigned lcore_id)
- uint32_t rte_power_freqs (unsigned lcore_id, uint32_t *freqs, uint32_t num)
- uint32_t rte_power_get_freq (unsigned lcore_id)
- int rte_power_set_freq (unsigned lcore_id, uint32_t index)
- int rte_power_freq_up (unsigned lcore_id)
- int rte_power_freq_down (unsigned lcore_id)
- int rte_power_freq_max (unsigned lcore_id)
- int rte_power_freq_min (unsigned lcore_id)

3.37.1 Detailed Description

RTE Power Management

3.37.2 Function Documentation

3.37.2.1 int rte_power_init (unsigned *lcore_id*)

Initialize power management for a specific lcore. It will check and set the governor to userspace for the lcore, get the available frequencies, and prepare to set new lcore frequency.

Parameters

lcore_id	lcore id.

Returns

- 0 on success.
- Negative on error.

3.37.2.2 int rte_power_exit (unsigned *lcore_id*)

Exit power management on a specific lcore. It will set the governor to which is before initialized.

Parameters

lcore_id	lcore id.

Returns

- 0 on success.
- Negative on error.

3.37.2.3 uint32_t rte_power_freqs (unsigned *lcore_id*, uint32_t * *freqs*, uint32_t *num*)

Get the available frequencies of a specific lcore. The return value will be the minimal one of the total number of available frequencies and the number of buffer. The index of available frequencies used in other interfaces should be in the range of 0 to this return value. It should be protected outside of this function for threadsafe.

Parameters

lcore_id	lcore id.
freqs	The buffer array to save the frequencies.
num	The number of frequencies to get.

Returns

The number of available frequencies.

3.37.2.4 uint32_t rte_power_get_freq (unsigned *lcore_id*)

Return the current index of available frequencies of a specific lcore. It will return 'RTE_POWER_INVALID-_FREQ_INDEX = (~0)' if error. It should be protected outside of this function for threadsafe.

Parameters

lcore_id	lcore id.

Returns

The current index of available frequencies.

3.37.2.5 int rte_power_set_freq (unsigned *lcore_id*, uint32_t *index*)

Set the new frequency for a specific lcore by indicating the index of available frequencies. It should be protected outside of this function for threadsafe.

Parameters

lcore_id	lcore id.
index	The index of available frequencies.

Returns

- 1 on success with frequency changed.
- 0 on success without frequency chnaged.
- Negative on error.

3.37.2.6 int rte_power_freq_up (unsigned *lcore_id*)

Scale up the frequency of a specific lcore according to the available frequencies. It should be protected outside of this function for threadsafe.

Parameters

lcore_id	lcore id.

Returns

- 1 on success with frequency changed.
- 0 on success without frequency chnaged.
- Negative on error.

3.37.2.7 int rte_power_freq_down (unsigned *lcore_id*)

Scale down the frequency of a specific lcore according to the available frequencies. It should be protected outside of this function for threadsafe.

Parameters

lcore_id	lcore id.

Returns

- 1 on success with frequency changed.
- 0 on success without frequency chnaged.
- Negative on error.

3.37.2.8 int rte_power_freq_max (unsigned *lcore_id*)

Scale up the frequency of a specific lcore to the highest according to the available frequencies. It should be protected outside of this function for threadsafe.

Parameters

lcore_id	lcore id.

Returns

- 1 on success with frequency changed.
- 0 on success without frequency chnaged.
- Negative on error.

3.37.2.9 int rte_power_freq_min (unsigned *lcore_id*)

Scale down the frequency of a specific lcore to the lowest according to the available frequencies. It should be protected outside of this function for threadsafe.

Parameters

lcore_id	lcore id.

Returns

- 1 on success with frequency changed.
- 0 on success without frequency chnaged.
- Negative on error.

3.38 rte_prefetch.h File Reference

Functions

- static void rte_prefetch0 (volatile void *p)
- static void rte_prefetch1 (volatile void *p)
- static void rte_prefetch2 (volatile void *p)

3.38.1 Detailed Description

Prefetch operations.

This file defines an API for prefetch macros / inline-functions, which are architecture-dependent. Prefetching occurs when a processor requests an instruction or data from memory to cache before it is actually needed, potentially speeding up the execution of the program.

3.38.2 Function Documentation

3.38.2.1 static void rte_prefetch0 (volatile void * p) [static]

Prefetch a cache line into all cache levels.

Parameters

p	Address to prefetch

3.38.2.2 static void rte_prefetch1 (volatile void * p) [static]

Prefetch a cache line into all cache levels except the 0th cache level.

Parameters

p	Address to prefetch

3.38.2.3 static void rte_prefetch2 (volatile void * p) [static]

Prefetch a cache line into all cache levels except the 0th and 1th cache levels.

Parameters

p	Address to prefetch

3.39 rte_random.h File Reference

Functions

- static void rte_srand (uint64_t seedval)
- static uint64_t rte_rand (void)

3.39.1 Detailed Description

Pseudo-random Generators in RTE

3.39.2 Function Documentation

3.39.2.1 static void rte_srand (uint64_t *seedval*) [static]

Seed the pseudo-random generator.

The generator is automatically seeded by the EAL init with a timer value. It may need to be re-seeded by the user with a real random value.

Parameters

seedval	The value of the seed.

3.39.2.2 static uint64_t rte_rand (void) [static]

Get a pseudo-random value.

This function generates pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic, called twice to generate a 64-bit value.

Returns

A pseudo-random value between 0 and (1<<64)-1.

3.40 rte_red.h File Reference

Data Structures

- struct rte_red_params
- struct rte_red_config
- struct rte_red

Defines

- #define RTE_RED_SCALING
- #define RTE_RED_S
- #define RTE_RED_MAX_TH_MAX
- #define RTE_RED_WQ_LOG2_MIN
- #define RTE_RED_WQ_LOG2_MAX
- #define RTE_RED_MAXP_INV_MIN
- #define RTE_RED_MAXP_INV_MAX
- #define RTE_RED_2POW16

Functions

- int rte_red_rt_data_init (struct rte_red *red)
- int rte_red_config_init (struct rte_red_config *red_cfg, const uint16_t wq_log2, const uint16_t min_th, const uint16_t max_th, const uint16_t maxp_inv)
- static uint32_t rte_fast_rand (void)
- static uint16_t __rte_red_calc_qempty_factor (uint8_t wq_log2, uint16_t m)
- static int rte_red_enqueue_empty (const struct rte_red_config *red_cfg, struct rte_red *red, const uint64_t time)
- static int __rte_red_drop (const struct rte_red_config *red_cfg, struct rte_red *red)
- static int rte_red_enqueue_nonempty (const struct rte_red_config *red_cfg, struct rte_red *red, const unsigned q)
- static int rte_red_enqueue (const struct rte_red_config *red_cfg, struct rte_red *red, const unsigned q, const uint64_t time)
- static void rte_red_mark_queue_empty (struct rte_red *red, const uint64_t time)

Variables

- uint32_t rte_red_rand_val

3.40.1 Detailed Description

RTE Random Early Detection (RED)

3.40.2 Define Documentation

3.40.2.1 #define RTE_RED_SCALING

Fraction size for fixed-point

3.40.2.2 #define RTE_RED_S

Packet size multiplied by number of leaf queues

3.40.2.3 #define RTE_RED_MAX_TH_MAX

Max threshold limit in fixed point format

3.40.2.4 #define RTE_RED_WQ_LOG2_MIN

Min inverse filter weight value

3.40.2.5 #define RTE_RED_WQ_LOG2_MAX

Max inverse filter weight value

3.40.2.6 #define RTE_RED_MAXP_INV_MIN

Min inverse mark probability value

3.40.2.7 #define RTE_RED_MAXP_INV_MAX

Max inverse mark probability value

3.40.2.8 #define RTE_RED_2POW16

2 power 16

3.40.3 Function Documentation

3.40.3.1 int rte_red_rt_data_init (struct rte_red * *red*)

Initialises run-time data.

Parameters

in,out	*data*	pointer to RED runtime data

Returns

Operation status

Return values

0	success
!0	error

3.40.3.2 int rte_red_config_init (struct rte_red_config * *red_cfg*, const uint16_t *wq_log2*, const uint16_t *min_th*, const uint16_t *max_th*, const uint16_t *maxp_inv*)

Configures a single RED configuration parameter structure.

Parameters

in,out	*config*	pointer to a RED configuration parameter structure
in	*wq_log2*	log2 of the filter weight, valid range is: RTE_RED_WQ_LOG2_MIN <= wq_log2 <= RTE_RED_WQ_LOG2_MAX
in	*min_th*	queue minimum threshold in number of packets
in	*max_th*	queue maximum threshold in number of packets
in	*maxp_inv*	inverse maximum mark probability

Returns

Operation status

Return values

0	success
!0	error

3.40.3.3 static uint32_t rte_fast_rand (void) [static]

Generate random number for RED.

Implemenetation based on: http://software.intel.com/en-us/articles/fast-random-number-generator-

10 bit shift has been found through empirical tests (was 16).

Returns

Random number between 0 and (2^22 - 1)

3.40.3.4 static uint16_t __rte_red_calc_qempty_factor (uint8_t *wq_log2*, uint16_t *m*) [static]

calculate factor to scale average queue size when queue becomes empty

Parameters

in	*wq_log2,where*	EWMA filter weight wq = 1/(2 ^ wq_log2)
in	*m*	exponent in the computed value (1 - wq) ^ m

Returns

computed value

Return values

((1	- wq) ^ m) scaled in fixed-point format

Basic math tells us that: a^b = 2^(b ∗ log2(a))

in our case: a = (1-Wq) b = m Wq = 1/ (2^log2n)

So we are computing this equation: factor = 2 ^ (m ∗ log2(1-Wq))

First we are computing: n = m ∗ log2(1-Wq)

To avoid dealing with signed numbers log2 values are positive but they should be negative because (1-Wq) is always < 1. Contents of log2 table values are also scaled for precision.

The tricky part is computing 2^n, for this I split n into integer part and fraction part. f - is fraction part of n n - is integer part of original n

Now using basic math we compute 2^n: 2^(f+n) = 2^f ∗ 2^n 2^f - we use lookup table 2^n - can be replaced with bit shift right oeprations

3.40.3.5 static int rte_red_enqueue_empty (const struct rte_red_config ∗ *red_cfg*, struct rte_red ∗ *red*, const uint64_t *time*) [static]

Updates queue average in condition when queue is empty.

Note: packet is never dropped in this particular case.

Parameters

in	*config*	pointer to a RED configuration parameter structure
in,out	*data*	pointer to RED runtime data
in	*time*	current time stamp

Returns

Operation status

Return values

0	enqueue the packet
1	drop the packet based on max threshold criterion
2	drop the packet based on mark probability criterion

We compute avg but we don't compare avg against min_th or max_th, nor calculate drop probability

m is the number of packets that might have arrived while the queue was empty. In this case we have time stamps provided by scheduler in byte units (bytes transmitted on network port). Such time stamp translates

into time units as port speed is fixed but such approach simplifies the code.

Check that m will fit into 16-bit unsigned integer

3.40.3.6 static int __rte_red_drop (const struct rte_red_config * *red_cfg*, struct rte_red * *red*) [static]

make a decision to drop or enqueue a packet based on mark probability criteria

Drop probability (Sally Floyd and Van Jacobson):

pb = (1 / maxp_inv) * (avg - min_th) / (max_th - min_th) pa = pb / (2 - count * pb)

(1 / maxp_inv) * (avg - min_th) ---------- max_th - min_th pa = ---------- count * (1 / maxp_inv) * (avg - min_th) 2 - ---------- max_th - min_th

avg - min_th pa = ---------- 2 * (max_th - min_th) * maxp_inv - count * (avg - min_th)

We define pa_const as: pa_const = 2 * (max_th - min_th) * maxp_inv. Then:

avg - min_th pa = ---------- pa_const - count * (avg - min_th)

Parameters

in	*config*	pointer to structure defining RED parameters
in,out	*data*	pointer to RED runtime data

Returns

operation status

Return values

0	enqueue the packet
1	drop the packet

3.40.3.7 static int rte_red_enqueue_nonempty (const struct rte_red_config * *red_cfg*, struct rte_red * *red*, const unsigned *q*) [static]

Decides if new packet should be enqeued or dropped in queue non-empty case.

Parameters

in	*config*	pointer to a RED configuration parameter structure
in,out	*data*	pointer to RED runtime data
in	*q*	current queue size (measured in packets)

Returns

Operation status

Return values

0	enqueue the packet
1	drop the packet based on max threshold criterion
2	drop the packet based on mark probability criterion

EWMA filter (Sally Floyd and Van Jacobson): avg = (1 - wq) ∗ avg + wq ∗ q avg = avg + q ∗ wq - avg ∗ wq

We select: wq = 2^(-n). Let scaled version of avg be: avg_s = avg ∗ 2^(N+n). We get: avg_s = avg_s + q ∗ 2^N - avg_s ∗ 2^(-n)

By using shift left/right operations, we get: avg_s = avg_s + (q << N) - (avg_s >> n) avg_s += (q << N) - (avg_s >> n)

3.40.3.8 static int rte_red_enqueue (const struct rte_red_config ∗ *red_cfg*, struct rte_red ∗ *red*, const unsigned *q*, const uint64_t *time*) [static]

Decides if new packet should be enqeued or dropped Updates run time data based on new queue size value. Based on new queue average and RED configuration parameters gives verdict whether to enqueue or drop the packet.

Parameters

in	*config*	pointer to a RED configuration parameter structure
in,out	*data*	pointer to RED runtime data
in	*q*	updated queue size in packets
in	*time*	current time stamp

Returns

Operation status

Return values

0	enqueue the packet
1	drop the packet based on max threshold criteria
2	drop the packet based on mark probability criteria

3.40.3.9 static void rte_red_mark_queue_empty (struct rte_red ∗ *red*, const uint64_t *time*) [static]

Callback to records time that queue became empty.

Parameters

in,out	*data*	pointer to RED runtime data
in	*time*	current time stamp

3.40.4 Variable Documentation

3.40.4.1 uint32_t rte_red_rand_val

Externs

3.41 rte_ring.h File Reference

Data Structures

- struct rte_ring
- struct rte_ring::prod
- struct rte_ring::cons

Defines

- #define RTE_RING_NAMESIZE
- #define RING_F_SP_ENQ
- #define RING_F_SC_DEQ
- #define RTE_RING_QUOT_EXCEED
- #define RTE_RING_SZ_MASK

Functions

- struct rte_ring * rte_ring_create (const char *name, unsigned count, int socket_id, unsigned flags)
- int rte_ring_set_water_mark (struct rte_ring *r, unsigned count)
- void rte_ring_dump (const struct rte_ring *r)
- static int rte_ring_mp_enqueue_bulk (struct rte_ring *r, void *const *obj_table, unsigned n)
- static int rte_ring_sp_enqueue_bulk (struct rte_ring *r, void *const *obj_table, unsigned n)
- static int rte_ring_enqueue_bulk (struct rte_ring *r, void *const *obj_table, unsigned n)
- static int rte_ring_mp_enqueue (struct rte_ring *r, void *obj)
- static int rte_ring_sp_enqueue (struct rte_ring *r, void *obj)
- static int rte_ring_enqueue (struct rte_ring *r, void *obj)
- static int rte_ring_mc_dequeue_bulk (struct rte_ring *r, void **obj_table, unsigned n)
- static int rte_ring_sc_dequeue_bulk (struct rte_ring *r, void **obj_table, unsigned n)
- static int rte_ring_dequeue_bulk (struct rte_ring *r, void **obj_table, unsigned n)
- static int rte_ring_mc_dequeue (struct rte_ring *r, void **obj_p)
- static int rte_ring_sc_dequeue (struct rte_ring *r, void **obj_p)

- static int rte_ring_dequeue (struct rte_ring *r, void **obj_p)
- static int rte_ring_full (const struct rte_ring *r)
- static int rte_ring_empty (const struct rte_ring *r)
- static unsigned rte_ring_count (const struct rte_ring *r)
- static unsigned rte_ring_free_count (const struct rte_ring *r)
- void rte_ring_list_dump (void)
- struct rte_ring * rte_ring_lookup (const char *name)
- static int rte_ring_mp_enqueue_burst (struct rte_ring *r, void *const *obj_table, unsigned n)
- static int rte_ring_sp_enqueue_burst (struct rte_ring *r, void *const *obj_table, unsigned n)
- static int rte_ring_enqueue_burst (struct rte_ring *r, void *const *obj_table, unsigned n)
- static int rte_ring_mc_dequeue_burst (struct rte_ring *r, void **obj_table, unsigned n)
- static int rte_ring_sc_dequeue_burst (struct rte_ring *r, void **obj_table, unsigned n)
- static int rte_ring_dequeue_burst (struct rte_ring *r, void **obj_table, unsigned n)

3.41.1 Detailed Description

RTE Ring

The Ring Manager is a fixed-size queue, implemented as a table of pointers. Head and tail pointers are modified atomically, allowing concurrent access to it. It has the following features:

- FIFO (First In First Out)
- Maximum size is fixed; the pointers are stored in a table.
- Lockless implementation.
- Multi- or single-consumer dequeue.
- Multi- or single-producer enqueue.
- Bulk dequeue.
- Bulk enqueue.

Note: the ring implementation is not preemptable. A lcore must not be interrupted by another task that uses the same ring.

3.41.2 Define Documentation

3.41.2.1 #define RTE_RING_NAMESIZE

The maximum length of a ring name.

3.41.2.2 #define RING_F_SP_ENQ

The default enqueue is "single-producer".

3.41.2.3 #define RING_F_SC_DEQ

The default dequeue is "single-consumer".

3.41.2.4 #define RTE_RING_QUOT_EXCEED

Quota exceed for burst ops

3.41.2.5 #define RTE_RING_SZ_MASK

Ring size mask

3.41.3 Function Documentation

3.41.3.1 struct rte_ring* rte_ring_create (const char * *name*, unsigned *count*, int *socket_id*, unsigned *flags*) [read]

Create a new ring named *name* in memory.

This function uses “memzone_reserve()“ to allocate memory. Its size is set to *count*, which must be a power of two. Water marking is disabled by default. Note that the real usable ring size is *count-1* instead of *count*.

Parameters

name	The name of the ring.
count	The size of the ring (must be a power of 2).
socket_id	The *socket_id* argument is the socket identifier in case of NUMA. The value can be *SOCKET_ID_ANY* if there is no NUMA constraint for the reserved zone.
flags	An OR of the following: • RING_F_SP_ENQ: If this flag is set, the default behavior when using “rte_ring_enqueue()“ or “rte_ring_enqueue_bulk()“ is "single-producer". Otherwise, it is "multi-producers". • RING_F_SC_DEQ: If this flag is set, the default behavior when using “rte_ring_dequeue()“ or “rte_ring_dequeue_bulk()“ is "single-consumer". Otherwise, it is "multi-consumers".

Returns

On success, the pointer to the new allocated ring. NULL on error with rte_errno set appropriately. Possible 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
- E_RTE_NO_TAILQ - no tailq list could be got for the ring list

- EINVAL - count provided is not a power of 2
- 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

3.41.3.2 int rte_ring_set_water_mark (struct rte_ring * r, unsigned count)

Change the high water mark.

If *count* is 0, water marking is disabled. Otherwise, it is set to the *count* value. The *count* value must be greater than 0 and less than the ring size.

This function can be called at any time (not necessarily at initialization).

Parameters

r	A pointer to the ring structure.
count	The new water mark value.

Returns

- 0: Success; water mark changed.
- -EINVAL: Invalid water mark value.

3.41.3.3 void rte_ring_dump (const struct rte_ring * r)

Dump the status of the ring to the console.

Parameters

r	A pointer to the ring structure.

3.41.3.4 static int rte_ring_mp_enqueue_bulk (struct rte_ring * r, void *const * obj_table, unsigned n) [static]

Enqueue several objects on the ring (multi-producers safe).

This function uses a "compare and set" instruction to move the producer index atomically.

Parameters

r	A pointer to the ring structure.
obj_table	A pointer to a table of void * pointers (objects).
n	The number of objects to add in the ring from the obj_table.

Returns

- 0: Success; objects enqueue.
- -EDQUOT: Quota exceeded. The objects have been enqueued, but the high water mark is exceeded.
- -ENOBUFS: Not enough room in the ring to enqueue, no object is enqueued.

3.41.3.5 static int rte_ring_sp_enqueue_bulk (struct rte_ring * *r*, void *const * *obj_table*, unsigned *n*) [static]

Enqueue several objects on a ring (NOT multi-producers safe).

Parameters

r	A pointer to the ring structure.
obj_table	A pointer to a table of void * pointers (objects).
n	The number of objects to add in the ring from the obj_table.

Returns

- 0: Success; objects enqueued.
- -EDQUOT: Quota exceeded. The objects have been enqueued, but the high water mark is exceeded.
- -ENOBUFS: Not enough room in the ring to enqueue; no object is enqueued.

3.41.3.6 static int rte_ring_enqueue_bulk (struct rte_ring * *r*, void *const * *obj_table*, unsigned *n*) [static]

Enqueue several objects on a ring.

This function calls the multi-producer or the single-producer version depending on the default behavior that was specified at ring creation time (see flags).

Parameters

r	A pointer to the ring structure.
obj_table	A pointer to a table of void * pointers (objects).
n	The number of objects to add in the ring from the obj_table.

Returns

- 0: Success; objects enqueued.
- -EDQUOT: Quota exceeded. The objects have been enqueued, but the high water mark is exceeded.
- -ENOBUFS: Not enough room in the ring to enqueue; no object is enqueued.

3.41.3.7 static int rte_ring_mp_enqueue (struct rte_ring * r, void * obj) [static]

Enqueue one object on a ring (multi-producers safe).

This function uses a "compare and set" instruction to move the producer index atomically.

Parameters

r	A pointer to the ring structure.
obj	A pointer to the object to be added.

Returns

- 0: Success; objects enqueued.
- -EDQUOT: Quota exceeded. The objects have been enqueued, but the high water mark is exceeded.
- -ENOBUFS: Not enough room in the ring to enqueue; no object is enqueued.

3.41.3.8 static int rte_ring_sp_enqueue (struct rte_ring * r, void * obj) [static]

Enqueue one object on a ring (NOT multi-producers safe).

Parameters

r	A pointer to the ring structure.
obj	A pointer to the object to be added.

Returns

- 0: Success; objects enqueued.
- -EDQUOT: Quota exceeded. The objects have been enqueued, but the high water mark is exceeded.
- -ENOBUFS: Not enough room in the ring to enqueue; no object is enqueued.

3.41.3.9 static int rte_ring_enqueue (struct rte_ring * r, void * obj) [static]

Enqueue one object on a ring.

This function calls the multi-producer or the single-producer version, depending on the default behaviour that was specified at ring creation time (see flags).

Parameters

r	A pointer to the ring structure.
obj	A pointer to the object to be added.

Returns

- 0: Success; objects enqueued.
- -EDQUOT: Quota exceeded. The objects have been enqueued, but the high water mark is exceeded.
- -ENOBUFS: Not enough room in the ring to enqueue; no object is enqueued.

3.41.3.10 static int rte_ring_mc_dequeue_bulk (struct rte_ring * r, void ** obj_table, unsigned n) [static]

Dequeue several objects from a ring (multi-consumers safe).

This function uses a "compare and set" instruction to move the consumer index atomically.

Parameters

r	A pointer to the ring structure.
obj_table	A pointer to a table of void * pointers (objects) that will be filled.
n	The number of objects to dequeue from the ring to the obj_table.

Returns

- 0: Success; objects dequeued.
- -ENOENT: Not enough entries in the ring to dequeue; no object is dequeued.

3.41.3.11 static int rte_ring_sc_dequeue_bulk (struct rte_ring * r, void ** obj_table, unsigned n) [static]

Dequeue several objects from a ring (NOT multi-consumers safe).

Parameters

r	A pointer to the ring structure.
obj_table	A pointer to a table of void * pointers (objects) that will be filled.
n	The number of objects to dequeue from the ring to the obj_table, must be strictly positive.

Returns

- 0: Success; objects dequeued.
- -ENOENT: Not enough entries in the ring to dequeue; no object is dequeued.

3.41.3.12 static int rte_ring_dequeue_bulk (struct rte_ring * r, void ** obj_table, unsigned n) [static]

Dequeue several objects from a ring.

This function calls the multi-consumers or the single-consumer version, depending on the default behaviour that was specified at ring creation time (see flags).

Parameters

r	A pointer to the ring structure.
obj_table	A pointer to a table of void * pointers (objects) that will be filled.
n	The number of objects to dequeue from the ring to the obj_table.

Returns

- 0: Success; objects dequeued.
- -ENOENT: Not enough entries in the ring to dequeue, no object is dequeued.

3.41.3.13 static int rte_ring_mc_dequeue (struct rte_ring * *r*, void ** *obj_p*) [static]

Dequeue one object from a ring (multi-consumers safe).

This function uses a "compare and set" instruction to move the consumer index atomically.

Parameters

r	A pointer to the ring structure.
obj_p	A pointer to a void * pointer (object) that will be filled.

Returns

- 0: Success; objects dequeued.
- -ENOENT: Not enough entries in the ring to dequeue; no object is dequeued.

3.41.3.14 static int rte_ring_sc_dequeue (struct rte_ring * *r*, void ** *obj_p*) [static]

Dequeue one object from a ring (NOT multi-consumers safe).

Parameters

r	A pointer to the ring structure.
obj_p	A pointer to a void * pointer (object) that will be filled.

Returns

- 0: Success; objects dequeued.
- -ENOENT: Not enough entries in the ring to dequeue, no object is dequeued.

3.41.3.15 static int rte_ring_dequeue (struct rte_ring * *r*, void ** *obj_p*) [static]

Dequeue one object from a ring.

This function calls the multi-consumers or the single-consumer version depending on the default behaviour that was specified at ring creation time (see flags).

Parameters

r	A pointer to the ring structure.
obj_p	A pointer to a void * pointer (object) that will be filled.

Returns

- 0: Success, objects dequeued.
- -ENOENT: Not enough entries in the ring to dequeue, no object is dequeued.

3.41.3.16 static int rte_ring_full (const struct rte_ring * r) [static]

Test if a ring is full.

Parameters

r	A pointer to the ring structure.

Returns

- 1: The ring is full.
- 0: The ring is not full.

3.41.3.17 static int rte_ring_empty (const struct rte_ring * r) [static]

Test if a ring is empty.

Parameters

r	A pointer to the ring structure.

Returns

- 1: The ring is empty.
- 0: The ring is not empty.

3.41.3.18 static unsigned rte_ring_count (const struct rte_ring * r) [static]

Return the number of entries in a ring.

Parameters

r	A pointer to the ring structure.

Returns

The number of entries in the ring.

3.41.3.19 static unsigned rte_ring_free_count (const struct rte_ring * r) [static]

Return the number of free entries in a ring.

Parameters

r	A pointer to the ring structure.

Returns

The number of free entries in the ring.

3.41.3.20 void rte_ring_list_dump (void)

Dump the status of all rings on the console

3.41.3.21 struct rte_ring* rte_ring_lookup (const char * *name*) [read]

Search a ring from its name

Parameters

name	The name of the ring.

Returns

The pointer to the ring matching the name, or NULL if not found, with rte_errno set appropriately. - Possible rte_errno values include:

- ENOENT - required entry not available to return.

3.41.3.22 static int rte_ring_mp_enqueue_burst (struct rte_ring * *r*, void *const * *obj_table*, unsigned *n*) [static]

Enqueue several objects on the ring (multi-producers safe).

This function uses a "compare and set" instruction to move the producer index atomically.

Parameters

r	A pointer to the ring structure.
obj_table	A pointer to a table of void * pointers (objects).
n	The number of objects to add in the ring from the obj_table.

Returns

- n: Actual number of objects enqueued.

3.41.3.23 static int rte_ring_sp_enqueue_burst (struct rte_ring * *r*, void *const * *obj_table*, unsigned *n*) [static]

Enqueue several objects on a ring (NOT multi-producers safe).

Parameters

r	A pointer to the ring structure.
obj_table	A pointer to a table of void * pointers (objects).
n	The number of objects to add in the ring from the obj_table.

Returns

- n: Actual number of objects enqueued.

3.41.3.24 static int rte_ring_enqueue_burst (struct rte_ring * *r*, void *const * *obj_table*, unsigned *n*) [static]

Enqueue several objects on a ring.

This function calls the multi-producer or the single-producer version depending on the default behavior that was specified at ring creation time (see flags).

Parameters

r	A pointer to the ring structure.
obj_table	A pointer to a table of void * pointers (objects).
n	The number of objects to add in the ring from the obj_table.

Returns

- n: Actual number of objects enqueued.

3.41.3.25 static int rte_ring_mc_dequeue_burst (struct rte_ring * *r*, void ** *obj_table*, unsigned *n*) [static]

Dequeue several objects from a ring (multi-consumers safe). When the request objects are more than the available objects, only dequeue the actual number of objects

This function uses a "compare and set" instruction to move the consumer index atomically.

Parameters

r	A pointer to the ring structure.
obj_table	A pointer to a table of void * pointers (objects) that will be filled.
n	The number of objects to dequeue from the ring to the obj_table.

Returns

- n: Actual number of objects dequeued, 0 if ring is empty

3.41.3.26 static int rte_ring_sc_dequeue_burst (struct rte_ring * *r*, void ** *obj_table*, unsigned *n*) [static]

Dequeue several objects from a ring (NOT multi-consumers safe).When the request objects are more than the available objects, only dequeue the actual number of objects

Parameters

r	A pointer to the ring structure.
obj_table	A pointer to a table of void * pointers (objects) that will be filled.
n	The number of objects to dequeue from the ring to the obj_table.

Returns

- n: Actual number of objects dequeued, 0 if ring is empty

3.41.3.27 static int rte_ring_dequeue_burst (struct rte_ring * *r*, void ** *obj_table*, unsigned *n*) [static]

Dequeue multiple objects from a ring up to a maximum number.

This function calls the multi-consumers or the single-consumer version, depending on the default behaviour that was specified at ring creation time (see flags).

Parameters

r	A pointer to the ring structure.
obj_table	A pointer to a table of void * pointers (objects) that will be filled.
n	The number of objects to dequeue from the ring to the obj_table.

Returns

- Number of objects dequeued, or a negative error code on error

3.42 rte_rwlock.h File Reference

Data Structures

- struct rte_rwlock_t

Defines

- #define RTE_RWLOCK_INITIALIZER

Functions

- static void rte_rwlock_init (rte_rwlock_t *rwl)
- static void rte_rwlock_read_lock (rte_rwlock_t *rwl)
- static void rte_rwlock_read_unlock (rte_rwlock_t *rwl)
- static void rte_rwlock_write_lock (rte_rwlock_t *rwl)
- static void rte_rwlock_write_unlock (rte_rwlock_t *rwl)

3.42.1 Detailed Description

RTE Read-Write Locks

This file defines an API for read-write locks. The lock is used to protect data that allows multiple readers in parallel, but only one writer. All readers are blocked until the writer is finished writing.

3.42.2 Define Documentation

3.42.2.1 #define RTE_RWLOCK_INITIALIZER

A static rwlock initializer.

3.42.3 Function Documentation

3.42.3.1 static void rte_rwlock_init (rte_rwlock_t * *rwl*) [static]

Initialize the rwlock to an unlocked state.

Parameters

rwl	A pointer to the rwlock structure.

3.42.3.2 static void rte_rwlock_read_lock (rte_rwlock_t * *rwl*) [static]

Take a read lock. Loop until the lock is held.

Parameters

rwl	A pointer to a rwlock structure.

3.42.3.3 static void rte_rwlock_read_unlock (rte_rwlock_t * *rwl*) [static]

Release a read lock.

Parameters

rwl	A pointer to the rwlock structure.

3.42.3.4 static void rte_rwlock_write_lock (rte_rwlock_t * *rwl*) [static]

Take a write lock. Loop until the lock is held.

Parameters

rwl	A pointer to a rwlock structure.

3.42.3.5 static void rte_rwlock_write_unlock (rte_rwlock_t * *rwl*) [static]

Release a write lock.

Parameters

rwl	A pointer to a rwlock structure.

3.43 rte_sched.h File Reference

Data Structures

- struct rte_sched_subport_params
- struct rte_sched_subport_stats

- struct rte_sched_pipe_params
- struct rte_sched_queue_stats
- struct rte_sched_port_params
- struct rte_sched_port_hierarchy

Defines

- #define RTE_SCHED_TRAFFIC_CLASSES_PER_PIPE
- #define RTE_SCHED_QUEUES_PER_TRAFFIC_CLASS
- #define RTE_SCHED_QUEUES_PER_PIPE
- #define RTE_SCHED_PIPE_PROFILES_PER_PORT
- #define RTE_SCHED_FRAME_OVERHEAD_DEFAULT

Functions

- struct rte_sched_port * rte_sched_port_config (struct rte_sched_port_params *params)
- void rte_sched_port_free (struct rte_sched_port *port)
- int rte_sched_subport_config (struct rte_sched_port *port, uint32_t subport_id, struct rte_sched_subport_params *params)
- int rte_sched_pipe_config (struct rte_sched_port *port, uint32_t subport_id, uint32_t pipe_id, int32_t pipe_profile)
- uint32_t rte_sched_port_get_memory_footprint (struct rte_sched_port_params *params)
- int rte_sched_subport_read_stats (struct rte_sched_port *port, uint32_t subport_id, struct rte_sched_subport_stats *stats, uint32_t *tc_ov)
- int rte_sched_queue_read_stats (struct rte_sched_port *port, uint32_t queue_id, struct rte_sched_queue_stats *stats, uint16_t *qlen)
- static void rte_sched_port_pkt_write (struct rte_mbuf *pkt, uint32_t subport, uint32_t pipe, uint32_t traffic_class, uint32_t queue, enum rte_meter_color color)
- static void rte_sched_port_pkt_read_tree_path (struct rte_mbuf *pkt, uint32_t *subport, uint32_t *pipe, uint32_t *traffic_class, uint32_t *queue)
- int rte_sched_port_enqueue (struct rte_sched_port *port, struct rte_mbuf **pkts, uint32_t n_pkts)
- int rte_sched_port_dequeue (struct rte_sched_port *port, struct rte_mbuf **pkts, uint32_t n_pkts)

3.43.1 Detailed Description

RTE Hierarchical Scheduler

The hierarchical scheduler prioritizes the transmission of packets from different users and traffic classes according to the Service Level Agreements (SLAs) defined for the current network node.

The scheduler supports thousands of packet queues grouped under a 5-level hierarchy: 1. Port:

- Typical usage: output Ethernet port;
- Multiple ports are scheduled in round robin order with equal priority; 2. Subport:
- Typical usage: group of users;

- Traffic shaping using the token bucket algorithm (one bucket per subport);
- Upper limit enforced per traffic class at subport level;
- Lower priority traffic classes able to reuse subport bandwidth currently unused by higher priority traffic classes of the same subport;
- When any subport traffic class is oversubscribed (configuration time event), the usage of subport member pipes with high demand for that traffic class pipes is truncated to a dynamically adjusted value with no impact to low demand pipes; 3. Pipe:
- Typical usage: individual user/subscriber;
- Traffic shaping using the token bucket algorithm (one bucket per pipe); 4. Traffic class:
- Traffic classes of the same pipe handled in strict priority order;
- Upper limit enforced per traffic class at the pipe level;
- Lower priority traffic classes able to reuse pipe bandwidth currently unused by higher priority traffic classes of the same pipe; 5. Queue:
- Typical usage: queue hosting packets from one or multiple connections of same traffic class belonging to the same user;
- Weighted Round Robin (WRR) is used to service the queues within same pipe traffic class.

3.43.2 Define Documentation

3.43.2.1 #define RTE_SCHED_TRAFFIC_CLASSES_PER_PIPE

Random Early Detection (RED) Number of traffic classes per pipe (as well as subport). Cannot be changed.

3.43.2.2 #define RTE_SCHED_QUEUES_PER_TRAFFIC_CLASS

Number of queues per pipe traffic class. Cannot be changed.

3.43.2.3 #define RTE_SCHED_QUEUES_PER_PIPE

Number of queues per pipe.

3.43.2.4 #define RTE_SCHED_PIPE_PROFILES_PER_PORT

Maximum number of pipe profiles that can be defined per port. Compile-time configurable.

3.43.2.5 #define RTE_SCHED_FRAME_OVERHEAD_DEFAULT

Ethernet framing overhead. Overhead fields per Ethernet frame: 1. Preamble: 7 bytes; 2. Start of Frame Delimiter (SFD): 1 byte; 3. Frame Check Sequence (FCS): 4 bytes; 4. Inter Frame Gap (IFG): 12 bytes. The FCS is considered overhead only if not included in the packet length (field pkt.pkt_len of struct rte_mbuf).

3.43.3 Function Documentation

3.43.3.1 struct rte_sched_port* rte_sched_port_config (struct rte_sched_port_params * *params*) [read]

Hierarchical scheduler port configuration

Parameters

params	Port scheduler configuration parameter structure

Returns

Handle to port scheduler instance upon success or NULL otherwise.

3.43.3.2 void rte_sched_port_free (struct rte_sched_port * *port*)

Hierarchical scheduler port free

Parameters

port	Handle to port scheduler instance

3.43.3.3 int rte_sched_subport_config (struct rte_sched_port * *port*, uint32_t *subport_id*, struct rte_sched_subport_params * *params*)

Hierarchical scheduler subport configuration

Parameters

port	Handle to port scheduler instance
subport_id	Subport ID
params	Subport configuration parameters

Returns

0 upon success, error code otherwise

3.43.3.4 int rte_sched_pipe_config (struct rte_sched_port * *port*, uint32_t *subport_id*, uint32_t *pipe_id*, int32_t *pipe_profile*)

Hierarchical scheduler pipe configuration

Parameters

port	Handle to port scheduler instance
subport_id	Subport ID
pipe_id	Pipe ID within subport
pipe_profile	ID of port-level pre-configured pipe profile

Returns

0 upon success, error code otherwise

3.43.3.5 uint32_t rte_sched_port_get_memory_footprint (struct rte_sched_port_params * *params*)

Hierarchical scheduler memory footprint size per port

Parameters

params	Port scheduler configuration parameter structure

Returns

Memory footprint size in bytes upon success, 0 otherwise

3.43.3.6 int rte_sched_subport_read_stats (struct rte_sched_port * *port*, uint32_t *subport_id*, struct rte_sched_subport_stats * *stats*, uint32_t * *tc_ov*)

Hierarchical scheduler subport statistics read

Parameters

port	Handle to port scheduler instance
subport_id	Subport ID
stats	Pointer to pre-allocated subport statistics structure where the statistics counters should be stored
tc_ov	Pointer to pre-allocated 4-entry array where the oversubscription status for each of the 4 subport traffic classes should be stored.

Returns

0 upon success, error code otherwise

3.43.3.7 int rte_sched_queue_read_stats (struct rte_sched_port * *port*, uint32_t *queue_id*, struct rte_sched_queue_stats * *stats*, uint16_t * *qlen*)

Hierarchical scheduler queue statistics read

Parameters

port	Handle to port scheduler instance
queue_id	Queue ID within port scheduler
stats	Pointer to pre-allocated subport statistics structure where the statistics counters should be stored
qlen	Pointer to pre-allocated variable where the current queue length should be stored.

Returns

0 upon success, error code otherwise

3.43.3.8 static void rte_sched_port_pkt_write (struct rte_mbuf * *pkt*, uint32_t *subport*, uint32_t *pipe*, uint32_t *traffic_class*, uint32_t *queue*, enum rte_meter_color *color*) [static]

Scheduler hierarchy path write to packet descriptor. Typically called by the packet classification stage.

Parameters

pkt	Packet descriptor handle
subport	Subport ID
pipe	Pipe ID within subport
traffic_class	Traffic class ID within pipe (0 .. 3)
queue	Queue ID within pipe traffic class (0 .. 3)

3.43.3.9 static void rte_sched_port_pkt_read_tree_path (struct rte_mbuf * *pkt*, uint32_t * *subport*, uint32_t * *pipe*, uint32_t * *traffic_class*, uint32_t * *queue*) [static]

Scheduler hierarchy path read from packet descriptor (struct rte_mbuf). Typically called as part of the hierarchical scheduler enqueue operation. The subport, pipe, traffic class and queue parameters need to be pre-allocated by the caller.

Parameters

pkt	Packet descriptor handle
subport	Subport ID
pipe	Pipe ID within subport
traffic_class	Traffic class ID within pipe (0 .. 3)
queue	Queue ID within pipe traffic class (0 .. 3)

3.43.3.10 int rte_sched_port_enqueue (struct rte_sched_port * port, struct rte_mbuf ** pkts, uint32_t n_pkts)

Hierarchical scheduler port enqueue. Writes up to n_pkts to port scheduler and returns the number of packets actually written. For each packet, the port scheduler queue to write the packet to is identified by reading the hierarchy path from the packet descriptor; if the queue is full or congested and the packet is not written to the queue, then the packet is automatically dropped without any action required from the caller.

Parameters

port	Handle to port scheduler instance
pkts	Array storing the packet descriptor handles
n_pkts	Number of packets to enqueue from the pkts array into the port scheduler

Returns

Number of packets successfully enqueued

3.43.3.11 int rte_sched_port_dequeue (struct rte_sched_port * port, struct rte_mbuf ** pkts, uint32_t n_pkts)

Hierarchical scheduler port dequeue. Reads up to n_pkts from the port scheduler and stores them in the pkts array and returns the number of packets actually read. The pkts array needs to be pre-allocated by the caller with at least n_pkts entries.

Parameters

port	Handle to port scheduler instance
pkts	Pre-allocated packet descriptor array where the packets dequeued from the port scheduler should be stored
n_pkts	Number of packets to dequeue from the port scheduler

Returns

Number of packets successfully dequeued and placed in the pkts array

3.44 rte_sctp.h File Reference

Data Structures

- struct sctp_hdr

3.44.1 Detailed Description

SCTP-related defines

3.45 rte_spinlock.h File Reference

Data Structures

- struct rte_spinlock_t
- struct rte_spinlock_recursive_t

Defines

- #define RTE_SPINLOCK_INITIALIZER
- #define RTE_SPINLOCK_RECURSIVE_INITIALIZER

Functions

- static void rte_spinlock_init (rte_spinlock_t *sl)
- static void rte_spinlock_lock (rte_spinlock_t *sl)
- static void rte_spinlock_unlock (rte_spinlock_t *sl)
- static int rte_spinlock_trylock (rte_spinlock_t *sl)
- static int rte_spinlock_is_locked (rte_spinlock_t *sl)
- static void rte_spinlock_recursive_init (rte_spinlock_recursive_t *slr)
- static void rte_spinlock_recursive_lock (rte_spinlock_recursive_t *slr)
- static void rte_spinlock_recursive_unlock (rte_spinlock_recursive_t *slr)
- static int rte_spinlock_recursive_trylock (rte_spinlock_recursive_t *slr)

3.45.1 Detailed Description

RTE Spinlocks

This file defines an API for read-write locks, which are implemented in an architecture-specific way. This kind of lock simply waits in a loop repeatedly checking until the lock becomes available.

All locks must be initialised before use, and only initialised once.

3.45.2 Define Documentation

3.45.2.1 #define RTE_SPINLOCK_INITIALIZER

A static spinlock initializer.

3.45.2.2 #define RTE_SPINLOCK_RECURSIVE_INITIALIZER

A static recursive spinlock initializer.

3.45.3 Function Documentation

3.45.3.1 static void rte_spinlock_init (rte_spinlock_t * sl) [static]

Initialize the spinlock to an unlocked state.

Parameters

sl	A pointer to the spinlock.

3.45.3.2 static void rte_spinlock_lock (rte_spinlock_t * sl) [static]

Take the spinlock.

Parameters

sl	A pointer to the spinlock.

3.45.3.3 static void rte_spinlock_unlock (rte_spinlock_t * sl) [static]

Release the spinlock.

Parameters

sl	A pointer to the spinlock.

3.45.3.4 static int rte_spinlock_trylock (rte_spinlock_t * sl) [static]

Try to take the lock.

Parameters

sl	A pointer to the spinlock.

Returns

1 if the lock is successfully taken; 0 otherwise.

3.45.3.5 static int rte_spinlock_is_locked (rte_spinlock_t * sl) [static]

Test if the lock is taken.

Parameters

sl	A pointer to the spinlock.

Returns

1 if the lock is currently taken; 0 otherwise.

3.45.3.6 static void rte_spinlock_recursive_init (rte_spinlock_recursive_t * *slr*) [static]

Initialize the recursive spinlock to an unlocked state.

Parameters

slr	A pointer to the recursive spinlock.

3.45.3.7 static void rte_spinlock_recursive_lock (rte_spinlock_recursive_t * *slr*) [static]

Take the recursive spinlock.

Parameters

slr	A pointer to the recursive spinlock.

3.45.3.8 static void rte_spinlock_recursive_unlock (rte_spinlock_recursive_t * *slr*) [static]

Release the recursive spinlock.

Parameters

slr	A pointer to the recursive spinlock.

3.45.3.9 static int rte_spinlock_recursive_trylock (rte_spinlock_recursive_t * *slr*) [static]

Try to take the recursive lock.

Parameters

slr	A pointer to the recursive spinlock.

Returns

1 if the lock is successfully taken; 0 otherwise.

3.46 rte_string_fns.h File Reference

Functions

- static int rte_snprintf (char *buffer, int buflen, const char *format,...)
- static int rte_strsplit (char *string, int stringlen, char **tokens, int maxtokens, char delim)

3.46.1 Detailed Description

Definitions of warnings for use of various insecure functions

3.46.2 Function Documentation

3.46.2.1 static int rte_snprintf (char * *buffer*, int *buflen*, const char * *format*, ...) [static]

Safer version of snprintf that writes up to buflen characters to the output buffer and ensures that the resultant string is null-terminated, that is, it writes at most buflen-1 actual string characters to buffer. The return value is the number of characters which should be written to the buffer, so string truncation can be detected by the caller by checking if the return value is greater than or equal to the buflen.

Parameters

buffer	The buffer into which the output is to be written
buflen	The size of the output buffer
format	The format string to be printed to the buffer

Returns

The number of characters written to the buffer, or if the string has been truncated, the number of characters which would have been written had the buffer been sufficiently big.

3.46.2.2 static int rte_strsplit (char * *string*, int *stringlen*, char ** *tokens*, int *maxtokens*, char *delim*) [static]

Takes string "string" parameter and splits it at character "delim" up to maxtokens-1 times - to give "maxtokens" resulting tokens. Like strtok or strsep functions, this modifies its input string, by replacing instances of "delim" with '\0'. All resultant tokens are returned in the "tokens" array which must have enough entries to hold "maxtokens".

Parameters

string	The input string to be split into tokens
stringlen	The max length of the input buffer
tokens	The array to hold the pointers to the tokens in the string
maxtokens	The number of elements in the tokens array. At most, maxtokens-1 splits of the string will be done.
delim	The character on which the split of the data will be done

Returns

The number of tokens in the tokens array.

3.47 rte_tailq.h File Reference

Data Structures

- struct rte_dummy
- struct rte_tailq_head

Defines

- #define RTE_TAILQ_RESERVE(name, struct_name)
- #define RTE_TAILQ_RESERVE_BY_IDX(idx, struct_name)
- #define RTE_TAILQ_LOOKUP(name, struct_name)
- #define RTE_TAILQ_LOOKUP_BY_IDX(idx, struct_name)

Functions

- TAILQ_HEAD (rte_dummy_head, rte_dummy)
- struct rte_tailq_head * rte_eal_tailq_reserve (const char *name)
- struct rte_tailq_head * rte_eal_tailq_reserve_by_idx (const unsigned idx)
- void rte_dump_tailq (void)
- struct rte_tailq_head * rte_eal_tailq_lookup (const char *name)
- struct rte_tailq_head * rte_eal_tailq_lookup_by_idx (const unsigned idx)

3.47.1 Detailed Description

Here defines rte_tailq APIs for only internal use

3.47.2 Define Documentation

3.47.2.1 #define RTE_TAILQ_RESERVE(*name, struct_name*)

Utility macro to make reserving a tailqueue for a particular struct easier.

Parameters

name	The name to be given to the tailq - used by lookup to find it later
struct_name	The name of the list type we are using. (Generally this is the same as the first parameter passed to TAILQ_HEAD macro)

Returns

The return value from rte_eal_tailq_reserve, typecast to the appropriate structure pointer type. NULL on error, since the tailq_head is the first element in the rte_tailq_head structure.

3.47.2.2 #define RTE_TAILQ_RESERVE_BY_IDX(*idx, struct_name*)

Utility macro to make reserving a tailqueue for a particular struct easier.

Parameters

idx	The tailq idx defined in rte_tail_t to be given to the tail queue. • used by lookup to find it later
struct_name	The name of the list type we are using. (Generally this is the same as the first parameter passed to TAILQ_HEAD macro)

Returns

The return value from rte_eal_tailq_reserve, typecast to the appropriate structure pointer type. NULL on error, since the tailq_head is the first element in the rte_tailq_head structure.

3.47.2.3 #define RTE_TAILQ_LOOKUP(*name, struct_name*)

Utility macro to make looking up a tailqueue for a particular struct easier.

Parameters

name	The name of tailq
struct_name	The name of the list type we are using. (Generally this is the same as the first parameter passed to TAILQ_HEAD macro)

Returns

The return value from rte_eal_tailq_lookup, typecast to the appropriate structure pointer type. NULL on error, since the tailq_head is the first element in the rte_tailq_head structure.

3.47.2.4 #define RTE_TAILQ_LOOKUP_BY_IDX(*idx, struct_name*)

Utility macro to make looking up a tailqueue for a particular struct easier.

Parameters

idx	The tailq idx defined in rte_tail_t to be given to the tail queue.
struct_name	The name of the list type we are using. (Generally this is the same as the first parameter passed to TAILQ_HEAD macro)

Returns

The return value from rte_eal_tailq_lookup, typecast to the appropriate structure pointer type. NULL on error, since the tailq_head is the first element in the rte_tailq_head structure.

3.47.3 Function Documentation

3.47.3.1 TAILQ_HEAD (rte_dummy_head , rte_dummy)

dummy

3.47.3.2 struct rte_tailq_head* rte_eal_tailq_reserve (const char * *name*) [read]

Reserve a slot in the tailq list for a particular tailq header Note: this function, along with rte_tailq_lookup, is not multi-thread safe, and both these functions should only be called from a single thread at a time

Parameters

name	The name to be given to the tail queue.

Returns

A pointer to the newly reserved tailq entry

3.47.3.3 struct rte_tailq_head* rte_eal_tailq_reserve_by_idx (const unsigned *idx*) [read]

Reserve a slot in the tailq list for a particular tailq header Note: this function, along with rte_tailq_lookup, is not multi-thread safe, and both these functions should only be called from a single thread at a time

Parameters

idx	The tailq idx defined in rte_tail_t to be given to the tail queue.

Returns

A pointer to the newly reserved tailq entry

3.47.3.4 void rte_dump_tailq (void)

Dump tail queues to the console.

3.47.3.5 struct rte_tailq_head* rte_eal_tailq_lookup (const char * *name*) [read]

Lookup for a tail queue.

Get a pointer to a tail queue header of an already reserved tail queue identified by the name given as an argument. Note: this function, along with rte_tailq_reserve, is not multi-thread safe, and both these functions should only be called from a single thread at a time

Parameters

name	The name of the queue.

Returns

A pointer to the tail queue head structure.

3.47.3.6 struct rte_tailq_head* rte_eal_tailq_lookup_by_idx (const unsigned *idx*) [read]

Lookup for a tail queue.

Get a pointer to a tail queue header of an already reserved tail queue identified by the name given as an argument. Note: this function, along with rte_tailq_reserve, is not multi-thread safe, and both these functions should only be called from a single thread at a time

Parameters

idx	The tailq idx defined in rte_tail_t to be given to the tail queue.

Returns

A pointer to the tail queue head structure.

3.48 rte_tailq_elem.h File Reference

3.48.1 Detailed Description

This file contains the type of the tailq elem recognised by DPDK, which can be used to fill out an array of structures describing the tailq.

In order to populate an array, the user of this file must define this macro: rte_tailq_elem(idx, name). For example:

```
enum rte_tailq_t {
#define rte_tailq_elem(idx, name)     idx,
#define rte_tailq_end(idx)            idx
#include <rte_tailq_elem.h>
};

const char* rte_tailq_names[RTE_MAX_TAILQ] = {
#define rte_tailq_elem(idx, name)     name,
#include <rte_tailq_elem.h>
};
```

Note that this file can be included multiple times within the same file.

3.49 rte_tcp.h File Reference

Data Structures

- struct tcp_hdr

3.49.1 Detailed Description

TCP-related defines

3.50 rte_timer.h File Reference

Data Structures

- union rte_timer_status
- struct rte_timer

Defines

- #define RTE_TIMER_STOP
- #define RTE_TIMER_PENDING
- #define RTE_TIMER_RUNNING
- #define RTE_TIMER_CONFIG
- #define RTE_TIMER_NO_OWNER
- #define RTE_TIMER_INITIALIZER

Typedefs

- typedef void(rte_timer_cb_t)(struct rte_timer *, void *)

Enumerations

- enum rte_timer_type

Functions

- void rte_timer_subsystem_init (void)
- void rte_timer_init (struct rte_timer *tim)
- int rte_timer_reset (struct rte_timer *tim, uint64_t ticks, enum rte_timer_type type, unsigned tim_lcore, rte_timer_cb_t fct, void *arg)
- void rte_timer_reset_sync (struct rte_timer *tim, uint64_t ticks, enum rte_timer_type type, unsigned tim_lcore, rte_timer_cb_t fct, void *arg)
- int rte_timer_stop (struct rte_timer *tim)
- void rte_timer_stop_sync (struct rte_timer *tim)
- int rte_timer_pending (struct rte_timer *tim)
- void rte_timer_manage (void)
- void rte_timer_dump_stats (void)

3.50.1 Detailed Description

RTE Timer

This library provides a timer service to RTE Data Plane execution units that allows the execution of callback functions asynchronously.

- Timers can be periodic or single (one-shot).
- The timers can be loaded from one core and executed on another. This has to be specified in the call to rte_timer_reset().
- High precision is possible. NOTE: this depends on the call frequency to rte_timer_manage() that check the timer expiration for the local core.
- If not used in an application, for improved performance, it can be disabled at compilation time by not calling the rte_timer_manage() to improve performance.

The timer library uses the rte_get_hpet_cycles() function that uses the HPET, when available, to provide a reliable time reference. [HPET routines are provided by EAL, which falls back to using the chip TSC (time-stamp counter) as fallback when HPET is not available]

This library provides an interface to add, delete and restart a timer. The API is based on the BSD callout(9) API with a few differences.

See the RTE architecture documentation for more information about the design of this library.

3.50.2 Define Documentation

3.50.2.1 #define RTE_TIMER_STOP

State: timer is stopped.

3.50.2.2 #define RTE_TIMER_PENDING

State: timer is scheduled.

3.50.2.3 #define RTE_TIMER_RUNNING

State: timer function is running.

3.50.2.4 #define RTE_TIMER_CONFIG

State: timer is being configured.

3.50.2.5 #define RTE_TIMER_NO_OWNER

Timer has no owner.

3.50.2.6 #define RTE_TIMER_INITIALIZER

A static initializer for a timer structure.

3.50.3 Typedef Documentation

3.50.3.1 typedef void(rte_timer_cb_t)(struct rte_timer *, void *)

Callback function type for timer expiry.

3.50.4 Enumeration Type Documentation

3.50.4.1 enum rte_timer_type

Timer type: Periodic or single (one-shot).

3.50.5 Function Documentation

3.50.5.1 void rte_timer_subsystem_init (void)

Initialize the timer library.

Initializes internal variables (list, locks and so on) for the RTE timer library.

3.50.5.2 void rte_timer_init (struct rte_timer * tim)

Initialize a timer handle.

The rte_timer_init() function initializes the timer handle *tim* for use. No operations can be performed on a timer before it is initialized.

Parameters

tim	The timer to initialize.

3.50.5.3 int rte_timer_reset (struct rte_timer * tim, uint64_t ticks, enum rte_timer_type type, unsigned tim_lcore, rte_timer_cb_t fct, void * arg)

Reset and start the timer associated with the timer handle.

The rte_timer_reset() function resets and starts the timer associated with the timer handle *tim*. When the timer expires after *ticks* HPET cycles, the function specified by *fct* will be called with the argument *arg* on core *tim_lcore*.

If the timer associated with the timer handle is already running (in the RUNNING state), the function will fail. The user has to check the return value of the function to see if there is a chance that the timer is in the RUNNING state.

If the timer is being configured on another core (the CONFIG state), it will also fail.

If the timer is pending or stopped, it will be rescheduled with the new parameters.

Parameters

tim	The timer handle.
ticks	The number of cycles (see rte_get_hpet_hz()) before the callback function is called.
type	The type can be either: • PERIODICAL: The timer is automatically reloaded after execution (returns to the PENDING state) • SINGLE: The timer is one-shot, that is, the timer goes to a STOPPED state after execution.
tim_lcore	The ID of the lcore where the timer callback function has to be executed. If tim_lcore is LCORE_ID_ANY, the timer library will launch it on a different core for each call (round-robin).
fct	The callback function of the timer.
arg	The user argument of the callback function.

Returns

- 0: Success; the timer is scheduled.
- (-1): Timer is in the RUNNING or CONFIG state.

3.50.5.4 void rte_timer_reset_sync (struct rte_timer * *tim*, uint64_t *ticks*, enum rte_timer_type *type*, unsigned *tim_lcore*, rte_timer_cb_t *fct*, void * *arg*)

Loop until rte_timer_reset() succeeds.

Reset and start the timer associated with the timer handle. Always succeed. See rte_timer_reset() for details.

Parameters

tim	The timer handle.
ticks	The number of cycles (see rte_get_hpet_hz()) before the callback function is called.
type	The type can be either: • PERIODICAL: The timer is automatically reloaded after execution (returns to the PENDING state) • SINGLE: The timer is one-shot, that is, the timer goes to a STOPPED state after execution.
tim_lcore	The ID of the lcore where the timer callback function has to be executed. If tim_lcore is LCORE_ID_ANY, the timer library will launch it on a different core for each call (round-robin).
fct	The callback function of the timer.
arg	The user argument of the callback function.

3.50.5.5 int rte_timer_stop (struct rte_timer * *tim*)

Stop a timer.

The rte_timer_stop() function stops the timer associated with the timer handle *tim*. It may fail if the timer is currently running or being configured.

If the timer is pending or stopped (for instance, already expired), the function will succeed. The timer handle tim must have been initialized using rte_timer_init(), otherwise, undefined behavior will occur.

This function can be called safely from a timer callback. If it succeeds, the timer is not referenced anymore by the timer library and the timer structure can be freed (even in the callback function).

Parameters

tim	The timer handle.

Returns

- 0: Success; the timer is stopped.
- (-1): The timer is in the RUNNING or CONFIG state.

3.50.5.6 void rte_timer_stop_sync (struct rte_timer * *tim*)

Loop until rte_timer_stop() succeeds.

After a call to this function, the timer identified by *tim* is stopped. See rte_timer_stop() for details.

Parameters

tim	The timer handle.

3.50.5.7 int rte_timer_pending (struct rte_timer * *tim*)

Test if a timer is pending.

The rte_timer_pending() function tests the PENDING status of the timer handle *tim*. A PENDING timer is one that has been scheduled and whose function has not yet been called.

Parameters

tim	The timer handle.

Returns

- 0: The timer is not pending.
- 1: The timer is pending.

3.50.5.8 void rte_timer_manage (void)

Manage the timer list and execute callback functions.

This function must be called periodically from all cores main_loop(). It browses the list of pending timers and runs all timers that are expired.

The precision of the timer depends on the call frequency of this function. However, the more often the function is called, the more CPU resources it will use.

3.50.5.9 void rte_timer_dump_stats (void)

Dump statistics about timers.

3.51 rte_udp.h File Reference

Data Structures

- struct udp_hdr

3.51.1 Detailed Description

UDP-related defines

3.52 rte_version.h File Reference

Defines

- #define RTE_VER_PREFIX
- #define RTE_VER_MAJOR
- #define RTE_VER_MINOR
- #define RTE_VER_PATCH_LEVEL
- #define RTE_VER_SUFFIX

Functions

- static const char * rte_version (void)

3.52.1 Detailed Description

Definitions of Intel(R) DPDK version numbers

3.52.2 Define Documentation

3.52.2.1 #define RTE_VER_PREFIX

String that appears before the version number

3.52.2.2 #define RTE_VER_MAJOR

Major version number i.e. the x in x.y.z

3.52.2.3 #define RTE_VER_MINOR

Minor version number i.e. the y in x.y.z

3.52.2.4 #define RTE_VER_PATCH_LEVEL

Patch level number i.e. the z in x.y.z

3.52.2.5 #define RTE_VER_SUFFIX

Extra string to be appended to version number, for example: pre1, EAR, final etc.

3.52.3 Function Documentation

3.52.3.1 static const char* rte_version (void) [static]

Function returning string of version number: "RTE x.y.z"

Returns

string

3.53 rte_warnings.h File Reference

3.53.1 Detailed Description

Definitions of warnings for use of various insecure functions