rte_eventdev.h File Reference

```
#include <rte_pci.h>
#include <rte_dev.h>
#include <rte_devargs.h>
#include <rte_errno.h>
```

Go to the source code of this file.

Data Structures

struct	**rte_event_dev_info**
struct	**rte_event_dev_config**
struct	**rte_event_queue_conf**
struct	**rte_event_port_conf**
struct	**rte_event**
struct	**rte_event_queue_link**

Macros

#define	**RTE_EVENT_DEV_CAP_QUEUE_QOS**	(1 << 0)
#define	**RTE_EVENT_DEV_CAP_EVENT_QOS**	(1 << 1)
#define	**RTE_EVENT_DEV_CFG_PER_DEQUEUE_WAIT**	(1 << 0)
#define	**RTE_EVENT_QUEUE_PRIORITY_HIGHEST**	0
#define	**RTE_EVENT_QUEUE_PRIORITY_NORMAL**	128
#define	**RTE_EVENT_QUEUE_PRIORITY_LOWEST**	255
#define	**RTE_EVENT_QUEUE_CFG_SINGLE_CONSUMER**	(1 << 0)
#define	**RTE_SCHED_TYPE_ORDERED**	0
#define	**RTE_SCHED_TYPE_ATOMIC**	1
#define	**RTE_SCHED_TYPE_PARALLEL**	2
#define	**RTE_EVENT_TYPE_ETHDEV**	0x0
#define	**RTE_EVENT_TYPE_CRYPTODEV**	0x1
#define	**RTE_EVENT_TYPE_TIMERDEV**	0x2
#define	**RTE_EVENT_TYPE_CORE**	0x3
#define	**RTE_EVENT_TYPE_MAX**	0x10
#define	**RTE_EVENT_PRIORITY_HIGHEST**	0
#define	**RTE_EVENT_PRIORITY_NORMAL**	128
#define	**RTE_EVENT_PRIORITY_LOWEST**	255
#define	**RTE_EVENT_QUEUE_SERVICE_PRIORITY_HIGHEST**	0
#define	**RTE_EVENT_QUEUE_SERVICE_PRIORITY_NORMAL**	128

#define **RTE_EVENT_QUEUE_SERVICE_PRIORITY_LOWEST** 255

Functions

uint8_t **rte_event_dev_count** (void)

uint8_t **rte_event_dev_get_dev_id** (const char *name)

int **rte_event_dev_socket_id** (uint8_t **dev_id**)

void **rte_event_dev_info_get** (uint8_t **dev_id**, struct **rte_event_dev_info** *dev_info)

int **rte_event_dev_configure** (uint8_t **dev_id**, struct **rte_event_dev_config** *config)

void **rte_event_queue_default_conf_get** (uint8_t **dev_id**, uint8_t queue_id, struct **rte_event_queue_conf** *queue_conf)

int **rte_event_queue_setup** (uint8_t **dev_id**, uint8_t queue_id, struct **rte_event_queue_conf** *queue_conf)

uint16_t **rte_event_queue_count** (uint8_t **dev_id**)

uint8_t **rte_event_queue_priority** (uint8_t **dev_id**, uint8_t queue_id)

void **rte_event_port_default_conf_get** (uint8_t **dev_id**, uint8_t port_id, struct **rte_event_port_conf** *port_conf)

int **rte_event_port_setup** (uint8_t **dev_id**, uint8_t port_id, struct **rte_event_port_conf** *port_conf)

uint8_t **rte_event_port_dequeue_depth** (uint8_t **dev_id**, uint8_t port_id)

uint8_t **rte_event_port_enqueue_depth** (uint8_t **dev_id**, uint8_t port_id)

uint8_t **rte_event_port_count** (uint8_t **dev_id**)

int **rte_event_dev_start** (uint8_t **dev_id**)

void **rte_event_dev_stop** (uint8_t **dev_id**)

int **rte_event_dev_close** (uint8_t **dev_id**)

void **rte_event_schedule** (uint8_t **dev_id**)

int **rte_event_enqueue** (uint8_t **dev_id**, uint8_t port_id, struct **rte_event** *ev, bool pin_event)

int **rte_event_enqueue_burst** (uint8_t **dev_id**, uint8_t port_id, struct **rte_event** ev[], int num, bool pin_event)

uint64_t **rte_event_dequeue_wait_time** (uint8_t **dev_id**, uint64_t ns)

bool **rte_event_dequeue** (uint8_t **dev_id**, uint8_t port_id, struct **rte_event** *ev, uint64_t wait)

int **rte_event_dequeue_burst** (uint8_t **dev_id**, uint8_t port_id, struct **rte_event** *ev, int num, uint64_t wait)

void **rte_event_release** (uint8_t **dev_id**, uint8_t port_id, uint8_t index)

int **rte_event_port_link** (uint8_t **dev_id**, uint8_t port_id, struct **rte_event_queue_link** link[], int num)

int **rte_event_port_unlink** (uint8_t **dev_id**, uint8_t port_id, uint8_t queues[], int num)

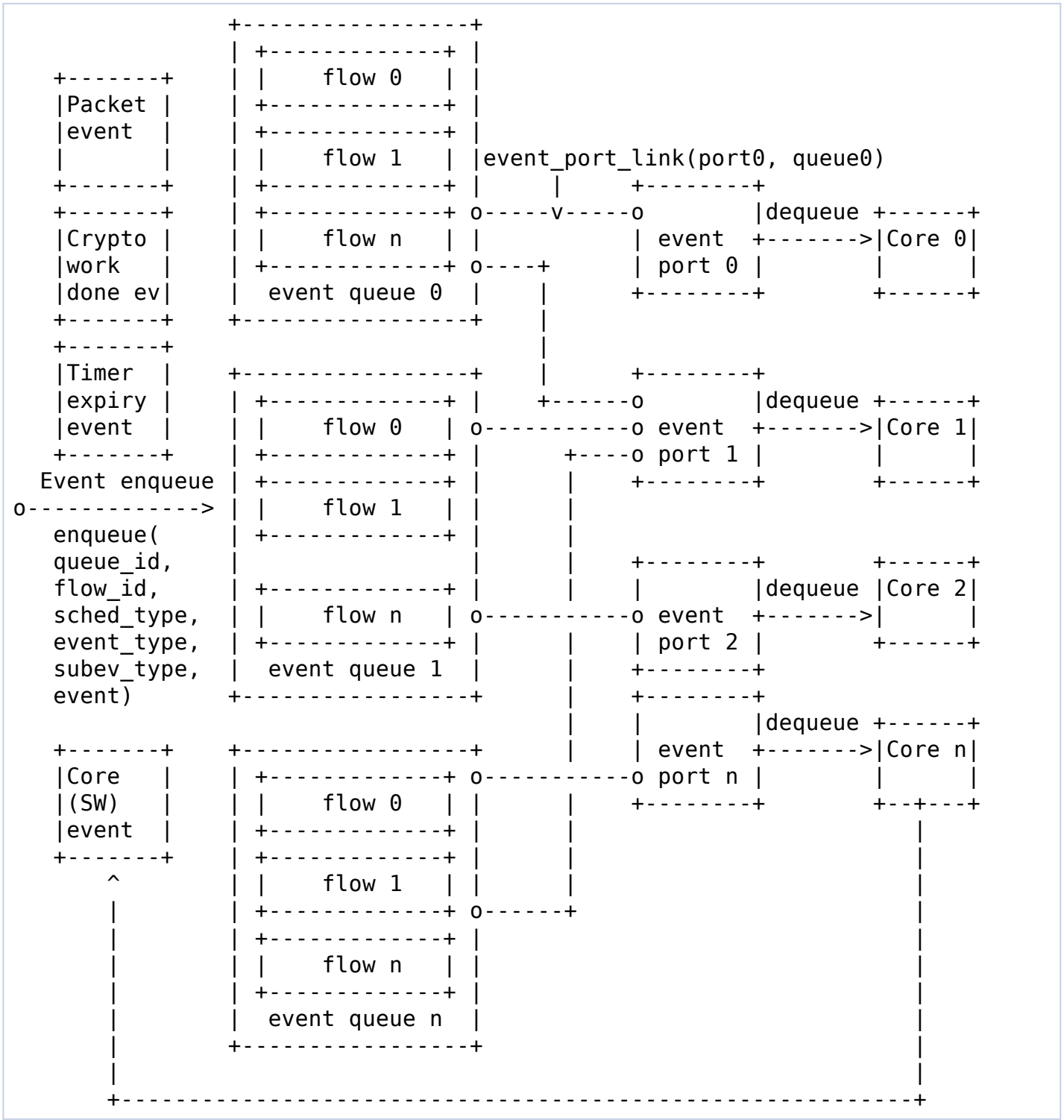
Detailed Description

RTE Event Device API

The Event Device API is composed of two parts:

- The application-oriented Event API that includes functions to setup an event device (configure it, setup its queues, ports and start it), to establish the link between queues to port and to receive events, and so on.
- The driver-oriented Event API that exports a function allowing an event poll Mode Driver (PMD) to simultaneously register itself as an event device driver.

Event device components:



Event device: A hardware or software-based event scheduler.

Event: A unit of scheduling that encapsulates a packet or other datatype like SW generated event from the core, Crypto work completion notification, Timer expiry event notification etc as well as metadata. The metadata includes flow ID, scheduling type, event priority, event_type, sub_event_type etc.

Event queue: A queue containing events that are scheduled by the event dev. An event queue contains events of different flows associated with scheduling types, such as atomic, ordered, or parallel.

Event port: An application's interface into the event dev for enqueue and dequeue operations. Each event port can be linked with one or more event queues for dequeue operations.

By default, all the functions of the Event 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 dequeue function of a PMD cannot be invoked in parallel on two logical cores to operates on same event port. Of course, this function can be invoked in parallel by different logical cores on different ports. It is the responsibility of the upper level application to enforce this rule.

In all functions of the Event API, the Event device is designated by an integer >= 0 named the device identifier *dev_id*

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

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

The role of the device init function consists of resetting the hardware or software event driver implementations.

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

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

- **rte_event_dev_configure()**
- **rte_event_queue_setup()**
- **rte_event_port_setup()**
- **rte_event_port_link()**
- **rte_event_dev_start()**

Then, the application can invoke, in any order, the functions exported by the Event API to schedule events, dequeue events, enqueue events, change event queue(s) to event port [un]link establishment and so on.

Application may use rte_event_[queue/port]_default_conf_get() to get the default configuration to set up an event queue or event port by overriding few default values.

If the application wants to change the configuration (i.e. call **rte_event_dev_configure()**, **rte_event_queue_setup()**, or **rte_event_port_setup()**), it must call **rte_event_dev_stop()** first to stop the

device and then do the reconfiguration before calling **rte_event_dev_start()** again. The schedule, enqueue and dequeue functions should not be invoked when the device is stopped.

Finally, an application can close an Event device by invoking the **rte_event_dev_close()** function.

Each function of the application Event API invokes a specific function of the PMD that controls the target device designated by its device identifier.

For this purpose, all device-specific functions of an Event driver are supplied through a set of pointers contained in a generic structure of type *event_dev_ops*. The address of the *event_dev_ops* structure is stored in the *rte_event_dev* structure by the device init function of the Event driver, which is invoked during the PCI/SoC device probing phase, as explained earlier.

In other words, each function of the Event API simply retrieves the *rte_event_dev* structure associated with the device identifier and performs an indirect invocation of the corresponding driver function supplied in the *event_dev_ops* structure of the *rte_event_dev* structure.

For performance reasons, the address of the fast-path functions of the Event driver is not contained in the *event_dev_ops* structure. Instead, they are directly stored at the beginning of the *rte_event_dev* structure to avoid an extra indirect memory access during their invocation.

RTE event device drivers do not use interrupts for enqueue or dequeue operation. Instead, Event drivers export Poll-Mode enqueue and dequeue functions to applications.

An event driven based application has following typical workflow on fastpath:

```
while (1) {
   rte_event_schedule(dev_id);
   rte_event_dequeue(...);
   (event processing)
   rte_event_enqueue(...);
}
```

The *schedule* operation is intended to do event scheduling, and the *dequeue* operation returns the scheduled events. An implementation is free to define the semantics between *schedule* and *dequeue*. For example, a system based on a hardware scheduler can define its **rte_event_schedule()** to be an NOOP, whereas a software scheduler can use the *schedule* operation to schedule events.

The events are injected to event device through *enqueue* operation by event producers in the system. The typical event producers are ethdev subsystem for generating packet events, core(SW) for generating events based on different stages of application processing, cryptodev for generating crypto work completion notification etc

The *dequeue* operation gets one or more events from the event ports. The application process the events and send to downstream event queue through **rte_event_enqueue()** if it is an intermediate stage of event processing, on the final stage, the application may send to different subsystem like ethdev to send the packet/event on the wire using ethdev **rte_eth_tx_burst()** API.

Definition in file **rte_eventdev.h**.

Macro Definition Documentation

#define RTE_EVENT_DEV_CAP_QUEUE_QOS (1 << 0)

Event scheduling prioritization is based on the priority associated with each event queue.

See also

rte_event_queue_setup(), RTE_EVENT_QUEUE_PRIORITY_NORMAL

Definition at line 277 of file rte_eventdev.h.

#define RTE_EVENT_DEV_CAP_EVENT_QOS (1 << 1)

Event scheduling prioritization is based on the priority associated with each event. Priority of each event is supplied in *rte_event* structure on each enqueue operation.

See also

rte_event_enqueue()

Definition at line 283 of file rte_eventdev.h.

#define RTE_EVENT_DEV_CFG_PER_DEQUEUE_WAIT (1 << 0)

Override the global *dequeue_wait_ns* and use per dequeue wait in ns.

See also

rte_event_dequeue_wait_time(), rte_event_dequeue()

Definition at line 356 of file rte_eventdev.h.

#define RTE_EVENT_QUEUE_PRIORITY_HIGHEST 0

Highest event queue priority

Definition at line 416 of file rte_eventdev.h.

#define RTE_EVENT_QUEUE_PRIORITY_NORMAL 128

Normal event queue priority

Definition at line 418 of file rte_eventdev.h.

#define RTE_EVENT_QUEUE_PRIORITY_LOWEST 255

Lowest event queue priority

Definition at line **420** of file **rte_eventdev.h**.

#define RTE_EVENT_QUEUE_CFG_SINGLE_CONSUMER (1 << 0)

This event queue links only to a single event port.

See also

rte_event_port_setup(), **rte_event_port_link()**

Definition at line **424** of file **rte_eventdev.h**.

#define RTE_SCHED_TYPE_ORDERED 0

Ordered scheduling

Events from an ordered flow of an event queue can be scheduled to multiple ports for concurrent processing while maintaining the original event order. This scheme enables the user to achieve high single flow throughput by avoiding SW synchronization for ordering between ports which bound to cores.

The source flow ordering from an event queue is maintained when events are enqueued to their destination queue within the same ordered flow context. An event port holds the context until application call **rte_event_dequeue()** from the same port, which implicitly releases the context. User may allow the scheduler to release the context earlier than that by calling **rte_event_release()**

Events from the source queue appear in their original order when dequeued from a destination queue. Event ordering is based on the received event(s), but also other (newly allocated or stored) events are ordered when enqueued within the same ordered context. Events not enqueued (e.g. released or stored) within the context are considered missing from reordering and are skipped at this time (but can be ordered again within another context).

See also

rte_event_dequeue(), **rte_event_release()**

Definition at line **686** of file **rte_eventdev.h**.

#define RTE_SCHED_TYPE_ATOMIC 1

Atomic scheduling

Events from an atomic flow of an event queue can be scheduled only to a single port at a time. The port is guaranteed to have exclusive (atomic) access to the associated flow context, which enables the user to avoid SW synchronization. Atomic flows also help to maintain event ordering since only one port at a time can process events from a flow of an event queue.

The atomic queue synchronization context is dedicated to the port until application call **rte_event_dequeue()** from the same port, which implicitly releases the context. User may allow the scheduler to release the context earlier than that by calling **rte_event_release()**

See also

rte_event_dequeue(), **rte_event_release()**

Definition at line **712** of file **rte_eventdev.h**.

#define RTE_SCHED_TYPE_PARALLEL 2

Parallel scheduling

The scheduler performs priority scheduling, load balancing, etc. functions but does not provide additional event synchronization or ordering. It is free to schedule events from a single parallel flow of an event queue to multiple events ports for concurrent processing. The application is responsible for flow context synchronization and event ordering (SW synchronization).

Definition at line **730** of file **rte_eventdev.h**.

#define RTE_EVENT_TYPE_ETHDEV 0x0

The event generated from ethdev subsystem

Definition at line **742** of file **rte_eventdev.h**.

#define RTE_EVENT_TYPE_CRYPTODEV 0x1

The event generated from crypodev subsystem

Definition at line **744** of file **rte_eventdev.h**.

#define RTE_EVENT_TYPE_TIMERDEV 0x2

The event generated from timerdev subsystem

Definition at line **746** of file **rte_eventdev.h**.

#define RTE_EVENT_TYPE_CORE 0x3

The event generated from core. Application may use *sub_event_type* to further classify the event

Definition at line **748** of file **rte_eventdev.h**.

#define RTE_EVENT_TYPE_MAX 0x10

Maximum number of event types

Definition at line **752** of file **rte_eventdev.h**.

#define RTE_EVENT_PRIORITY_HIGHEST 0

Highest event priority

Definition at line **756** of file **rte_eventdev.h**.

#define RTE_EVENT_PRIORITY_NORMAL 128

Normal event priority

Definition at line **758** of file **rte_eventdev.h**.

#define RTE_EVENT_PRIORITY_LOWEST 255

Lowest event priority

Definition at line **760** of file **rte_eventdev.h**.

#define RTE_EVENT_QUEUE_SERVICE_PRIORITY_HIGHEST 0

Highest event queue servicing priority

Definition at line 1083 of file rte_eventdev.h.

#define RTE_EVENT_QUEUE_SERVICE_PRIORITY_NORMAL 128

Normal event queue servicing priority

Definition at line 1085 of file rte_eventdev.h.

#define RTE_EVENT_QUEUE_SERVICE_PRIORITY_LOWEST 255

Lowest event queue servicing priority

Definition at line 1087 of file rte_eventdev.h.

Function Documentation

uint8_t rte_event_dev_count (void)

Get the total number of event devices that have been successfully initialised.

Returns

The total number of usable event devices.

uint8_t rte_event_dev_get_dev_id (const char * name)

Get the device identifier for the named event device.

Parameters

name Event device name to select the event device identifier.

Returns

Returns event device identifier on success.

- <0: Failure to find named event device.

int rte_event_dev_socket_id (uint8_t dev_id)

Return the NUMA socket to which a device is connected.

Parameters

dev_id	The identifier of the device.

Returns

The NUMA socket id to which the device is connected or a default of zero if the socket could not be determined.

- -1: dev_id value is out of range.

void rte_event_dev_info_get (uint8_t dev_id,
struct rte_event_dev_info * dev_info
)

Retrieve the contextual information of an event device.

Parameters

	dev_id	The identifier of the device.
[out]	**dev_info**	A pointer to a structure of type *rte_event_dev_info* to be filled with the contextual information of the device.

int rte_event_dev_configure (uint8_t dev_id,
struct rte_event_dev_config * config
)

Configure an event device.

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

The caller may use **rte_event_dev_info_get()** to get the capability of each resources available for this event device.

Parameters

dev_id	The identifier of the device to configure.
config	The event device configuration structure.

Returns

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

void rte_event_queue_default_conf_get (uint8_t dev_id,
uint8_t queue_id,
struct rte_event_queue_conf * queue_conf
)

Retrieve the default configuration information of an event queue designated by its *queue_id* from the event driver for an event device.

This function intended to be used in conjunction with **rte_event_queue_setup()** where caller needs to set up the queue by overriding few default values.

Parameters

	dev_id	The identifier of the device.
	queue_id	The index of the event queue to get the configuration information. The value must be in the range [0, nb_event_queues - 1] previously supplied to **rte_event_dev_configure()**.
[out]	**queue_conf**	The pointer to the default event queue configuration data.

See also

rte_event_queue_setup()

int rte_event_queue_setup (uint8_t dev_id,
uint8_t queue_id,
struct rte_event_queue_conf * queue_conf
)

Allocate and set up an event queue for an event device.

Parameters

dev_id	The identifier of the device.
queue_id	The index of the event queue to setup. The value must be in the range [0, nb_event_queues - 1] previously supplied to **rte_event_dev_configure()**.
queue_conf	The pointer to the configuration data to be used for the event queue. NULL value is allowed, in which case default configuration used.

See also

rte_event_queue_default_conf_get()

Returns

- 0: Success, event queue correctly set up.
- <0: event queue configuration failed

uint16_t rte_event_queue_count (uint8_t dev_id)

Get the number of event queues on a specific event device

Parameters

dev_id	Event device identifier.

Returns

- The number of configured event queues

```
uint8_t rte_event_queue_priority ( uint8_t dev_id,
                                   uint8_t queue_id
                                 )
```

Get the priority of the event queue on a specific event device

Parameters

dev_id	Event device identifier.
queue_id	Event queue identifier.

Returns

- If the device has RTE_EVENT_DEV_CAP_QUEUE_QOS capability then the configured priority of the event queue in [RTE_EVENT_QUEUE_PRIORITY_HIGHEST, RTE_EVENT_QUEUE_PRIORITY_LOWEST] range else the value one

```
void rte_event_port_default_conf_get ( uint8_t                       dev_id,
                                       uint8_t                       port_id,
                                       struct rte_event_port_conf *  port_conf
                                     )
```

Retrieve the default configuration information of an event port designated by its *port_id* from the event driver for an event device.

This function intended to be used in conjunction with **rte_event_port_setup()** where caller needs to set up the port by overriding few default values.

Parameters

	dev_id	The identifier of the device.
	port_id	The index of the event port to get the configuration information. The value must be in the range [0, nb_event_ports - 1] previously supplied to **rte_event_dev_configure()**.
[out]	**port_conf**	The pointer to the default event port configuration data

See also

rte_event_port_setup()

```
int rte_event_port_setup ( uint8_t                          dev_id,
                           uint8_t                          port_id,
                           struct rte_event_port_conf *     port_conf
                         )
```

Allocate and set up an event port for an event device.

Parameters

dev_id	The identifier of the device.
port_id	The index of the event port to setup. The value must be in the range [0, nb_event_ports - 1] previously supplied to **rte_event_dev_configure()**.
port_conf	The pointer to the configuration data to be used for the queue. NULL value is allowed, in which case default configuration used.

See also

rte_event_port_default_conf_get()

Returns

- 0: Success, event port correctly set up.
- <0: Port configuration failed
- (-EDQUOT) Quota exceeded(Application tried to link the queue configured with RTE_EVENT_QUEUE_CFG_SINGLE_CONSUMER to more than one event ports)

```
uint8_t rte_event_port_dequeue_depth ( uint8_t  dev_id,
                                       uint8_t  port_id
                                     )
```

Get the number of dequeue queue depth configured for event port designated by its *port_id* on a specific event device

Parameters

dev_id	Event device identifier.
port_id	Event port identifier.

Returns

- The number of configured dequeue queue depth

See also

rte_event_dequeue_burst()

uint8_t rte_event_port_enqueue_depth (uint8_t dev_id,
uint8_t port_id
)

Get the number of enqueue queue depth configured for event port designated by its *port_id* on a specific event device

Parameters

dev_id Event device identifier.

port_id Event port identifier.

Returns

- The number of configured enqueue queue depth

See also

rte_event_enqueue_burst()

uint8_t rte_event_port_count (uint8_t dev_id)

Get the number of ports on a specific event device

Parameters

dev_id Event device identifier.

Returns

- The number of configured ports

int rte_event_dev_start (uint8_t dev_id)

Start an event device.

The device start step is the last one and consists of setting the event queues to start accepting the events and schedules to event ports.

On success, all basic functions exported by the API (event enqueue, event dequeue and so on) can be invoked.

Parameters

dev_id Event device identifier

Returns

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

void rte_event_dev_stop (uint8_t dev_id)

Stop an event device. The device can be restarted with a call to **rte_event_dev_start()**

Parameters

dev_id Event device identifier.

int rte_event_dev_close (uint8_t dev_id)

Close an event device. The device cannot be restarted!

Parameters

dev_id Event device identifier

Returns

- 0 on successfully closing device
- <0 on failure to close device

void rte_event_schedule (uint8_t dev_id)

Schedule one or more events in the event dev.

An event dev implementation may define this is a NOOP, for instance if the event dev performs its scheduling in hardware.

Parameters

dev_id	The identifier of the device.

int rte_event_enqueue (uint8_t dev_id, uint8_t port_id, struct rte_event * ev, bool pin_event)

Enqueue the event object supplied in the ***rte_event*** structure on an event device designated by its *dev_id* through the event port specified by *port_id*. The event object specifies the event queue on which this event will be enqueued.

Parameters

dev_id	Event device identifier.
port_id	The identifier of the event port.
ev	Pointer to struct **rte_event**
pin_event	Hint to the scheduler that the event can be pinned to the same port for the next scheduling stage. For implementations that support it, this allows the same core to process the next stage in the pipeline for a given event, taking advantage of cache locality. The pinned event will be received through **rte_event_dequeue()**. This is a hint and the event is not guaranteed to be pinned to the port. This hint is valid only when the event is dequeued with **rte_event_dequeue()** followed by **rte_event_enqueue()**.

Returns

- 0 on success
- <0 on failure. Failure can occur if the event port's output queue is backpressured, for instance.

```
int rte_event_enqueue_burst ( uint8_t           dev_id,
                              uint8_t           port_id,
                              struct rte_event  ev[],
                              int               num,
                              bool              pin_event
                            )
```

Enqueue a burst of events objects supplied in ***rte_event*** structure on an event device designated by its *dev_id* through the event port specified by *port_id*. Each event object specifies the event queue on which it will be enqueued.

The **rte_event_enqueue_burst()** function is invoked to enqueue multiple event objects. It is the burst variant of **rte_event_enqueue()** function.

The *num* parameter is the number of event objects to enqueue which are supplied in the *ev* array of ***rte_event*** structure.

The **rte_event_enqueue_burst()** function returns the number of events objects it actually enqueued. A return value equal to *num* means that all event objects have been enqueued.

Parameters

dev_id	The identifier of the device.
port_id	The identifier of the event port.
ev	An array of *num* pointers to ***rte_event*** structure which contain the event object enqueue operations to be processed.
num	The number of event objects to enqueue, typically number of **rte_event_port_enqueue_depth()** available for this port.
pin_event	Hint to the scheduler that the event can be pinned to the same port for the next scheduling stage. For implementations that support it, this allows the same core to process the next stage in the pipeline for a given event, taking advantage of cache locality. The pinned event will be received through **rte_event_dequeue()**. This is a hint and the event is not guaranteed to be pinned to the port. This hint is valid only when the event is dequeued with **rte_event_dequeue()** followed by **rte_event_enqueue()**.

Returns

The number of event objects actually enqueued on the event device. The return value can be less than the value of the *num* parameter when the event devices queue is full or if invalid parameters are specified in a ***rte_event***. If return value is less than *num*, the remaining events at the end of ev[] are not consumed, and the caller has to take care of them.

See also

rte_event_enqueue(), **rte_event_port_enqueue_depth()**

uint64_t rte_event_dequeue_wait_time (uint8_t dev_id,
uint64_t ns
)

Converts nanoseconds to *wait* value for **rte_event_dequeue()**

If the device is configured with RTE_EVENT_DEV_CFG_PER_DEQUEUE_WAIT flag then application can use this function to convert wait value in nanoseconds to implementations specific wait value supplied in **rte_event_dequeue()**

Parameters

dev_id	The identifier of the device.
ns	Wait time in nanosecond

Returns

Value for the *wait* parameter in **rte_event_dequeue()** function

See also

rte_event_dequeue(), **RTE_EVENT_DEV_CFG_PER_DEQUEUE_WAIT**

rte_event_dev_configure()

```
bool rte_event_dequeue ( uint8_t              dev_id,
                         uint8_t              port_id,
                         struct rte_event *   ev,
                         uint64_t             wait
                       )
```

Dequeue an event from the event port specified by *port_id* on the event device designated by its *dev_id*.

rte_event_dequeue() does not dictate the specifics of scheduling algorithm as each eventdev driver may have different criteria to schedule an event. However, in general, from an application perspective scheduler may use the following scheme to dispatch an event to the port.

1) Selection of event queue based on a) The list of event queues are linked to the event port. b) If the device has RTE_EVENT_DEV_CAP_QUEUE_QOS capability then event queue selection from list is based on event queue priority relative to other event queue supplied as *priority* in **rte_event_queue_setup()** c) If the device has RTE_EVENT_DEV_CAP_EVENT_QOS capability then event queue selection from the list is based on event priority supplied as *priority* in **rte_event_enqueue_burst()** 2) Selection of event a) The number of flows available in selected event queue. b) Schedule type method associated with the event

On a successful dequeue, the event port holds flow id and schedule type context associated with the dispatched event. The context is automatically released in the next **rte_event_dequeue()** invocation, or **rte_event_release()** can be called to release the context early.

Parameters

	dev_id	The identifier of the device.
	port_id	The identifier of the event port.
[out]	**ev**	Pointer to struct **rte_event**. On successful event dispatch, implementation updates the event attributes.
	wait	0 - no-wait, returns immediately if there is no event. >0 - wait for the event, if the device is configured with RTE_EVENT_DEV_CFG_PER_DEQUEUE_WAIT then this function will wait until the event available or *wait* time. if the device is not configured with RTE_EVENT_DEV_CFG_PER_DEQUEUE_WAIT then this function will wait until the event available or *dequeue_wait_ns* ns which was previously supplied to **rte_event_dev_configure()**

Returns

When true, a valid event has been dispatched by the scheduler.

```
int rte_event_dequeue_burst ( uint8_t              dev_id,
                              uint8_t              port_id,
                              struct rte_event *   ev,
                              int                  num,
                              uint64_t             wait
                            )
```

Dequeue a burst of events objects from the event port designated by its *event_port_id*, on an event device designated by its *dev_id*.

The **rte_event_dequeue_burst()** function is invoked to dequeue multiple event objects. It is the burst variant of **rte_event_dequeue()** function.

The *num* parameter is the maximum number of event objects to dequeue which are returned in the *ev* array of ***rte_event*** structure.

The **rte_event_dequeue_burst()** function returns the number of events objects it actually dequeued. A return value equal to *num* means that all event objects have been dequeued.

The number of events dequeued is the number of scheduler contexts held by this port. These contexts are automatically released in the next **rte_event_dequeue()** invocation, or **rte_event_release()** can be called once per event to release the contexts early.

Parameters

	dev_id	The identifier of the device.
	port_id	The identifier of the event port.
[out]	**ev**	An array of *num* pointers to ***rte_event*** structure which is populated with the dequeued event objects.
	num	The maximum number of event objects to dequeue, typically number of **rte_event_port_dequeue_depth()** available for this port.
	wait	0 - no-wait, returns immediately if there is no event. >0 - wait for the event, if the device is configured with RTE_EVENT_DEV_CFG_PER_DEQUEUE_WAIT then this function will wait until the event available or *wait* time. if the device is not configured with RTE_EVENT_DEV_CFG_PER_DEQUEUE_WAIT then this function will wait until the event available or *dequeue_wait_ns* ns which was previously supplied to **rte_event_dev_configure()**

Returns

The number of event objects actually dequeued from the port. The return value can be less than the value of the *num* parameter when the event port's queue is not full.

See also

rte_event_dequeue(), **rte_event_port_dequeue_depth()**

```
void rte_event_release ( uint8_t dev_id,
                         uint8_t port_id,
                         uint8_t index
                       )
```

Release the current flow context associated with a schedule type which dequeued from a given event queue though the event port designated by its *port_id*

If current flow's scheduler type method is *RTE_SCHED_TYPE_ATOMIC* then this function hints the scheduler that the user has completed critical section processing in the current atomic context. The scheduler is now allowed to schedule events from the same flow from an event queue to another port. However, the context may be still held until the next **rte_event_dequeue()** or **rte_event_dequeue_burst()** call, this call allows but does not force the scheduler to release the context early.

Early atomic context release may increase parallelism and thus system performance, but the user needs to design carefully the split into critical vs non-critical sections.

If current flow's scheduler type method is *RTE_SCHED_TYPE_ORDERED* then this function hints the scheduler that the user has done all that need to maintain event order in the current ordered context. The scheduler is allowed to release the ordered context of this port and avoid reordering any following enqueues.

Early ordered context release may increase parallelism and thus system performance.

If current flow's scheduler type method is *RTE_SCHED_TYPE_PARALLEL* or no scheduling context is held then this function may be an NOOP, depending on the implementation.

If multiple events are dequeued with **rte_event_dequeue_burst()**, **rte_event_release()** will release each flow context associated with a schedule type of an event though *index*, it denotes the order in which it was dequeued with **rte_event_dequeue_burst()**

Parameters

dev_id	The identifier of the device.
port_id	The identifier of the event port.
index	The index of the event that dequeued with **rte_event_dequeue_burst()** which needs to release. The value zero used if the event dequeued with **rte_event_dequeue()**

See also

rte_event_dequeue(), **rte_event_dequeue_burst()**

```
int rte_event_port_link ( uint8_t                    dev_id,
                          uint8_t                    port_id,
                          struct rte_event_queue_link link[],
                          int                        num
                        )
```

Link multiple source event queues supplied in ***rte_event_queue_link*** structure as *queue_id* to the destination event port designated by its *port_id* on the event device designated by its *dev_id*.

The link establishment shall enable the event port *port_id* from receiving events from the specified event queue *queue_id*

An event queue may link to one or more event ports. The number of links can be established from an event queue to event port is implementation defined.

Event queue(s) to event port link establishment can be changed at runtime without re-configuring the device to support scaling and to reduce the latency of critical work by establishing the link with more event ports at runtime.

Parameters

dev_id	The identifier of the device.
port_id	Event port identifier to select the destination port to link.
link	An array of *num* pointers to ***rte_event_queue_link*** structure which contain the event queue to event port link establishment attributes. NULL value is allowed, in which case this function links all the configured event queues *nb_event_queues* which previously supplied to **rte_event_dev_configure()** to the event port *port_id* with normal servicing priority(RTE_EVENT_QUEUE_SERVICE_PRIORITY_NORMAL).
num	The number of links to establish

Returns

The number of links actually established on the event device. The return value can be less than the value of the *num* parameter when the implementation has the limitation on specific queue to port link establishment or if invalid parameters are specified in a ***rte_event_queue_link***. If the return value is less than *num*, the remaining links at the end of link[] are not established, and the caller has to take care of them. If return value is less than *num* then implementation shall update the rte_errno accordingly, Possible rte_errno values are (-EDQUOT) Quota exceeded(Application tried to link the queue configured with RTE_EVENT_QUEUE_CFG_SINGLE_CONSUMER to more than one event ports) (-EINVAL) Invalid parameter

```
int rte_event_port_unlink ( uint8_t dev_id,
                            uint8_t port_id,
                            uint8_t queues[],
                            int     num
                          )
```

Unlink multiple source event queues supplied in *queues* from the destination event port designated by its *port_id* on the event device designated by its *dev_id*.

The unlink establishment shall disable the event port *port_id* from receiving events from the specified event queue *queue_id*

Event queue(s) to event port unlink establishment can be changed at runtime without re-configuring the device.

Parameters

dev_id	The identifier of the device.
port_id	Event port identifier to select the destination port to unlink.
queues	An array of *num* event queues to be unlinked from the event port. NULL value is allowed, in which case this function unlinks all the event queue(s) from the event port *port_id*.
num	The number of unlinks to establish

Returns

The number of unlinks actually established on the event device. The return value can be less than the value of the *num* parameter when the implementation has the limitation on specific queue to port unlink establishment or if invalid parameters are specified. If the return value is less than *num*, the remaining queues at the end of queues[] are not established, and the caller has to take care of them. If return value is less than *num* then implementation shall update the rte_errno accordingly, Possible rte_errno values are (-EINVAL) Invalid parameter

Generated by doxygen 1.8.11