Open Event Machine Library

API Spec

Applies to Product Release: 01.10.00.00:
Publication Date: February, 2014

Contributors to this document

Texas Instruments, Incorporated
821 avenue Jack Kilby
06270 Villeneuve-Loubet Cedex,
FRANCE

Contents

Chapter 1

Main Page

This document contains information about the Texas Instruments implementation of the Open Event Machine.

The file event_machine.h represents the interface of the event machine library. It includes all the specifications needed to use the library. It shall be included as follows to access the event machine functionality.

```
#include <ti/runtime/openem/event_machine.h>
```

The file event_machine_firmware.h contains the binary of the micro risc controller implementing the event machine scheduler. The binary shall be loaded in the micro risc controller when initializing the QMSS. This file shall be included as follows.

```
#include <ti/runtime/openem/firmware/event_machine_firmware.h>
```

Limitations of the Texas Instruments implementation of the Open Event Machine are listed in the release notes document.

This document is divided into the following sections:

Chapter 2

Introduction

2.1 General

Event Machine (EM) is an architectural abstraction and framework of an event driven multicore optimized processing concept originally developed for networking data plane. It offers an easy programming concept for scalable and dynamically load balanced multicore applications with a very low overhead run-to-completion principle.

Main elements in the concept are events, queues, scheduler, dispatcher and the execution objects (EO). Event is an application specific piece of data (like a message or a network packet) describing work, something to do. Any processing in EM must be triggered by an event. Events are sent to asynchronous application specific queues. A single thread on all cores of an EM instance runs a dispatcher loop (a "core" is used here to refer to a core or one HW thread on multi-threaded cores). Dispatcher interfaces with the scheduler and asks for an event. Scheduler evaluates the state of all queues and gives the highest priority event to the dispatcher, which forwards it to the EO mapped to the queue the event came from by calling the registered receive function. As the event is handled and receive function returns, the next event is received from the scheduler and again forwarded to the mapped EO. This happens in parallel on all cores included. Everything is highly efficient run to completion single thread, no context switching nor pre-emption (priorities are handled by the scheduler). EM can run on bare metal for best performance or under an operating system with special arrangements (e.g. one thread per core with thread affinity).

The concept and this API are made to be easy to implement for multiple general purpose or networking oriented multicore packet processing systems on chip, which typically also contain accelerators for packet processing needs. Efficient integration with modern HW accelerators has been a major driver in EM concept.

One general principle of this API is that all calls are multicore safe, i.e. no data structure gets broken, if calls are simultaneously made by multiple cores, but unless explicitly documented per API call, the application also needs to take the parallel world into consideration. For example if one core asks for a queue mode and another one changes the mode at the same time, the returned mode may be invalid (valid data, but the old or the new!). Thus modifications should be done under atomic context (if load balancing

is used) or otherwise synchronized by the application. One simple way of achieving this is to use one EO with an atomic queue to do all the management functionality. That guarantees synchronized operations (but also serializes them limiting performance)

EM_64_BIT or EM_32_BIT (needs to be defined at makefile) defines whether (most of) the types used in the API are 32 or 64 bits wide. NOTE, that this is a major decision, since it may limit value passing between different systems using the defined types directly. Using 64-bits may allow more efficient underlying implementation, as more data can be coded in 64-bit identifiers for instance.

2.2 Some principles

- This API attempts to guide towards a portable application architecture, but is not defined for portability by re-compilation. Many things are system specific giving more possibilities for efficient use of HW resources.
- EM does not define event content (one exception, see em_alloc()). This is a choice made for performance, since most HW devices use proprietary descriptors. This API enables to use those directly.
- EM does not define detailed queue scheduling disciplines or API to set those up (or actually anything to configure a system). The priority value in this API is a (mapped) system specific QoS class label only
- In general EM does not implement full SW platform or middleware solution, it implements a sub- set of such, a driver level part. For best performance it can be used directly from applications.

2.3 Open Event Machine optional fork-join helper.

An event group can be used to trigger join of parallel operations. The number of parallel operations need to be known by the event group creator, but the separate events handlers don't need to know anything about the other related events.

1. a group is created with em_event_group_create()

2. the number of parallel events is set with em_event_group_apply()

3. the parallel events are sent normally, but using em_send_group() instead of em_send()

4. as the receive function of the last event is completed, the given notification event(s) are sent automatically and can trigger the next operation

5. the sequence continues from step 2. for new set of events (if the group is reused)

So here the original initiator only needs to know how the task is split into parallel events, the event handlers and the one continuing the work (join) are not involved (assuming the task itself can be separately processed)

Note, that this only works with events targeted to an EO, i.e. SW events

2.4 32 bit version

This is documentation represent the 32 bit version of Event Machine API. Define EM_-64_BIT or EM_32_BIT to select between 64 and 32 bit versions.

Chapter 3

Error

3.1 mechanism

The error handler prototype to be provided by the application to the OpenEM when activating the error mecanism shall comply with the following prototype:

em_status_t (∗em_error_handler_t)(em_eo_t eo, em_status_t error, em_escope_t escope, va_list args)

In the current implementation, the OpenEM does not consider using other parameters other than the execution object (eo), the error code (error) and the error scope (escope).

See also

em_error_handler_t(), em_register_error_handler(), em_unregister_error_handler, em_eo_register_error_handler, em_eo_unregister_error_handler()

Chapter 4

Trace

4.1 mechanism

The trace handler prototype to be provided by the application to the OpenEM when activating the trace mechanism shall comply with the following prototype:

em_status_t (*ti_em_trace_handler_t)(ti_em_tscope_t tscope, ...)

In the curent implementation, the OpenEM assumes the trace of the following APIs to be associated to the following set of input parameters:

- void ti_em_preschedule(void) -> TI_EM_TSCOPE_PRESCHEDULE
- em_status_t ti_em_dispatch_once (void) -> TI_EM_TSCOPE_DISPATCH
- void* ti_em_claim_local (void) -> TI_EM_TSCOPE_CLAIM_LOCAL, (void*)(buffer_ptr)
- em_event_t em_alloc(size_t size, em_event_type_t type, em_pool_id_t pool_id) -> TI_EM_TSCOPE_ALLOC, (em_event_t)(event), (uint32_t)(size), (em_event_type_t)(type), (em_pool_id_t)(pool_id)
- void em_atomic_processing_end(void) -> TI_EM_TSCOPE_ATOMIC_PROCESSING_END
- void em_free(em_event_t event) -> TI_EM_TSCOPE_FREE, (em_event_t)(event)
- em_status_t em_send(em_event_t event, em_queue_t queue) -> TI_EM_TSCOPE_SEND, (em_event_t)(event), (em_queue_t)(queue)
- em_status_t em_send_group(em_event_t event, em_queue_t queue, em_event_group_t group) -> TI_EM_TSCOPE_SEND, (em_event_t)(event), (em_queue_t)(queue), (em_event_group_t)(eventGroup)

See also

ti_em_trace_handler_t(), ti_em_register_trace_handler(), ti_em_unregister_trace_-handler()

Chapter 5

Module Documentation

5.1 Generic API

Generic declarations.

Data Structures

- struct em_notif_t

Defines

- #define EM_CORE_MASK_SIZE 64
- #define EM_EO_NAME_LEN (32)
- #define EM_EO_UNDEF EM_UNDEF_U32
- #define EM_ERROR 0xffffffff
- #define EM_ERROR_FATAL_MASK (0x80000000u)
- #define EM_ERROR_IS_FATAL(error) (EM_ERROR_FATAL_MASK & (error))
- #define EM_ERROR_SET_FATAL(error) (EM_ERROR_FATAL_MASK | (error))
- #define EM_ESCOPE(escope) (EM_ESCOPE_BIT & (escope))
- #define EM_ESCOPE_ALLOC (EM_ESCOPE_API_MASK | 0x0401)
- #define EM_ESCOPE_API(escope) (((escope) & EM_ESCOPE_MASK) == EM_ESCOPE_API_MASK)
- #define EM_ESCOPE_API_MASK (EM_ESCOPE_BIT | (EM_ESCOPE_API_TYPE << 24))
- #define EM_ESCOPE_API_TYPE (0xFFu)
- #define EM_ESCOPE_ATOMIC_PROCESSING_END (EM_ESCOPE_API_MASK | 0x0404)
- #define EM_ESCOPE_BIT (0x80000000u)
- #define EM_ESCOPE_CORE_COUNT (EM_ESCOPE_API_MASK | 0x0302)

- #define EM_ESCOPE_CORE_ID (EM_ESCOPE_API_MASK | 0x0301)
- #define EM_ESCOPE_EO_ADD_QUEUE (EM_ESCOPE_API_MASK | 0x0204)
- #define EM_ESCOPE_EO_CREATE (EM_ESCOPE_API_MASK | 0x0201)
- #define EM_ESCOPE_EO_DELETE (EM_ESCOPE_API_MASK | 0x0202)
- #define EM_ESCOPE_EO_GET_NAME (EM_ESCOPE_API_MASK | 0x0203)
- #define EM_ESCOPE_EO_REGISTER_ERROR_HANDLER (EM_ESCOPE_-API_MASK | 0x0206)
- #define EM_ESCOPE_EO_REMOVE_QUEUE (EM_ESCOPE_API_MASK | 0x0205)
- #define EM_ESCOPE_EO_START (EM_ESCOPE_API_MASK | 0x0208)
- #define EM_ESCOPE_EO_STOP (EM_ESCOPE_API_MASK | 0x0209)
- #define EM_ESCOPE_EO_UNREGISTER_ERROR_HANDLER (EM_ESCOPE_-API_MASK | 0x0207)
- #define EM_ESCOPE_ERROR (EM_ESCOPE_API_MASK | 0x0503)
- #define EM_ESCOPE_FREE (EM_ESCOPE_API_MASK | 0x0402)
- #define EM_ESCOPE_INTERNAL(escope) (((escope) & EM_ESCOPE_MASK) == EM_ESCOPE_INTERNAL_MASK)
- #define EM_ESCOPE_INTERNAL_MASK (EM_ESCOPE_BIT | (EM_ESCOPE_-INTERNAL_TYPE << 24))
- #define EM_ESCOPE_INTERNAL_TYPE (0x00u)
- #define EM_ESCOPE_MASK (0xFF000000)
- #define EM_ESCOPE_QUEUE_CREATE (EM_ESCOPE_API_MASK | 0x0001)
- #define EM_ESCOPE_QUEUE_CREATE_STATIC (EM_ESCOPE_API_MASK | 0x0002)
- #define EM_ESCOPE_QUEUE_DELETE (EM_ESCOPE_API_MASK | 0x0003)
- #define EM_ESCOPE_QUEUE_DISABLE (EM_ESCOPE_API_MASK | 0x0006)
- #define EM_ESCOPE_QUEUE_DISABLE_ALL (EM_ESCOPE_API_MASK | 0x0007)
- #define EM_ESCOPE_QUEUE_ENABLE (EM_ESCOPE_API_MASK | 0x0004)
- #define EM_ESCOPE_QUEUE_ENABLE_ALL (EM_ESCOPE_API_MASK | 0x0005)
- #define EM_ESCOPE_QUEUE_GET_CONTEXT (EM_ESCOPE_API_MASK | 0x0009)
- #define EM_ESCOPE_QUEUE_GET_GROUP (EM_ESCOPE_API_MASK | 0x000D)
- #define EM_ESCOPE_QUEUE_GET_NAME (EM_ESCOPE_API_MASK | 0x000A)
- #define EM_ESCOPE_QUEUE_GET_PRIORITY (EM_ESCOPE_API_MASK | 0x000B)
- #define EM_ESCOPE_QUEUE_GET_TYPE (EM_ESCOPE_API_MASK | 0x000C)
- #define EM_ESCOPE_QUEUE_GROUP_CREATE (EM_ESCOPE_API_MASK | 0x0101)
- #define EM_ESCOPE_QUEUE_GROUP_DELETE (EM_ESCOPE_API_MASK | 0x0102)
- #define EM_ESCOPE_QUEUE_GROUP_FIND (EM_ESCOPE_API_MASK | 0x0104)
- #define EM_ESCOPE_QUEUE_GROUP_MASK (EM_ESCOPE_API_MASK | 0x0105)
- #define EM_ESCOPE_QUEUE_GROUP_MODIFY (EM_ESCOPE_API_MASK | 0x0103)

- #define EM_ESCOPE_QUEUE_SET_CONTEXT (EM_ESCOPE_API_MASK | 0x0008)
- #define EM_ESCOPE_REGISTER_ERROR_HANDLER (EM_ESCOPE_API_MASK | 0x0501)
- #define EM_ESCOPE_SEND (EM_ESCOPE_API_MASK | 0x0403)
- #define EM_ESCOPE_UNREGISTER_ERROR_HANDLER (EM_ESCOPE_API_MASK | 0x0502)
- #define EM_EVENT_GROUP_MAX_NOTIF (8)
- #define EM_EVENT_GROUP_UNDEF EM_UNDEF_U32
- #define EM_EVENT_UNDEF (0x00000000u)
- #define EM_MAX_EOS (256)
- #define EM_MAX_EVENT_GROUPS (1024)
- #define EM_MAX_QUEUE_GROUPS (16u)
- #define EM_MAX_QUEUES (TI_EM_QUEUE_SET_NUM * TI_EM_QUEUE_NUM_IN_SET)
- #define EM_OK 0
- #define EM_POOL_DEFAULT (0u)
- #define EM_QUEUE_GROUP_DEFAULT (1u)
- #define EM_QUEUE_GROUP_NAME_LEN (8u)
- #define EM_QUEUE_GROUP_UNDEF EM_UNDEF_U32
- #define EM_QUEUE_NAME_LEN (32)
- #define EM_QUEUE_STATIC_MAX ((TI_EM_QUEUE_SET_NUM * TI_EM_STATIC_QUEUE_NUM_IN_SET) - 1)
- #define EM_QUEUE_STATIC_MIN (1)
- #define EM_QUEUE_STATIC_NUM (EM_QUEUE_STATIC_MAX - EM_QUEUE_STATIC_MIN + 1)
- #define EM_QUEUE_UNDEF EM_UNDEF_U32
- #define EM_UNDEF_U16 (0x0000u)
- #define EM_UNDEF_U32 (0x00000000u)
- #define EM_UNDEF_U64 (0x0000000000000000u)
- #define EM_UNDEF_U8 (0x00u)
- #define PRI_EGRP PRIu32
- #define PRI_EO PRIu32
- #define PRI_QGRP PRIu32
- #define PRI_QUEUE PRIu32

Typedefs

- typedef union em_core_mask_t em_core_mask_t
- typedef uint32_t em_eo_t
- typedef em_status_t(* em_error_handler_t)(em_eo_t eo, em_status_t error, em_escope_t escope, va_list args)
- typedef uint32_t em_escope_t
- typedef uint32_t em_event_group_t
- typedef uint32_t em_event_t
- typedef enum em_event_type_major_e em_event_type_major_e

- typedef enum em_event_type_sw_minor_e em_event_type_sw_minor_e
- typedef uint32_t em_event_type_t
- typedef struct em_notif_t em_notif_t
- typedef uint32_t em_pool_id_t
- typedef uint32_t em_queue_group_t
- typedef enum em_queue_prio_e em_queue_prio_e
- typedef uint32_t em_queue_prio_t
- typedef uint32_t em_queue_t
- typedef enum em_queue_type_e em_queue_type_e
- typedef uint32_t em_queue_type_t
- typedef void(∗ em_receive_func_t)(void ∗eo_ctx, em_event_t event, em_event_type_t type, em_queue_t queue, void ∗q_ctx)
- typedef em_status_t(∗ em_start_func_t)(void ∗eo_ctx, em_eo_t eo)
- typedef em_status_t(∗ em_start_local_func_t)(void ∗eo_ctx, em_eo_t eo)
- typedef enum em_status_e em_status_e
- typedef uint32_t em_status_t
- typedef em_status_t(∗ em_stop_func_t)(void ∗eo_ctx, em_eo_t eo)
- typedef em_status_t(∗ em_stop_local_func_t)(void ∗eo_ctx, em_eo_t eo)

Enumerations

- enum em_event_type_major_e
- enum em_event_type_sw_minor_e
- enum em_queue_prio_e
- enum em_queue_type_e
- enum em_status_e

Functions

- em_event_t em_alloc (size_t size, em_event_type_t type, em_pool_id_t pool_id)
- void em_atomic_processing_end (void)
- int em_core_count (void)
- int em_core_id (void)
- int em_core_id_get_physical (int core)
- static void em_core_mask_clr (int core, em_core_mask_t ∗mask)
- static void em_core_mask_copy (em_core_mask_t ∗dest, const em_core_mask_t ∗src)
- static int em_core_mask_count (const em_core_mask_t ∗mask)
- static int em_core_mask_equal (const em_core_mask_t ∗mask1, const em_core_mask_t ∗mask2)
- void em_core_mask_get_physical (em_core_mask_t ∗phys, const em_core_mask_t ∗logic)
- static int em_core_mask_isset (int core, const em_core_mask_t ∗mask)
- static int em_core_mask_iszero (const em_core_mask_t ∗mask)
- static void em_core_mask_set (int core, em_core_mask_t ∗mask)
- static void em_core_mask_set_count (int count, em_core_mask_t ∗mask)

- static void em_core_mask_zero (em_core_mask_t *mask)
- em_status_t em_eo_add_queue (em_eo_t eo, em_queue_t queue)
- em_eo_t em_eo_create (const char *name, em_start_func_t start, em_start_local_func_t local_start, em_stop_func_t stop, em_stop_local_func_t local_stop, em_receive_func_t receive, const void *eo_ctx)
- em_status_t em_eo_delete (em_eo_t eo)
- size_t em_eo_get_name (em_eo_t eo, char *name, size_t maxlen)
- em_status_t em_eo_register_error_handler (em_eo_t eo, em_error_handler_t handler)
- em_status_t em_eo_remove_queue (em_eo_t eo, em_queue_t queue, int num_notif, const em_notif_t *notif_tbl)
- em_status_t em_eo_start (em_eo_t eo, em_status_t *result, int num_notif, const em_notif_t *notif_tbl)
- em_status_t em_eo_stop (em_eo_t eo, int num_notif, const em_notif_t *notif_tbl)
- em_status_t em_eo_unregister_error_handler (em_eo_t eo)
- void em_error (em_status_t error, em_escope_t escope,...)
- int em_error_format_string (char *str, size_t size, em_eo_t eo, em_status_t error, em_escope_t escope, va_list args)
- em_status_t em_event_group_apply (em_event_group_t group, int count, int num_notif, const em_notif_t *notif_tbl)
- em_event_group_t em_event_group_create (void)
- em_event_group_t em_event_group_current (void)
- em_status_t em_event_group_delete (em_event_group_t event_group)
- em_status_t em_event_group_increment (int count)
- void * em_event_pointer (const em_event_t event)
- void em_free (em_event_t event)
- em_event_type_t em_get_type_major (em_event_type_t type)
- em_event_type_t em_get_type_minor (em_event_type_t type)
- em_queue_t em_queue_create (const char *name, em_queue_type_t type, em_queue_prio_t prio, em_queue_group_t group)
- em_status_t em_queue_create_static (const char *name, em_queue_type_t type, em_queue_prio_t prio, em_queue_group_t group, em_queue_t queue)
- em_status_t em_queue_delete (em_queue_t queue)
- em_status_t em_queue_disable (em_queue_t queue, int num_notif, const em_notif_t *notif_tbl)
- em_status_t em_queue_disable_all (em_eo_t eo, int num_notif, const em_notif_t *notif_tbl)
- em_status_t em_queue_enable (em_queue_t queue)
- em_status_t em_queue_enable_all (em_eo_t eo)
- void * em_queue_get_context (em_queue_t queue)
- em_queue_group_t em_queue_get_group (em_queue_t queue)
- size_t em_queue_get_name (em_queue_t queue, char *name, size_t maxlen)
- em_queue_prio_t em_queue_get_priority (em_queue_t queue)
- em_queue_type_t em_queue_get_type (em_queue_t queue)
- em_queue_group_t em_queue_group_create (const char *name, const em_core_mask_t *mask, int num_notif, const em_notif_t *notif_tbl)

- em_status_t em_queue_group_delete (em_queue_group_t group, int num_notif, const em_notif_t *notif_tbl)
- em_queue_group_t em_queue_group_find (const char *name)
- em_status_t em_queue_group_mask (em_queue_group_t group, em_core_mask_t *mask)
- em_status_t em_queue_group_modify (em_queue_group_t group, const em_core_mask_t *new_mask, int num_notif, const em_notif_t *notif_tbl)
- em_status_t em_queue_set_context (em_queue_t queue, const void *context)
- em_status_t em_register_error_handler (em_error_handler_t handler)
- em_status_t em_send (em_event_t event, em_queue_t queue)
- em_status_t em_send_group (em_event_t event, em_queue_t queue, em_event_group_t group)
- em_status_t em_unregister_error_handler (void)

5.1.1 Detailed Description

Generic declarations.

5.1.2 Define Documentation

5.1.2.1 #define EM_CORE_MASK_SIZE 64

Size of the core mask in bits

5.1.2.2 #define EM_EO_NAME_LEN (32)

Max EO name string lenght.

Note

This value can be modified by an application. But the event machine library needs to be recompiled.

5.1.2.3 #define EM_EO_UNDEF EM_UNDEF_U32

Invalid EO id

5.1.2.4 #define EM_ERROR 0xffffffff

Operation not successful

5.1.2.5 #define EM_ERROR_FATAL_MASK (0x80000000u)

Fatal error mask

5.1.2.6 #define EM_ERROR_IS_FATAL(*error*) (EM_ERROR_FATAL_MASK & (error))

Test if error is fatal

5.1.2.7 #define EM_ERROR_SET_FATAL(*error*) (EM_ERROR_FATAL_MASK | (error))

Set a fatal error code

5.1.2.8 #define EM_ESCOPE(*escope*) (EM_ESCOPE_BIT & (escope))

Test if the error scope identifies an EM function (API or other internal)

5.1.2.9 #define EM_ESCOPE_ALLOC (EM_ESCOPE_API_MASK | 0x0401)

Error scope for the em_alloc API.

5.1.2.10 #define EM_ESCOPE_API(*escope*) (((escope) & EM_ESCOPE_MASK) == EM_ESCOPE_API_MASK)

Test if the error scope identifies an API function

5.1.2.11 #define EM_ESCOPE_API_MASK (EM_ESCOPE_BIT | (EM_ESCOPE_API_TYPE << 24))

EM API functions error scope mask

5.1.2.12 #define EM_ESCOPE_API_TYPE (0xFFu)

EM API functions error scope type

5.1.2.13 #define EM_ESCOPE_ATOMIC_PROCESSING_END (EM_ESCOPE_API_MASK | 0x0404)

Error scope for the em_atomic_processing_end API.

5.1.2.14 #define EM_ESCOPE_BIT (0x80000000u)

All EM internal error scopes should have bit 31 set NOTE: High bit is RESERVED for EM internal escopes and should not be used by the application.

5.1.2.15 #define EM_ESCOPE_CORE_COUNT (EM_ESCOPE_API_MASK | 0x0302)

Error scope for the em_core_count API.

5.1.2.16 #define EM_ESCOPE_CORE_ID (EM_ESCOPE_API_MASK | 0x0301)

Error scope for the em_core_id API.

5.1.2.17 #define EM_ESCOPE_EO_ADD_QUEUE (EM_ESCOPE_API_MASK | 0x0204)

Error scope for the em_eo_add_queue API.

5.1.2.18 #define EM_ESCOPE_EO_CREATE (EM_ESCOPE_API_MASK | 0x0201)

Error scope for the em_eo_create API.

5.1.2.19 #define EM_ESCOPE_EO_DELETE (EM_ESCOPE_API_MASK | 0x0202)

Error scope for the em_eo_delete API.

5.1.2.20 #define EM_ESCOPE_EO_GET_NAME (EM_ESCOPE_API_MASK | 0x0203)

Error scope for the em_eo_get_name API.

5.1.2.21 #define EM_ESCOPE_EO_REGISTER_ERROR_HANDLER (EM_ESCOPE_API_MASK | 0x0206)

Error scope for the em_eo_register_error_handler API.

5.1.2.22 #define EM_ESCOPE_EO_REMOVE_QUEUE (EM_ESCOPE_API_MASK | 0x0205)

Error scope for the em_eo_remove_queue API.

5.1.2.23 #define EM_ESCOPE_EO_START (EM_ESCOPE_API_MASK | 0x0208)

Error scope for the em_eo_start API.

5.1.2.24 #define EM_ESCOPE_EO_STOP (EM_ESCOPE_API_MASK | 0x0209)

Error scope for the em_eo_stop API.

5.1.2.25 #define EM_ESCOPE_EO_UNREGISTER_ERROR_HANDLER (EM_ESCOPE_API_MASK | 0x0207)

Error scope for the em_eo_unregister_error_handler API.

5.1.2.26 #define EM_ESCOPE_ERROR (EM_ESCOPE_API_MASK | 0x0503)

Error scope for the em_error API.

5.1.2.27 #define EM_ESCOPE_FREE (EM_ESCOPE_API_MASK | 0x0402)

Error scope for the em_free API.

5.1.2.28 #define EM_ESCOPE_INTERNAL(*escope*) (((escope) & EM_ESCOPE_MASK) == EM_ESCOPE_INTERNAL_MASK)

Test if the error scope identifies an EM Internal function

5.1.2.29 #define EM_ESCOPE_INTERNAL_MASK (EM_ESCOPE_BIT | (EM_ESCOPE_INTERNAL_TYPE << 24))

EM internal error scope type

5.1.2.30 #define EM_ESCOPE_INTERNAL_TYPE (0x00u)

EM internal error scope mask

5.1.2.31 #define EM_ESCOPE_MASK (0xFF000000)

Mask selects the high byte of the 32-bit escope

5.1.2.32 #define EM_ESCOPE_QUEUE_CREATE (EM_ESCOPE_API_MASK | 0x0001)

Error scope for the em_queue_create API.

5.1.2.33 #define EM_ESCOPE_QUEUE_CREATE_STATIC (EM_ESCOPE_API_MASK | 0x0002)

Error scope for the em_queue_create_static API.

5.1.2.34 #define EM_ESCOPE_QUEUE_DELETE (EM_ESCOPE_API_MASK | 0x0003)

Error scope for the em_queue_delete API.

5.1.2.35 #define EM_ESCOPE_QUEUE_DISABLE (EM_ESCOPE_API_MASK | 0x0006)

Error scope for the em_queue_disable API.

5.1.2.36 #define EM_ESCOPE_QUEUE_DISABLE_ALL (EM_ESCOPE_API_MASK | 0x0007)

Error scope for the em_queue_disable_all API.

5.1.2.37 #define EM_ESCOPE_QUEUE_ENABLE (EM_ESCOPE_API_MASK | 0x0004)

Error scope for the em_queue_enable API.

5.1.2.38 #define EM_ESCOPE_QUEUE_ENABLE_ALL (EM_ESCOPE_API_MASK | 0x0005)

Error scope for the em_queue_enable_all API.

5.1.2.39 #define EM_ESCOPE_QUEUE_GET_CONTEXT (EM_ESCOPE_API_MASK | 0x0009)

Error scope for the em_queue_get_context API.

5.1.2.40 #define EM_ESCOPE_QUEUE_GET_GROUP (EM_ESCOPE_API_MASK | 0x000D)

Error scope for the em_queue_group_create API.

5.1.2.41 #define EM_ESCOPE_QUEUE_GET_NAME (EM_ESCOPE_API_MASK | 0x000A)

Error scope for the em_queue_get_name API.

5.1.2.42 #define EM_ESCOPE_QUEUE_GET_PRIORITY (EM_ESCOPE_API_MASK | 0x000B)

Error scope for the em_queue_get_priority API.

5.1.2.43 #define EM_ESCOPE_QUEUE_GET_TYPE (EM_ESCOPE_API_MASK | 0x000C)

Error scope for the em_queue_get_type API.

5.1.2.44 #define EM_ESCOPE_QUEUE_GROUP_CREATE (EM_ESCOPE_API_MASK | 0x0101)

Error scope for the em_queue_group_create API.

5.1.2.45 #define EM_ESCOPE_QUEUE_GROUP_DELETE (EM_ESCOPE_API_MASK | 0x0102)

Error scope for the em_queue_group_delete API.

5.1.2.46 #define EM_ESCOPE_QUEUE_GROUP_FIND (EM_ESCOPE_API_MASK | 0x0104)

Error scope for the em_queue_group_find API.

5.1.2.47 #define EM_ESCOPE_QUEUE_GROUP_MASK (EM_ESCOPE_API_MASK | 0x0105)

Error scope for the em_queue_group_mask API.

5.1.2.48 #define EM_ESCOPE_QUEUE_GROUP_MODIFY (EM_ESCOPE_API_MASK | 0x0103)

Error scope for the em_queue_group_modify API.

5.1.2.49 #define EM_ESCOPE_QUEUE_SET_CONTEXT (EM_ESCOPE_API_MASK | 0x0008)

Error scope for the em_queue_set_context API.

5.1.2.50 #define EM_ESCOPE_REGISTER_ERROR_HANDLER (EM_ESCOPE_API_MASK | 0x0501)

Error scope for the em_register_error_handler API.

5.1.2.51 #define EM_ESCOPE_SEND (EM_ESCOPE_API_MASK | 0x0403)

Error scope for the em_send API.

5.1.2.52 #define EM_ESCOPE_UNREGISTER_ERROR_HANDLER (EM_ESCOPE_API_MASK | 0x0502)

Error scope for the em_unregister_error_handler API.

5.1.2.53 #define EM_EVENT_GROUP_MAX_NOTIF (8)

Maximum number of event notifications supported by one event group.

5.1.2.54 #define EM_EVENT_GROUP_UNDEF EM_UNDEF_U32

Invalid event group id

5.1.2.55 #define EM_EVENT_UNDEF (0x00000000u)

Undefined event

5.1.2.56 #define EM_MAX_EOS (256)

Maximum number of execution objects supported by the event machine. EM_MAX_-EOS <= TI_EM_EO_NUM_MAX.

Note

This value can be modified by an application. But the event machine library needs to be recompiled.

5.1.2.57 #define EM_MAX_EVENT_GROUPS (1024)

Maximum number of event groups supported by the event machine. EM_MAX_-EVENT_GROUPS <= TI_EM_EVENT_GROUP_NUM_MAX.

Note

This value can be modified by an application. But the event machine library needs to be recompiled.

5.1.2.58 #define EM_MAX_QUEUE_GROUPS (16u)

Maximum number of queue groups supported by the event machine.

Note

This value can be modified by an application. But the event machine library needs to be recompiled.

5.1.2.59 #define EM_MAX_QUEUES (TI_EM_QUEUE_SET_NUM * TI_EM_QUEUE_NUM_IN_SET)

Maximum number of queues supported by the event machine.

Note

This value can be modified by an application. But the event machine library needs to be recompiled.

5.1.2.60 #define EM_OK 0

Operation successful

5.1.2.61 #define EM_POOL_DEFAULT (0u)

Define default memory pool

5.1.2.62 #define EM_QUEUE_GROUP_DEFAULT (1u)

Default queue group for EM

5.1.2.63 #define EM_QUEUE_GROUP_NAME_LEN (8u)

Max queue group name string lenght.

Note

This value can be modified by an application. But the event machine library needs to be recompiled.

5.1.2.64 #define EM_QUEUE_GROUP_UNDEF EM_UNDEF_U32

Invalid queue group id

5.1.2.65 #define EM_QUEUE_NAME_LEN (32)

Max queue name string length.

Note

This value can be modified by an application. But the event machine library needs to be recompiled.

5.1.2.66 #define EM_QUEUE_STATIC_MAX ((TI_EM_QUEUE_SET_NUM * TI_EM_STATIC_QUEUE_NUM_IN_SET) - 1)

Static queues: maximal value

5.1.2.67 #define EM_QUEUE_STATIC_MIN (1)

Static queues: minimal value

5.1.2.68 #define EM_QUEUE_STATIC_NUM (EM_QUEUE_STATIC_MAX - EM_QUEUE_STATIC_MIN + 1)

Static queues: number of status queues

5.1.2.69 #define EM_QUEUE_UNDEF EM_UNDEF_U32

Invalid queue

5.1.2.70 #define EM_UNDEF_U16 (0x0000u)

Invalid identifier (16-bit)

5.1.2.71 #define EM_UNDEF_U32 (0x00000000u)

Invalid identifier (32-bit)

5.1.2.72 #define EM_UNDEF_U64 (0x0000000000000000u)

Invalid identifier (64-bit)

5.1.2.73 #define EM_UNDEF_U8 (0x00u)

Invalid identifier (8-bit)

5.1.2.74 #define PRI_EGRP PRIu32

em_event_group_t printf format

5.1.2.75 #define PRI_EO PRIu32

em_eo_t printf format

5.1.2.76 #define PRI_QGRP PRIu32

em_queue_group_t printf format

5.1.2.77 #define PRI_QUEUE PRIu32

em_queue_t printf format

5.1.3 Typedef Documentation

5.1.3.1 typedef union em_core_mask_t em_core_mask_t

Type for queue group core mask. Each bit represents one core, core 0 is the lsb (1 << em_core_id()) Note, that EM will enumerate the core identifiers to always start from 0 and be contiguous meaning the core numbers are not necessarily physical. This type can handle up to 64 cores.

See also

em_queue_group_create()

5.1.3.2 typedef uint32_t em_eo_t

Execution Object identifier

See also

em_eo_create()

5.1.3.3 typedef em_status_t(* em_error_handler_t)(em_eo_t eo, em_status_t error, em_escope_t escope, va_list args)

Error handler.

Error handler maybe called after EM notices an error or user have called em_error().

User can register EO specific and/or EM global error handlers. When an error is noticed, EM calls EO specific error handler, if registered. If there's no EO specific handler registered (for the EO) or the error is noticed outside of an EO context, EM calls the global error handler (if registered). If no error handlers are found, EM just returns an error code depending on the API function.

Error handler is called with the original error code from the API call or em_error(). Error scope identifies the source of the error and how the error code and variable arguments should be interpreted (number of arguments and types).

Parameters

eo	Execution object id
error	The error code
escope	Error scope. Identifies the scope for interpreting the error code and variable arguments.
args	Variable number and type of arguments

Returns

The function may not return depending on implementation/error code/error scope. If it returns, it can return the original or modified error code or even EM_OK, if it could fix the problem.

See also

em_register_error_handler(), em_eo_register_error_handler()

5.1.3.4 typedef uint32_t em_escope_t

Error scope.

Identifies the error scope for interpreting error codes and variable arguments.

See also

em_error_handler_t(), em_error()

5.1.3.5 typedef uint32_t em_event_group_t

Event group id. This is used for fork-join event handling.

See also

em_event_group_create()

5.1.3.6 typedef uint32_t em_event_t

Event type

In Event Machine application processing is driven by events. An event describes a piece of work. Structure of an event is implementation and event type specific. It may be a directly accessible buffer of memory, a descriptor containing a list of buffer pointers, a descriptor of a packet buffer, etc.

Applications use event type to interpret the event structure.

Since em_event_t may not carry a direct pointer value to the event structure, em_event_pointer() must be used to translate an event to an event structure pointer (for maintaining portability).

See also

em_event_pointer()

5.1.3.7 typedef enum em_event_type_major_e em_event_type_major_e

The structure describes the major event types

5.1.3.8 typedef enum em_event_type_sw_minor_e em_event_type_sw_minor_e

The structure describes the Minor event types for the major EM_EVENT_TYPE_SW type

5.1.3.9 typedef uint32_t em_event_type_t

Event type. This is given to application for each received event and also needed for event allocation. It's an integer, but split into major and minor part. major-field categorizes the event and minor is more detailed system specific description. Major-part will not change by HW, but minor can be HW/SW platform specific and thus could be split into more sub-fields as needed. Application should use the access functions for reading major and minor part.

The only event type with defined content is EM_EVENT_TYPE_SW with minor type 0, which needs to be portable (direct pointer to data).

See also

em_get_type_major(), em_get_type_minor(), em_receive_func_t()

5.1.3.10 typedef struct em_notif_t em_notif_t

Notification structure allows user to define a notification event and destination queue pair. EM will notify user by sending the event into the queue.

5.1.3.11 typedef uint32_t em_pool_id_t

Memory pool id.

Defines memory pool in em_alloc(). Default pool id is defined by EM_POOL_DEFAULT.

See also

em_alloc(), event_machine_hw_config.h

5.1.3.12 typedef uint32_t em_queue_group_t

Queue group identifier

Each queue belongs to one queue group, that defines a core mask for scheduling events, i.e. define which cores participate in the load balancing. Group can also allow only a single core for no load balancing.

Groups needs to be created as needed. One default group (EM_QUEUE_GROUP_-DEFAULT) always exists, and that allows scheduling to all the cores running this execution binary instance.

See also

em_queue_group_create()

5.1.3.13 typedef enum em_queue_prio_e em_queue_prio_e

The structure describes the portable queue priorities

5.1.3.14 typedef uint32_t em_queue_prio_t

Queue priority class

Queue priority defines a system dependent QoS class, not just an absolute priority. EM gives freedom to implement the actual scheduling disciplines and the corresponding numeric values as needed, i.e. the actual values are system dependent and thus not portable, but the 5 pre-defined enums (em_queue_prio_e) are always valid. Application platform or middleware needs to define and distribute the other available values.

See also

em_queue_create(), event_machine_hw_config.h

5.1.3.15 typedef uint32_t em_queue_t

Queue identifier

See also

em_queue_create(), em_receive_func_t(), em_send()

5.1.3.16 typedef enum em_queue_type_e em_queue_type_e

The structure describes the queue types

5.1.3.17 typedef uint32_t em_queue_type_t

Queue type.

Affects the scheduling principle

See also

em_queue_create(), event_machine_hw_config.h

5.1.3.18 typedef void(* em_receive_func_t)(void *eo_ctx, em_event_t event, em_event_type_t type, em_queue_t queue, void *q_ctx)

Receive event

Application receives events through the EO receive function. It implements main part of the application logic. EM calls the receive function when it has dequeued an event from one of the EO's queues. Application processes the event and returns immediately (in run-to-completion fashion).

On multicore systems several events (from the same or different queues) may be dequeued in parallel and thus same receive function may be executed concurrently on several cores. Parallel execution may be limited by queue group setup or using queues with atomic scheduling mode.

EO and queue context pointers are user defined (in EO and queue creation time) and may be used any way needed. For example, EO context may be used to store global EO state information, which is common to all queues and events. In addition, queue context may be used to store queue (or user data flow) specific state data. EM never touches the data, just passes the pointers.

Event (handle) must be converted to an event structure pointer with em_event_pointer(). Event type specifies the event structure, which is implementation or application specific. Queue id specifies the queue which the event was dequeued from.

Parameters

eo_ctx	EO context data. The pointer is passed in em_eo_create(), EM does not touch the data.

event	Event handle
type	Event type
queue	Queue from which this event came from
q_ctx	Queue context data. The pointer is passed in em_queue_set_context(), EM does not touch the data.

See also

em_event_pointer(), em_free(), em_alloc(), em_send(), em_queue_set_context(), em_eo_create()

5.1.3.19 typedef em_status_t(∗ em_start_func_t)(void ∗eo_ctx, em_eo_t eo)

Execution object start, global.

If load balancing/several cores share the EO, this function is called once on one core only (any). Purpose of this global start is to provide a placeholder for first level initialization, like allocating memory and initializing shared data. After this global start returns, the core local version (if defined), is called (see em_start_local_func_t below). If there is no core local start, then event dispatching is immediately enabled.

If Execution object does not return EM_OK, the system will not call the core local init and will not enable event dispatching.

This function should never be called directly from the application, but using the, which maintains state information!

Parameters

eo_ctx	Execution object internal state/instance data
eo	Execution object id

Returns

EM_OK if successful.

See also

em_eo_start(), em_eo_create()

5.1.3.20 typedef em_status_t(∗ em_start_local_func_t)(void ∗eo_ctx, em_eo_t eo)

Execution object start, core local.

This is similar to the global start above, but this one is called after the global start has completed and is called on all cores.

Purpose of this optional local start is to work as a placeholder for core local initialization, e.g. allocating core local memory for example. The global start is only called on one core. The use of local start is optional. Note, that application should never directly call this function, this will be called via em_eo_start().

If this does not return EM_OK on all involved cores, the event dispatching is not enabled.

Parameters

eo_ctx	Execution object internal state/instance data
eo	Execution object id

Returns

EM_OK if successful.

See also

em_eo_start(), em_eo_create()

5.1.3.21 typedef enum em_status_e em_status_e

The structure lists the error codes

5.1.3.22 typedef uint32_t em_status_t

Error/Status code. EM_OK (0) is the general code for success, other values describe failed operation.

See also

event_machine_hw_config.h, em_error_handler_t(), em_error()

5.1.3.23 typedef em_status_t(∗ em_stop_func_t)(void ∗eo_ctx, em_eo_t eo)

Execution object stop, global.

If load balancing/several cores share the EO, this function is called once on one core (any) after the (optional) core local stop return on all cores. System disables event dispatching before calling this and also makes sure this does not get called before all cores have been notified the stop condition for this EO (can't dispatch new events) event if there is no core local stop defined. Return value is only for logging purposes, EM does not use it currently.

Note, that application should never directly call this stop function, but use the em_eo_stop() instead, which maintains state information and guarantees synchronized operation.

Parameters

eo_ctx	Execution object internal state data
eo	Execution object id

Returns

EM_OK if successful.

See also

em_eo_stop(), em_eo_create()

5.1.3.24 typedef em_status_t(* em_stop_local_func_t)(void *eo_ctx, em_eo_t eo)

Execution object stop, core local.

If load balancing/several cores share the EO, this function is called once on each core before the global stop (reverse order of start). System disables event dispatching before calling this and also makes sure this does not get called before the core has been notified the stop condition for this EO (won't dispatch new events) Return value is only for logging purposes, EM does not use it currently.

Note, that application should never directly call this stop function, em_eo_stop() will trigger this.

Parameters

eo_ctx	Execution object internal state data
eo	Execution object id

Returns

EM_OK if successful.

See also

em_eo_stop(), em_eo_create()

5.1.4 Enumeration Type Documentation

5.1.4.1 enum em_event_type_major_e

The structure describes the major event types

Enumerator:

EM_EVENT_TYPE_SW event from SW (EO)
EM_EVENT_TYPE_PACKET event from packet HW
EM_EVENT_TYPE_TIMER event from timer HW
EM_EVENT_TYPE_CRYPTO event from crypto HW

5.1.4.2 enum em_event_type_sw_minor_e

The structure describes the Minor event types for the major EM_EVENT_TYPE_SW type

5.1.4.3 enum em_queue_prio_e

The structure describes the portable queue priorities

Enumerator:

EM_QUEUE_PRIO_UNDEF Undefined.

EM_QUEUE_PRIO_LOWEST lowest priority

EM_QUEUE_PRIO_LOW low priority

EM_QUEUE_PRIO_NORMAL normal priority

EM_QUEUE_PRIO_HIGH high priority

EM_QUEUE_PRIO_HIGHEST highest priority

5.1.4.4 enum em_queue_type_e

The structure describes the queue types

Enumerator:

EM_QUEUE_TYPE_UNDEF Application receives events one by one, non-concurrently to guarantee exclusive processing and ordering.

EM_QUEUE_TYPE_ATOMIC Application receives events one by one, non-concurrently to guarantee exclusive processing and ordering.

EM_QUEUE_TYPE_PARALLEL Application may receive events fully concurrently, egress event ordering (when processed in parallel) not guaranteed.

EM_QUEUE_TYPE_PARALLEL_ORDERED Application may receive events concurrently, but system takes care of egress order (between two queues)

5.1.4.5 enum em_status_e

The structure lists the error codes

Enumerator:

EM_ERR_BAD_CONTEXT Illegal context for this function call.

EM_ERR_BAD_STATE Illegal (eo, queue, ...) state for this function call.

EM_ERR_BAD_ID ID not from a valid range.

EM_ERR_ALLOC_FAILED Resource allocation failed.

EM_ERR_NOT_FREE Resource already reserved by someone else.

EM_ERR_NOT_FOUND Resource not found.

EM_ERR_TOO_LARGE Value over the limit.

EM_ERR_LIB_FAILED Failure in a library function.

EM_ERR_NOT_IMPLEMENTED Implementation missing (placeholder)

EM_ERR_BAD_POINTER Pointer from bad memory area (e.g. NULL)

EM_ERR Other error. This is the last error code (for bounds checking).

5.1.5 Function Documentation

5.1.5.1 em_event_t em_alloc (size_t *size*, em_event_type_t *type*, em_pool_id_t *pool_id*)

Allocate an event.

Memory address of the allocated event is system specific and can depend on given pool id, event size and type. Returned event (handle) may refer to a memory buffer or a HW specific descriptor, i.e. the event structure is system specific.

Use em_event_pointer() to convert an event (handle) to a pointer to the event structure.

EM_EVENT_TYPE_SW with minor type 0 is reserved for direct portability. It is always guaranteed to return a 64-bit aligned contiguous data buffer, that can directly be used by the application up to the given size (no HW specific descriptors etc are visible).

EM_POOL_DEFAULT can be used as pool id if there's no need to use any specific memory pool.

Additionally it is guaranteed, that two separate buffers never share a cache line to avoid false sharing.

Parameters

size	Event size in octets
type	Event type to allocate
pool_id	Event pool id

Returns

the allocated event or EM_EVENT_UNDEF on an error

See also

em_free(), em_send(), em_event_pointer(), em_receive_func_t()

5.1.5.2 void em_atomic_processing_end (void)

Release atomic processing context.

When an event was received from an atomic queue, the function can be used to release the atomic context before receive function return. After the call, scheduler is allowed to schedule another event from the same queue to another core. This increases parallelism and may improve performance - however the exclusive processing and ordering (!) might be lost after the call.

Can only be called from within the event receive function!

The call is ignored, if current event was not received from an atomic queue.

Pseudo-code example:

```
receive_func(void* eo_ctx, em_event_t event, em_event_type_t type, em_queue_t q
    ueue, void* q_ctx);
```

```
{
    if(is_my_atomic_queue(q_ctx))
    {
        update_sequence_number(event);  // this needs to be done atomically
        em_atomic_processing_end();
        ...                             // do other processing (potentially) in
     parallel
    }
}
```

See also

em_receive_func_t()

5.1.5.3 int em_core_count (void)

The number of cores running within the same EM instance (sharing the EM state).

Returns

Number of EM cores (or HW threads)

See also

em_core_id()

5.1.5.4 int em_core_id (void)

Logical core id.

Returns the logical id of the current core. EM enumerates cores (or HW threads) to always start from 0 and be contiguous, i.e. valid core identifiers are 0...em_core_count()-1

Returns

Current logical core id

See also

em_core_count()

5.1.5.5 int em_core_id_get_physical (int *core*)

Converts a logical core id to a physical core id

Mainly needed when interfacing HW specific APIs

Parameters

core	logical (Event Machine) core id

Returns

Physical core id

5.1.5.6 static void em_core_mask_clr (int *core*, em_core_mask_t * *mask*) [inline, static]

Clear a bit in the mask.

Parameters

core	Core id
mask	Core mask

5.1.5.7 static void em_core_mask_copy (em_core_mask_t * *dest*, const em_core_mask_t * *src*) [inline, static]

Copy core mask

Parameters

dest	Destination core mask
src	Source core mask

5.1.5.8 static int em_core_mask_count (const em_core_mask_t * *mask*) [inline, static]

Count the number of bits set in the mask.

Parameters

mask	Core mask

Returns

Number of bits set

5.1.5.9 static int em_core_mask_equal (const em_core_mask_t * *mask1*, const em_core_mask_t * *mask2*) [inline, static]

Test if two masks are equal

Parameters

mask1	First core mask
mask2	Second core mask

Returns

Non-zero if the two masks are equal

5.1.5.10 void em_core_mask_get_physical (em_core_mask_t * *phys*, const em_core_mask_t * *logic*)

Converts a logical core mask to a physical core mask

Mainly needed when interfacing HW specific APIs

Parameters

phys	Core mask of physical core ids
logic	Core mask of logical (Event Machine) core ids

5.1.5.11 static int em_core_mask_isset (int *core*, const em_core_mask_t * *mask*) [inline, static]

Test if a bit is set in the mask.

Parameters

core	Core id
mask	Core mask

Returns

Non-zero if core id is set in the mask

5.1.5.12 static int em_core_mask_iszero (const em_core_mask_t * *mask*) [inline, static]

Test if the mask is all zero.

Parameters

mask	Core mask

Returns

Non-zero if the mask is all zero

5.1.5.13 static void em_core_mask_set (int *core*, em_core_mask_t * *mask*) [inline, static]

Set a bit in the mask.

Parameters

core	Core id
mask	Core mask

5.1.5.14 static void em_core_mask_set_count (int *count*, em_core_mask_t * *mask*) [inline, static]

Set a range (0...count-1) of bits in the mask.

Parameters

count	Number of bits to set
mask	Core mask

5.1.5.15 static void em_core_mask_zero (em_core_mask_t * *mask*) [inline, static]

Zero the whole mask.

Parameters

mask	Core mask

5.1.5.16 em_status_t em_eo_add_queue (em_eo_t *eo*, em_queue_t *queue*)

Add a queue to an EO.

Note, that this does not enable the queue. Although queues added in (or before) the start function will be enabled automatically.

Parameters

eo	EO id
queue	Queue id

Returns

EM_OK if successful.

See also

em_queue_create(), em_eo_create(), em_queue_enable(), em_eo_remove_queue()

5.1.5.17 em_eo_t em_eo_create (const char * *name*, em_start_func_t *start*, em_start_local_func_t *local_start*, em_stop_func_t *stop*, em_stop_local_func_t *local_stop*, em_receive_func_t *receive*, const void * *eo_ctx*)

Create Execution Object (EO).

This will allocate identifier and initialize internal data for a new EO. It is left in a non-active state, i.e. no events are dispatched before em_eo_start() has called. Start, stop and receive callback function pointers are mandatory parameters.

The name given is copied to EO internal data and can be used e.g. for debugging. The maximum length stored is EM_EO_NAME_LEN.

Parameters

name	Name of the EO (NULL if no name)
start	Start function
local_start	Core local start function (NULL if no local start)
stop	Stop function
local_stop	Core local stop function (NULL if no local stop)
receive	Receive function
eo_ctx	User defined EO context data, EM just passes the pointer (NULL if not context)

Returns

New EO id if successful, otherwise EM_EO_UNDEF

See also

em_eo_start(), em_eo_delete(), em_queue_create(), em_eo_add_queue()
em_start_func_t(), em_stop_func_t(), em_receive_func_t()

5.1.5.18 em_status_t em_eo_delete (em_eo_t *eo*)

Delete Execution Object (EO).

This will immediately delete the given EO and free the identifier.

NOTE, that EO can only be deleted after it has been stopped using em_eo_stop(), otherwise another core might still access the EO data! Deletion will fail, if the EO is not stopped.

This will delete all possibly remaining queues.

Parameters

eo	EO id to delete

Returns

EM_OK if successful.

See also

em_eo_stop(), em_eo_create()

5.1.5.19 size_t em_eo_get_name (em_eo_t *eo*, char * *name*, size_t *maxlen*)

Returns the name given to the EO when it was created.

A copy of the name string (up to 'maxlen' characters) is written to the user buffer 'name'. String is always null terminated even if the given buffer length is less than the name length.

If the EO has no name, function returns 0 and writes empty string.

This is only for debugging purposes.

Parameters

eo	EO id
name	Destination buffer
maxlen	Maximum length (including the terminating '0')

Returns

Number of characters written (excludes the terminating '0')

See also

em_eo_create()

5.1.5.20 em_status_t em_eo_register_error_handler (em_eo_t *eo*, em_error_handler_t *handler*)

Register EO specific error handler.

The EO specific error handler is called if error is noticed or em_error() is called in the context of the EO. Note, the function will override any previously registered error hanler.

Parameters

eo	EO id
handler	New error handler.

Returns

EM_OK if successful.

See also

em_register_error_handler(), em_error_handler_t()

5.1.5.21 em_status_t em_eo_remove_queue (em_eo_t *eo*, em_queue_t *queue*, int *num_notif*, const em_notif_t * *notif_tbl*)

Removes a queue from an EO.

Function disables scheduling of the queue and removes the queue from the EO. The operation is asynchronous, to quarantee that all cores have completed processing of events from the queue (i.e. there's no cores in middle of the receive function) before removing it.

If the caller needs to know when the context deletion actually occurred, the num_notif and notif_tbl can be used. The given notification event(s) will be sent to given queue(s), when the removal has completed. If such notification is not needed, use 0 as num_notif.

If the queue to be removed is still enabled, it will first be disabled.

Parameters

eo	EO id
queue	Queue id to remove
num_notif	How many notification events given, 0 for no notification
notif_tbl	Array of pairs of event and queue identifiers

Returns

EM_OK if successful.

See also

em_eo_add_queue(), em_queue_disable(), em_queue_delete()

5.1.5.22 em_status_t em_eo_start (em_eo_t *eo*, em_status_t * *result*, int *num_notif*, const em_notif_t * *notif_tbl*)

Start Execution Object (EO).

Calls global EO start function. If that returns EM_OK, an internal event to trigger local start is sent to all cores belonging to the queue group of this EO. If the global start function does not return EM_OK the local start is not called and event dispatching is not enabled for this EO.

If the caller needs to know when the EO start was actually completed on all cores, the num_notif and notif_tbl can be used. The given notification event(s) will be sent to given queue(s), when the start is completed on all cores. If local start does not exist the notification(s) are sent as the global start returns. If such notification is not needed, use 0 as num_notif.

Parameters

eo	EO id
result	Optional pointer to em_status_t, which gets updated to the return value of the actual EO global start function
num_notif	If not 0, defines the number of events to send as all cores have returned from the start function (in notif_tbl).
notif_tbl	Array of em_notif_t, the optional notification events (data copied)

Returns

EM_OK if successful.

See also

em_start_func_t(), em_start_local_func_t(), em_eo_stop()

5.1.5.23 em_status_t em_eo_stop (em_eo_t *eo*, int *num_notif*, const em_notif_t * *notif_tbl*)

Stop Execution Object (EO).

Disables event dispatch from all related queues, calls core local stop on all cores and finally calls the global stop function of the EO, when all cores have returned from the (optional) core local stop. Call to the global EO stop is asynchronous and only done, when all cores have completed processing of the receive function and/or core local stop. This guarantees no core is accessing EO data during EO global stop function.

This function returns immediately.

If the caller needs to know when the EO stop was actually completed, the num_notif and notif_tbl can be used. The given notification event(s) will be sent to given queue(s), when the stop actually completes. If such notification is not needed, use 0 as num_notif.

Parameters

eo	EO id
num_notif	How many notification events given, 0 for no notification
notif_tbl	Array of pairs of event and queue identifiers

Returns

EM_OK if successful.

See also

em_stop_func_t(), em_stop_local_func_t(), em_eo_start()

5.1.5.24 em_status_t em_eo_unregister_error_handler (em_eo_t *eo*)

Unregister EO specific error handler.

Removes previously registered EO specific error handler.

Parameters

eo	EO id

Returns

EM_OK if successful.

5.1.5.25 void em_error (em_status_t *error*, em_escope_t *escope*, ...)

Report an error.

Reported errors are handled by the appropriate (EO specific or the global) error handler.

Depending on the error/scope/implementation, the function call may not return.

Parameters

error	Error code
escope	Error scope. Identifies the scope for interpreting the error code and variable arguments.
...	Variable number and type of arguments

See also

em_register_error_handler(), em_error_handler_t()

5.1.5.26 int em_error_format_string (char * *str*, size_t *size*, em_eo_t *eo*, em_status_t *error*, em_escope_t *escope*, va_list *args*)

Format error string

Creates an implementation dependent error report string from EM internal errors.

Parameters

str	Output string pointer
size	Maximum string lenght in characters
eo	EO id
error	Error code (EM internal)
escope	Error scope (EM internal)
args	Variable arguments

Returns

Output string length

5.1.5.27 em_status_t em_event_group_apply (em_event_group_t *group*, int *count*, int *num_notif*, const em_notif_t * *notif_tbl*)

Apply event group configuration.

The function sets (or resets) the event count and notification parameters for the event group. After it returns events sent to the group are counted against the (updated) count. Notification events are sent when all (counted) events have been processed. A new apply call is needed to reset the event group (counting).

Parameters

group	Group id
count	Number of events in the group
num_notif	Number of noticifation events to send
notif_tbl	Table of notifications (events and target queues)

Returns

EM_OK if successful.

See also

em_event_group_create(), em_send_group()

5.1.5.28 em_event_group_t em_event_group_create (void)

Create new event group id for fork-join.

Returns

New event group id or EM_EVENT_GROUP_UNDEF

See also

em_event_group_delete(), em_event_group_apply()

5.1.5.29 em_event_group_t em_event_group_current (void)

Current event group

Returns the event group of the currently received event or EM_EVENT_GROUP_-UNDEF if the current event does not belong into any event group. Current group is needed when sending new events into the group.

Returns

Current event group id or EM_EVENT_GROUP_UNDEF

See also

em_event_group_create()

5.1.5.30 em_status_t em_event_group_delete (em_event_group_t *event_group*)

Delete (unallocate) an event group id

An event group must not be deleted before it has been completed (notifications sent) or canceled.

Parameters

event_group	Event group to delete

Returns

EM_OK if successful.

See also

em_event_group_create()

5.1.5.31 em_status_t em_event_group_increment (int *count*)

Increment event group count

Increments event count of the current event group. Enables sending new events into the current group. Must be called before sending. Note that event count cannot be decremented.

Parameters

count	Number of events to add in the group

Returns

EM_OK if successful.

See also

em_send_group()

5.1.5.32 void* em_event_pointer (const em_event_t *event*)

Get pointer to event structure

Returns pointer to the event structure or NULL. Event structure is implementation and event type specific. It may be a directly accessible buffer of memory, a descriptor containing a list of buffer pointers, a descriptor of a packet buffer, etc.

Parameters

event	Event from receive/alloc

Returns

Event pointer or NULL

5.1.5.33 void em_free (em_event_t *event*)

Free an event.

It is assumed the implementation can detect from which memory area/pool the event was originally allocated from.

Free transfers the ownership of the event to the system and application must not touch the event (or related memory buffers) after calling it.

Application must only free events it owns. For example, sender must not free an event after sending it.

Parameters

event	Event to be freed

See also

em_alloc(), em_receive_func_t()

5.1.5.34 em_event_type_t em_get_type_major (em_event_type_t *type*)

Get major event type.

Event type includes major and minor part. This function returns the major part. It can be compared against enumeration em_event_type_major_e.

Parameters

type	Event type

Returns

Major event type

5.1.5.35 em_event_type_t em_get_type_minor (em_event_type_t *type*)

Get minor event type.

Event type includes major and minor part. This function returns the minor part. It can be compared against the enumeration specified by the major part.

EM_EVENT_TYPE_SW_DEFAULT is reserved for (SW) events that are generic and directly accessible buffers of memory.

Parameters

type	Event type

Returns

Minor event type

5.1.5.36 em_queue_t em_queue_create (const char * *name*, em_queue_type_t *type*, em_queue_prio_t *prio*, em_queue_group_t *group*)

Create a new queue with a dynamic queue id.

The given name string is copied to EM internal data structure. The maximum string length is EM_QUEUE_NAME_LEN.

Parameters

name	Queue name for debugging purposes (optional, NULL ok)
type	Queue scheduling type
prio	Queue priority
group	Queue group for this queue

Returns

New queue id or EM_QUEUE_UNDEF on an error

See also

em_queue_group_create(), em_queue_delete()

5.1.5.37 em_status_t em_queue_create_static (const char * *name*, em_queue_type_t *type*, em_queue_prio_t *prio*, em_queue_group_t *group*, em_queue_t *queue*)

Create a new queue with a static queue id.

Note, that system may have limited amount of static identifiers, so unless really needed use dynamic queues instead. The range of static identifiers is system dependent, but macros EM_QUEUE_STATIC_MIN and EM_QUEUE_STATIC_MAX can be used to abstract, i.e. use EM_QUEUE_STATIC_MIN+x for the application.

The given name string is copied to EM internal data structure. The maximum string length is EM_QUEUE_NAME_LEN.

Parameters

name	Queue name for debugging purposes (optional, NULL ok)
type	Queue scheduling type
prio	Queue priority
group	Queue group for this queue
queue	Requested queue id from the static range

Returns

EM_OK if successful.

See also

em_queue_group_create(), em_queue_delete()

5.1.5.38 em_status_t em_queue_delete (em_queue_t *queue*)

Delete a queue.

Unallocates the queue id. NOTE: this is an immediate deletion and can ∗only∗ be done after the queue has been removed from scheduling using em_eo_remove_queue() !

Parameters

queue	Queue id to delete

Returns

EM_OK if successful.

See also

em_eo_remove_queue(), em_queue_create(), em_queue_create_static()

5.1.5.39 em_status_t em_queue_disable (em_queue_t *queue*, int *num_notif*, const em_notif_t ∗ *notif_tbl*)

Disable scheduling for the queue.

Note, that this might be an asynchronous operation and actually complete later as other cores may still be handling existing events. If application needs to know exactly when all processing is completed, it can use the notification arguments - the given notification(s) are sent after all cores have completed.

Implicit disable is done for all queues, that are mapped to an EO when it's stop-function is called (via em_eo_stop()).

All events sent to a non-enabled queue may get discarded or held depending on the system. Queue enable/disable is not meant to be used for additional scheduling nor used frequently. Main purpose is to synchronize startup or recovery actions.

Parameters

queue	Queue to disable
num_notif	Number of entries in notif_tbl, use 0 for no notification
notif_tbl	Notification events to send

Returns

EM_OK if successful.

See also

em_eo_stop(), em_queue_disable_all(), em_queue_enable()

5.1.5.40 em_status_t em_queue_disable_all (em_eo_t *eo*, int *num_notif*, const em_notif_t * *notif_tbl*)

Disable scheduling for all the EO's queues.

Otherwise identical to em_queue_disable().

Parameters

eo	EO id
num_notif	Number of entries in notif_tbl, use 0 for no notification
notif_tbl	Notification events to send

Returns

EM_OK if successful.

See also

em_eo_stop(), em_queue_disable(), em_queue_enable_all()

5.1.5.41 em_status_t em_queue_enable (em_queue_t *queue*)

Enable event scheduling for the queue.

All events sent to a non-enabled queue may get discarded or held depending on the system. Queue enable/disable is not meant to be used for additional scheduling nor used frequently. Main purpose is to synchronize startup or recovery actions.

Parameters

queue	Queue to enable

Returns

EM_OK if successful.

See also

em_eo_start(), em_queue_enable_all(), em_queue_disable()

5.1.5.42 em_status_t em_queue_enable_all (em_eo_t *eo*)

Enable event scheduling for all the EO's queues.

Otherwise identical to em_queue_enable().

Parameters

eo	EO id

Returns

EM_OK if successful.

See also

em_queue_enable(), em_queue_disable_all()

5.1.5.43 void* em_queue_get_context (em_queue_t *queue*)

Get queue specific (application) context.

Returns the value application has earlier set with em_queue_set_context().

Parameters

queue	Queue which context is requested

Returns

Queue specific context pointer or NULL on an error

See also

em_queue_set_context()

5.1.5.44 em_queue_group_t em_queue_get_group (em_queue_t *queue*)

Get queue's queue group

Parameters

queue	Queue identifier

Returns

Queue group or EM_QUEUE_GROUP_UNDEF on error.

See also

em_queue_create(), em_queue_group_create(), em_queue_group_modify()

5.1.5.45 size_t em_queue_get_name (em_queue_t *queue*, char * *name*, size_t *maxlen*)

Get queue name.

Returns the name given to a queue when it was created. A copy of the queue name string (up to 'maxlen' characters) is written to the user given buffer. String is always null terminated even if the given buffer length is less than the name length.

If the queue has no name, function returns 0 and writes empty string.

This is only for debugging purposes.

Parameters

queue	Queue id
name	Destination buffer
maxlen	Maximum length (including the terminating '0')

Returns

Number of characters written (excludes the terminating '0').

See also

em_queue_create()

5.1.5.46 em_queue_prio_t em_queue_get_priority (em_queue_t *queue*)

Get queue priority.

Parameters

queue	Queue identifier

Returns

Priority class or EM_QUEUE_PRIO_UNDEF on an error

See also

em_queue_create()

5.1.5.47 em_queue_type_t em_queue_get_type (em_queue_t *queue*)

Get queue type (scheduling mode).

Parameters

queue	Queue identifier

Returns

Queue type or EM_QUEUE_TYPE_UNDEF on an error

See also

em_queue_create()

5.1.5.48 em_queue_group_t em_queue_group_create (const char * *name*, const em_core_mask_t * *mask*, int *num_notif*, const em_notif_t * *notif_tbl*)

Create a new queue group to control queue to core mapping.

Allocates a new queue group identifier with a given core mask. The group name can have max EM_QUEUE_GROUP_NAME_LEN characters and must be unique since it's used to identify the group. Cores added to the queue group can be changed later with em_queue_group_modify().

This operation may be asynchronous, i.e. the creation may complete well after this function has returned. Provide notification events, if application cares about the actual completion time. EM will send notifications when the operation has completed.

The core mask is visible through em_queue_group_mask() only after the create operation is complete.

Note, that depending on the system, the operation can also happen one core at a time, so an intermediate mask may be active momentarily.

Only manipulate the core mask with the access macros defined in event_machine_core_mask.h as the implementation underneath may change.

EM has a default group (EM_QUEUE_GROUP_DEFAULT) containing all cores. It's named "default", otherwise naming scheme is system specific.

Note, some systems may have a low number of queue groups available.

Attention

Only call em_queue_enable() after em_queue_group_create() has completed - use notifications to synchronize.

Parameters

name	Queue group name. Unique name for identifying the group.
mask	Core mask for the queue group
num_notif	Number of entries in notif_tbl (use 0 for no notification)
notif_tbl	Array of notifications to send to signal completion of operation

Returns

Queue group or EM_QUEUE_GROUP_UNDEF on error.

See also

em_queue_group_find(), em_queue_group_modify(), em_queue_group_delete()

5.1.5.49 em_status_t em_queue_group_delete (em_queue_group_t *group*, int *num_notif*, const em_notif_t * *notif_tbl*)

Delete the queue group.

Removes all cores from the queue group and free's the identifier for re-use. All queues in the group must be deleted with em_queue_delete() before deleting the group.

Parameters

group	Queue group to delete
num_notif	Number of entries in notif_tbl (use 0 for no notification)
notif_tbl	Array of notifications to send to signal completion of operation

Returns

EM_OK if successful.

See also

em_queue_group_create(), em_queue_group_modify(), em_queue_delete()

5.1.5.50 em_queue_group_t em_queue_group_find (const char * *name*)

Finds queue group by name.

This returns the situation at the moment of the inquiry. If another core is modifying the group at the same time the result may not be up-to-date. Application may need to synchronize group modifications.

Parameters

name	Name of the queue qroup to find

Returns

Queue group or EM_QUEUE_GROUP_UNDEF on an error

See also

em_queue_group_create(), em_queue_group_modify()

5.1.5.51 em_status_t em_queue_group_mask (em_queue_group_t *group*, em_core_mask_t * *mask*)

Get current core mask for a queue group.

This returns the situation at the moment of the inquiry. If another core is modifying the group at the same time the result may not be up-to-date. Application may need to

synchronize group modifications.

Parameters

group	Queue group
mask	Core mask for the queue group

Returns

EM_OK if successful.

See also

em_queue_group_create(), em_queue_group_modify()

5.1.5.52 em_status_t em_queue_group_modify (em_queue_group_t *group*, const em_core_mask_t * *new_mask*, int *num_notif*, const em_notif_t * *notif_tbl*)

Modify core mask of an existing queue group.

The function compares the new core mask to the current mask and changes the queue group to core mapping accordingly.

This operation may be asynchronous, i.e. the change may complete well after this function has returned. Provide notification events, if application cares about the actual completion time. EM will send notifications when the operation has completed.

The new core mask is visible through em_queue_group_mask() only after the modify operation is complete.

Note, that depending on the system, the change can also happen one core at a time, so an intermediate mask may be active momentarily.

Only manipulate core mask with the access macros defined in event_machine_core_-mask.h as the implementation underneath may change.

Parameters

group	Queue group to modify
new_mask	New core mask
num_notif	Number of entries in notif_tbl (use 0 for no notification)
notif_tbl	Array of notifications to send

Returns

EM_OK if successful.

See also

em_queue_group_create(), em_queue_group_find(), em_queue_group_delete(), em_-queue_group_mask()

5.1.5.53 em_status_t em_queue_set_context (em_queue_t *queue*, const void * *context*)

Set queue specific (application) context.

This is just a single pointer associated with a queue. Application can use it to access some context data quickly (without a lookup). The context is given as argument for the receive function. EM does not use the value, it just passes it.

Parameters

queue	Queue to which associate the context
context	Context pointer

Returns

EM_OK if successful.

See also

em_receive_func_t(), em_queue_get_context()

5.1.5.54 em_status_t em_register_error_handler (em_error_handler_t *handler*)

Register the global error handler.

The global error handler is called on errors (or em_error() calls) outside of any EO context or if there's no EO specific error handler registered. Note, the function will override any previously registered global error handler.

Parameters

handler	Error handler.

Returns

EM_OK if successful.

See also

em_eo_register_error_handler(), em_unregister_error_handler(), em_error_handler_t()

5.1.5.55 em_status_t em_send (em_event_t *event*, em_queue_t *queue*)

Send an event to a queue.

Event must have been allocated with em_alloc(), or received via receive-function. Sender must not touch the event after calling em_send as the ownership is moved to system and then to the receiver. If return status is *not* EM_OK, the ownership has not moved and the application is still responsible for the event (e.g. may free it).

EM does not define guaranteed event delivery, i.e. EM_OK return value only means the event was accepted for delivery. It could still be lost during the delivery (e.g. due to disabled/removed queue, queue or system congestion, etc).

Parameters

event	Event to be sent
queue	Destination queue

Returns

EM_OK if successful (accepted for delivery).

See also

em_alloc()

5.1.5.56 em_status_t em_send_group (em_event_t *event*, em_queue_t *queue*, em_event_group_t *group*)

Send event with group number.

Any valid event and destination queue parameters can be used. Event group id indicates which event group the event belongs to. Event group has to be first created and applied.

Parameters

event	Event to send
queue	Destination queue
group	Event group

Returns

EM_OK if successful.

See also

em_send(), em_event_group_create(), em_event_group_apply(), em_event_group_increment()

5.1.5.57 em_status_t em_unregister_error_handler (void)

Unregister the global error handler.

Removes previously registered global error handler.

Returns

EM_OK if successful.

See also

em_register_error_handler()

5.2 TI specific API

Implementation specific declarations.

Data Structures

- struct ti_em_buffer_config_t_
- struct ti_em_chain_config_t_
- struct ti_em_chain_rio_config_t_
- struct ti_em_chain_xge_config_t_
- struct ti_em_config_t_
- struct ti_em_device_rio_route_t_
- struct ti_em_device_xge_route_t_
- struct ti_em_iterator_t_
- struct ti_em_pair_t_
- struct ti_em_pool_config_t_
- struct ti_em_poststore_config_t_
- struct ti_em_preload_config_t_
- struct ti_em_process_route_t_

Defines

- #define TI_EM_AP_PRIVATE_EVENT_NUM (256u)
- #define TI_EM_BUF_MODE_LOOSE (1)
- #define TI_EM_BUF_MODE_TIGHT (0)
- #define TI_EM_BUFFER_POOL_ID_NUM (256u)
- #define TI_EM_CD_PRIVATE_EVENT_NUM (64u)
- #define TI_EM_CHAIN_DISABLED (0u)
- #define TI_EM_CHAIN_ENABLED (1u)
- #define TI_EM_CHAIN_TX_QUEUE_NUM (4u)
- #define TI_EM_CHAINING_PKTDMA (0)
- #define TI_EM_COH_MODE_OFF (0x0)
- #define TI_EM_COH_MODE_ON (0x1)
- #define TI_EM_COH_MODE_RESERVED0 (0x2)
- #define TI_EM_COH_MODE_RESERVED1 (0x3)
- #define TI_EM_CORE_NUM (8u)
- #define TI_EM_DEVICE_NUM (256)
- #define TI_EM_DMA_QUEUE_NUM (TI_EM_CORE_NUM + 2)
- #define TI_EM_EO_NUM_MAX (TI_EM_QUEUE_NUM_MAX)
- #define TI_EM_EVENT_GROUP_NUM_MAX (16384u)
- #define TI_EM_EVENT_TYPE_PRELOAD_MSK (0xC0)
- #define TI_EM_EVENT_TYPE_PRELOAD_OFF (0u)
- #define TI_EM_EVENT_TYPE_PRELOAD_ON_SIZE_A (1u<<6)
- #define TI_EM_EVENT_TYPE_PRELOAD_ON_SIZE_B (2u<<6)
- #define TI_EM_EVENT_TYPE_PRELOAD_ON_SIZE_C (3u<<6)
- #define TI_EM_HW_QUEUE_NUM
- #define TI_EM_HW_QUEUE_STEP (32u)
- #define TI_EM_INTERRUPT_DISABLE (0)
- #define TI_EM_ITERATOR_WSIZE (8u)
- #define TI_EM_PDSP_GLOBAL_DATA_SIZE (8192u)

- #define TI_EM_PF_LEN (1u)
- #define TI_EM_POOL_NUM (32u)
- #define TI_EM_PRELOAD_DISABLED (0u)
- #define TI_EM_PRELOAD_ENABLED (1u)
- #define TI_EM_PRIO_NUM (EM_QUEUE_PRIO_HIGHEST + 1)
- #define TI_EM_PRIVATE_EVENT_DSC_SIZE (16u)
- #define TI_EM_PROCESS_NUM (2u)
- #define TI_EM_PUSH_POLICY_HEAD (0x1)
- #define TI_EM_PUSH_POLICY_TAIL (0x0)
- #define TI_EM_QUEUE_GROUP_NUM_MAX (TI_EM_PRIO_NUM ∗ TI_EM_CORE_NUM)
- #define TI_EM_QUEUE_MODE_HW (1u)
- #define TI_EM_QUEUE_MODE_SD (0u)
- #define TI_EM_QUEUE_NUM_IN_SET (8u)
- #define TI_EM_QUEUE_NUM_MAX (16384u)
- #define TI_EM_QUEUE_SET_NUM (256u)
- #define TI_EM_SCHEDULER_THREAD_NUM (4u)
- #define TI_EM_SLOT_SPCB (0)
- #define TI_EM_STATIC_QUEUE_NUM_IN_SET (2u)
- #define TI_EM_STATIC_QUEUE_NUM_MAX (256u)
- #define TI_EM_STREAM_NUM (256u)
- #define TI_EM_TSCOPE_ALLOC (1)
- #define TI_EM_TSCOPE_ATOMIC_PROCESSING_END (6)
- #define TI_EM_TSCOPE_CLAIM_LOCAL (5)
- #define TI_EM_TSCOPE_DISPATCH (3)
- #define TI_EM_TSCOPE_FREE (7)
- #define TI_EM_TSCOPE_PRESCHEDULE (4)
- #define TI_EM_TSCOPE_SEND (2)
- #define TI_EM_XGE_CHAIN_HEADER_SIZE (8u)
- #define TI_EM_XGE_CRC_SIZE (4)
- #define TI_EM_XGE_ENET_HEADER_SIZE (18u)
- #define TI_EM_XGE_FRAME_SIZE_MIN (64u)
- #define TI_EM_XGE_HEADER_SIZE (TI_EM_XGE_ENET_HEADER_SIZE + TI_EM_XGE_CHAIN_HEADER_SIZE)
- #define TI_EM_XGE_PAYLOAD_SIZE_MIN (TI_EM_XGE_FRAME_SIZE_MIN - TI_EM_XGE_HEADER_SIZE - TI_EM_XGE_CRC_SIZE)
- #define TI_EM_XGE_RX_FRAGMENT_SIZE_MIN (TI_EM_XGE_PAYLOAD_SIZE_MIN + TI_EM_XGE_CRC_SIZE)
- #define TI_EM_XGE_RX_HEADER_SIZE (TI_EM_XGE_HEADER_SIZE)
- #define TI_EM_XGE_RX_MISS_QUEUE_NUM (TI_EM_XGE_VLAN_PRIO_NUM)
- #define TI_EM_XGE_TX_DIVERT_QUEUE_NUM (32u)
- #define TI_EM_XGE_TX_FRAGMENT_SIZE 0
- #define TI_EM_XGE_TX_HEADER_SIZE (TI_EM_XGE_HEADER_SIZE + TI_EM_XGE_PAYLOAD_SIZE_MIN)
- #define TI_EM_XGE_TX_QUEUE_NUM (1u)
- #define TI_EM_XGE_VLAN_PRIO_NUM (8u)

Typedefs

- typedef uint32_t ti_em_buf_mode_t
- typedef uint32_t ti_em_coh_mode_t
- typedef struct ti_em_config_t_ ti_em_config_t
- typedef enum ti_em_counter_type_e ti_em_counter_type_e
- typedef uint32_t ti_em_counter_type_t
- typedef uint32_t ti_em_counter_value_t
- typedef uint32_t ti_em_destination_id_t
- typedef uint8_t ti_em_device_id_t
- typedef uint32_t ti_em_dma_id_t
- typedef uint32_t ti_em_flow_id_t
- typedef void(∗ ti_em_free_func_t)(void ∗buffer_ptr, size_t buffer_size)
- typedef uint8_t ti_em_interrupt_id_t
- typedef uint32_t ti_em_packet_t
- typedef uint8_t ti_em_pdsp_id_t
- typedef uint8_t ti_em_process_id_t
- typedef uint32_t ti_em_process_type_t
- typedef uint32_t ti_em_push_policy_t
- typedef uint16_t ti_em_queue_id_t
- typedef uint8_t ti_em_queue_mode_t
- typedef uint32_t ti_em_sem_id_t
- typedef uint32_t ti_em_stream_id_t
- typedef em_status_t(∗ ti_em_trace_handler_t)(ti_em_tscope_t tscope,...)
- typedef uint32_t ti_em_tscope_t
- typedef enum ti_em_xge_rx_miss_type_e ti_em_xge_rx_miss_type_e
- typedef uint32_t ti_em_xge_rx_miss_type_t

Enumerations

- enum ti_em_counter_type_e
- enum ti_em_xge_rx_miss_type_e

Functions

- em_event_t ti_em_alloc_local (size_t size, em_event_type_t type)
- em_event_t ti_em_alloc_with_buffers (em_event_type_t event_type, em_pool_-id_t pool_id, int32_t buffer_num, ti_em_buffer_config_t ∗buffer_config_tbl)
- uint32_t ti_em_atomic_processing_locality (void)
- size_t ti_em_buffer_size (em_event_t event)
- em_status_t ti_em_chain_rx_flow_open (int dma_idx, int flow_idx, int dst_queue_-idx, int free_queue_idxSizeA, int sizeA, int free_queue_idxSizeB, int sizeB, int free_queue_idxSizeC, int sizeC, int free_queue_idxSizeD, int sizeD, int free_-queue_idxOverflow, int error_handling)
- void ∗ ti_em_claim_local (void)

- em_event_t ti_em_combine (ti_em_pair_t event_pair, ti_em_iterator_t *iterator_ptr)
- em_status_t ti_em_counter_get (ti_em_counter_type_t type, ti_em_counter_value_t *value_ptr, int flag)
- em_status_t ti_em_device_add_rio_route (ti_em_device_id_t device_idx, ti_em_device_rio_route_t device_route)
- em_status_t ti_em_device_add_xge_route (ti_em_device_id_t device_idx, ti_em_device_xge_route_t device_route)
- em_status_t ti_em_dispatch_once (void)
- size_t ti_em_event_size (em_event_t event)
- em_status_t ti_em_exit_global (void)
- void ti_em_flush (em_event_t event)
- em_event_t ti_em_free_with_buffers (em_event_t event, int32_t buffer_num, ti_em_buffer_config_t *buffer_config_tbl)
- em_event_t ti_em_from_packet (ti_em_packet_t *packet_ptr)
- ti_em_queue_id_t ti_em_get_absolute_queue_id (ti_em_dma_id_t dma_idx, ti_em_queue_id_t queue_idx)
- ti_em_buf_mode_t ti_em_get_buf_mode (const em_event_t event)
- ti_em_coh_mode_t ti_em_get_coh_mode (const em_event_t event)
- size_t ti_em_get_eo_size_fast (void)
- size_t ti_em_get_eo_size_slow (void)
- size_t ti_em_get_event_group_size_fast (void)
- size_t ti_em_get_event_group_size_slow (void)
- size_t ti_em_get_pcb_size (void)
- em_status_t ti_em_get_ps_words (em_event_t event, uint32_t *ps_word_ptr, size_t ps_wsize)
- size_t ti_em_get_ps_wsize (em_event_t event)
- size_t ti_em_get_queue_group_size_fast (void)
- size_t ti_em_get_queue_group_size_slow (void)
- size_t ti_em_get_queue_size_fast (void)
- size_t ti_em_get_queue_size_slow (void)
- size_t ti_em_get_tcb_size (void)
- em_event_type_t ti_em_get_type (const em_event_t event)
- em_event_type_t ti_em_get_type_preload (em_event_type_t type)
- em_status_t ti_em_hw_queue_close (int queue_idx)
- em_status_t ti_em_hw_queue_open (int queue_idx)
- em_status_t ti_em_init_global (const ti_em_config_t *config_ptr)
- em_status_t ti_em_init_local (void)
- em_status_t ti_em_interrupt_disable (void)
- em_status_t ti_em_interrupt_enable (void)
- em_status_t ti_em_iterator_next (ti_em_iterator_t *iterator_ptr)
- void * ti_em_iterator_pointer (ti_em_iterator_t *iterator_ptr)
- em_status_t ti_em_iterator_previous (ti_em_iterator_t *iterator_ptr)
- size_t ti_em_iterator_size (ti_em_iterator_t *iterator_ptr)
- em_status_t ti_em_iterator_start (em_event_t event, ti_em_iterator_t *iterator_ptr)
- em_status_t ti_em_iterator_stop (ti_em_iterator_t *iterator_ptr)

- void ti_em_packet_restore_free_info (ti_em_packet_t *packet_ptr)
- void ti_em_packet_set_buffer_info (ti_em_packet_t *packet_ptr, ti_em_buffer_config_t buffer_config)
- void ti_em_packet_set_default (ti_em_packet_t *packet_ptr)
- void ti_em_packet_set_event_group (ti_em_packet_t *packet_ptr, em_event_group_t event_group_hdl)
- void ti_em_packet_set_pool_info (ti_em_packet_t *packet_ptr, ti_em_pool_config_t pool_config)
- void ti_em_packet_set_queue (ti_em_packet_t *packet_ptr, em_queue_t queue_hdl)
- void ti_em_packet_set_type (ti_em_packet_t *packet_ptr, em_event_type_t event_type)
- void ti_em_preschedule (void)
- em_status_t ti_em_process_add_route (ti_em_process_route_t process_route, ti_em_process_id_t process_idx)
- em_queue_t ti_em_queue_create_hw (const char *name, ti_em_queue_id_t hw_queue_idx)
- em_status_t ti_em_queue_create_hw_static (const char *name, ti_em_queue_id_t hw_queue_idx, em_queue_t queue)
- ti_em_device_id_t ti_em_queue_get_device_id (em_queue_t queue_hdl)
- ti_em_queue_mode_t ti_em_queue_get_mode (em_queue_t queue_hdl)
- ti_em_process_id_t ti_em_queue_get_process_id (em_queue_t queue_hdl)
- ti_em_queue_id_t ti_em_queue_get_queue_id (em_queue_t queue_hdl)
- em_queue_t ti_em_queue_make_global (em_queue_t queue_hdl, ti_em_device_id_t device_idx, ti_em_process_id_t process_idx)
- em_event_t ti_em_receive (em_queue_t queue)
- em_status_t ti_em_register_trace_handler (ti_em_trace_handler_t handler)
- em_status_t ti_em_rio_rx_flow_open (int dma_idx, int flow_idx, int dst_queue_idx, int free_queue_idxSizeA, int sizeA, int free_queue_idxSizeB, int sizeB, int free_queue_idxSizeC, int sizeC, int free_queue_idxSizeD, int sizeD, int free_queue_idxOverflow, int deviceIdx, int processIdx, int error_handling)
- em_status_t ti_em_rio_tx_channel_open (int dma_idx)
- ti_em_queue_id_t ti_em_rio_tx_queue_open (int rio_tx_queue_idx)
- em_status_t ti_em_rx_channel_close (int dma_idx, int channel_idx)
- em_status_t ti_em_rx_channel_open (int dma_idx, int channel_idx)
- em_status_t ti_em_rx_flow_close (int dma_idx, int flow_idx)
- em_status_t ti_em_rx_flow_open (int dma_idx, int flow_idx, int dst_queue_idx, int free_queue_idx, int error_handling)
- em_status_t ti_em_set_ps_words (em_event_t event, uint32_t *ps_word_ptr, size_t ps_wsize)
- em_status_t ti_em_set_ps_wsize (em_event_t event, size_t ps_wsize)
- void ti_em_set_queue (em_event_t event, em_queue_t queue_hdl, em_event_group_t event_group_hdl)
- void ti_em_set_type (em_event_t event, em_event_type_t event_type)
- ti_em_pair_t ti_em_split (em_event_t event, ti_em_iterator_t *iterator_ptr)
- void ti_em_tag_set_queue (uint32_t *tag_ptr, em_queue_t queue_hdl)
- void ti_em_tag_set_type (uint32_t *tag_ptr, em_event_type_t event_type)

- ti_em_packet_t * ti_em_to_packet (em_event_t event)
- em_status_t ti_em_tx_channel_close (int dma_idx, int channel_idx)
- em_status_t ti_em_tx_channel_open (int dma_idx, int channel_idx)
- em_status_t ti_em_unregister_trace_handler (void)
- em_status_t ti_em_xge_rx_channel_close (int dma_idx, int channel_idx)
- em_status_t ti_em_xge_rx_channel_open (int dma_idx, int channel_idx)
- em_status_t ti_em_xge_rx_flow_close (int dma_idx, int flow_idx)
- em_status_t ti_em_xge_rx_flow_open (int dma_idx, int flow_idx, int dst_queue_idx, int free_queue_idx0, int free_queue_idx1, int error_handling)
- em_status_t ti_em_xge_rx_miss_disable (ti_em_xge_rx_miss_type_t miss_type)
- em_status_t ti_em_xge_rx_miss_enable (ti_em_xge_rx_miss_type_t miss_type)
- em_status_t ti_em_xge_tx_channel_close (int dma_idx, int channel_idx)
- em_status_t ti_em_xge_tx_channel_open (int dma_idx, int channel_idx)
- ti_em_queue_id_t ti_em_xge_tx_queue_base_idx_get (void)
- ti_em_queue_id_t ti_em_xge_tx_queue_open (ti_em_queue_id_t queue_base_idx, int vlan_priority)

5.2.1 Detailed Description

Implementation specific declarations.

Attention

Unless otherwise specified values of symbolic constants cannot be changed.

5.2.2 Define Documentation

5.2.2.1 #define TI_EM_AP_PRIVATE_EVENT_NUM (256u)

Maximum number of AP private tokens used by the EM

5.2.2.2 #define TI_EM_BUF_MODE_LOOSE (1)

Buffer mode - Loose buffer

5.2.2.3 #define TI_EM_BUF_MODE_TIGHT (0)

Buffer mode - Tight buffer

5.2.2.4 #define TI_EM_BUFFER_POOL_ID_NUM (256u)

Maximum number of pool Id that can be set in the event descriptor pool index field (8 bits)

5.2.2.5 #define TI_EM_CD_PRIVATE_EVENT_NUM (64u)

Maximum number of CD private tokens used by the EM

5.2.2.6 #define TI_EM_CHAIN_DISABLED (0u)

Chaining disabled on Open-EM process.

5.2.2.7 #define TI_EM_CHAIN_ENABLED (1u)

Chaining enabled on Open-EM process.

5.2.2.8 #define TI_EM_CHAIN_TX_QUEUE_NUM (4u)

Maximum number of TX queues for chaining and poststoring. Must be a power of 2: 1, 2, 4 or 8.

5.2.2.9 #define TI_EM_CHAINING_PKTDMA (0)

ID for the various chaining mechanisms (i.e. chaining modules)

5.2.2.10 #define TI_EM_COH_MODE_OFF (0x0)

Cache coherency mode - Off

5.2.2.11 #define TI_EM_COH_MODE_ON (0x1)

Cache coherency mode - On

5.2.2.12 #define TI_EM_COH_MODE_RESERVED0 (0x2)

Cache coherency mode - Reserved0

5.2.2.13 #define TI_EM_COH_MODE_RESERVED1 (0x3)

Cache coherency mode - Reserved1

5.2.2.14 #define TI_EM_CORE_NUM (8u)

Maximum number of cores on a device on which the Event Machine can be executed.

5.2.2.15 #define TI_EM_DEVICE_NUM (256)

Maximum number of devices on which the Open-EM can be executed.

5.2.2.16 #define TI_EM_DMA_QUEUE_NUM (TI_EM_CORE_NUM + 2)

Total number of PKTDMA queues used by the EM

5.2.2.17 #define TI_EM_EO_NUM_MAX (TI_EM_QUEUE_NUM_MAX)

Maximum number of execution objects.

5.2.2.18 #define TI_EM_EVENT_GROUP_NUM_MAX (16384u)

Maximum number of event groups.

5.2.2.19 #define TI_EM_EVENT_TYPE_PRELOAD_MSK (0xC0)

Event type preload mask.

5.2.2.20 #define TI_EM_EVENT_TYPE_PRELOAD_OFF (0u)

Major event types (portable) : Data Preloading Off

Note

Application should always ignore the actual values.

5.2.2.21 #define TI_EM_EVENT_TYPE_PRELOAD_ON_SIZE_A (1u<<6)

Major event types (portable) : Data Preloading up to size A

Note

Application should always ignore the actual values.

5.2.2.22 #define TI_EM_EVENT_TYPE_PRELOAD_ON_SIZE_B (2u<<6)

Major event types (portable) : Data Preloading up to size B

Note

Application should always ignore the actual values.

5.2.2.23 #define TI_EM_EVENT_TYPE_PRELOAD_ON_SIZE_C (3u<<6)

Major event types (portable) : Data Preloading up to size C

Note

Application should always ignore the actual values.

5.2.2.24 #define TI_EM_HW_QUEUE_NUM

Value:

```
(11                         \
                + TI_EM_PRIO_NUM                \
                + (3 * TI_EM_CORE_NUM)            \
                + (3 * TI_EM_CORE_NUM * TI_EM_PRIO_NUM)     \
                + (TI_EM_CORE_NUM * TI_EM_PRIO_NUM * TI_EM_PF_LEN) \
                + TI_EM_XGE_VLAN_PRIO_NUM)
```

Total number of queues used by the EM

5.2.2.25 #define TI_EM_HW_QUEUE_STEP (32u)

Alignment of QM queue base index

5.2.2.26 #define TI_EM_INTERRUPT_DISABLE (0)

Enable/disable PDSP interrupt generation

5.2.2.27 #define TI_EM_ITERATOR_WSIZE (8u)

Number of 32 bits words in an ti_em_iterator_t array.

5.2.2.28 #define TI_EM_PDSP_GLOBAL_DATA_SIZE (8192u)

Multi-core Navigator PDSP memory size to allocated for private events and private firmware data.

5.2.2.29 #define TI_EM_PF_LEN (1u)

Number of PF events per core.

5.2.2.30 #define TI_EM_POOL_NUM (32u)

Maximum number of buffer pools that can be specified.

Note

This value can be modified by an application. But the event machine library needs to be recompiled.

5.2.2.31 #define TI_EM_PRELOAD_DISABLED (0u)

Preload disabled.

5.2.2.32 #define TI_EM_PRELOAD_ENABLED (1u)

Preload enabled.

5.2.2.33 #define TI_EM_PRIO_NUM (EM_QUEUE_PRIO_HIGHEST + 1)

Highest Priority

5.2.2.34 #define TI_EM_PRIVATE_EVENT_DSC_SIZE (16u)

Size for all the private token used by the EM

5.2.2.35 #define TI_EM_PROCESS_NUM (2u)

Maximum number of Open-EM processes on a device on which the Event Machine can be executed.

5.2.2.36 #define TI_EM_PUSH_POLICY_HEAD (0x1)

Push policy to head

5.2.2.37 #define TI_EM_PUSH_POLICY_TAIL (0x0)

Push policy to tail

5.2.2.38 #define TI_EM_QUEUE_GROUP_NUM_MAX (TI_EM_PRIO_NUM * TI_EM_CORE_NUM)

Maximum number of queue groups.

5.2.2.39 #define TI_EM_QUEUE_MODE_HW (1u)

Queue associated with an hardware (HW) queue.

5.2.2.40 #define TI_EM_QUEUE_MODE_SD (0u)

Queue associated with a scheduling (SD) queue.

5.2.2.41 #define TI_EM_QUEUE_NUM_IN_SET (8u)

Maximum number of queues in a set. TI_EM_QUEUE_SET_NUM ∗ TI_EM_QUEUE_-NUM_IN_SET <= TI_EM_QUEUE_NUM_MAX.

5.2.2.42 #define TI_EM_QUEUE_NUM_MAX (16384u)

Maximum number of queues.

5.2.2.43 #define TI_EM_QUEUE_SET_NUM (256u)

Maximum number of queue sets. Must be a power of 2.

5.2.2.44 #define TI_EM_SCHEDULER_THREAD_NUM (4u)

Maximum number of Navigator PDSP instances on which this process scheduler can be executed. Only relevant on a Keystone II target.

5.2.2.45 #define TI_EM_SLOT_SPCB (0)

Definitions of the various PDSP communication memory slots

5.2.2.46 #define TI_EM_STATIC_QUEUE_NUM_IN_SET (2u)

Number of static queues in a set. TI_EM_QUEUE_SET_NUM∗TI_EM_STATIC_-QUEUE_NUM_IN_SET <= TI_EM_STATIC_QUEUE_NUM_MAX

5.2.2.47 #define TI_EM_STATIC_QUEUE_NUM_MAX (256u)

Maximum number of static queues. TI_EM_STATIC_QUEUE_NUM_MAX <= TI_-EM_QUEUE_NUM_MAX.

5.2.2.48 #define TI_EM_STREAM_NUM (256u)

Maximum number of XGE streams.

5.2.2.49 #define TI_EM_TSCOPE_ALLOC (1)

Trace scope for the em_alloc API.

5.2.2.50 #define TI_EM_TSCOPE_ATOMIC_PROCESSING_END (6)

Trace scope for the em_atomic_processing_end API.

5.2.2.51 #define TI_EM_TSCOPE_CLAIM_LOCAL (5)

Trace scope for the ti_em_claim_local API.

5.2.2.52 #define TI_EM_TSCOPE_DISPATCH (3)

Trace scope for the ti_em_dispatch_once API.

5.2.2.53 #define TI_EM_TSCOPE_FREE (7)

Trace scope for the em_free API.

5.2.2.54 #define TI_EM_TSCOPE_PRESCHEDULE (4)

Trace scope for the ti_em_preschedule API.

5.2.2.55 #define TI_EM_TSCOPE_SEND (2)

Trace scope for the em_send API.

5.2.2.56 #define TI_EM_XGE_CHAIN_HEADER_SIZE (8u)

Size (in bytes) of the XGE header.

5.2.2.57 #define TI_EM_XGE_CRC_SIZE (4)

Size (in bytes) of the CRC added by XGE.

5.2.2.58 #define TI_EM_XGE_ENET_HEADER_SIZE (18u)

Size (in bytes) of the XGE header.

5.2.2.59 #define TI_EM_XGE_FRAME_SIZE_MIN (64u)

Minimal size (in bytes) of the XGE frame.

5.2.2.60 #define TI_EM_XGE_HEADER_SIZE (TI_EM_XGE_ENET_HEADER_SIZE + TI_EM_XGE_CHAIN_HEADER_SIZE)

Size (in bytes) of the XGE header (18+8=26).

5.2.2.61 #define TI_EM_XGE_PAYLOAD_SIZE_MIN (TI_EM_XGE_FRAME_SIZE_MIN - TI_EM_XGE_HEADER_SIZE - TI_EM_XGE_CRC_SIZE)

Minimum size (in bytes) of the fragment size (64 - 26 - 4 = 34 Bytes).

5.2.2.62 #define TI_EM_XGE_RX_FRAGMENT_SIZE_MIN (TI_EM_XGE_PAYLOAD_SIZE_MIN + TI_EM_XGE_CRC_SIZE)

Minimal size (in bytes) to be allocated for the XGE RX fragment buffer (34 + 4 = 38).

5.2.2.63 #define TI_EM_XGE_RX_HEADER_SIZE (TI_EM_XGE_HEADER_SIZE)

Size (in bytes) to be allocated for the XGE RX header buffer (26).

5.2.2.64 #define TI_EM_XGE_RX_MISS_QUEUE_NUM (TI_EM_XGE_VLAN_PRIO_NUM)

Number of contiguous RX miss queues that need to be reserved for chaining over XGE

5.2.2.65 #define TI_EM_XGE_TX_DIVERT_QUEUE_NUM (32u)

Number of contiguous TX divert queues that need to be reserved for chaining over XGE

5.2.2.66 #define TI_EM_XGE_TX_FRAGMENT_SIZE 0

Size (in bytes) to be allocated for the XGE TX fragment buffer

5.2.2.67 #define TI_EM_XGE_TX_HEADER_SIZE (TI_EM_XGE_HEADER_SIZE + TI_EM_XGE_PAYLOAD_SIZE_MIN)

Size (in bytes) to be allocated for the XGE TX header buffer (26 + 34 = 60).

5.2.2.68 #define TI_EM_XGE_TX_QUEUE_NUM (1u)

Maximum number of contiguous XGE TX queues

5.2.2.69 #define TI_EM_XGE_VLAN_PRIO_NUM (8u)

Number of VLAN priorities that need to be reserved for chaining over XGE

5.2.3 Typedef Documentation

5.2.3.1 typedef uint32_t ti_em_buf_mode_t

Identifies the event buffer mode.

5.2.3.2 typedef uint32_t ti_em_coh_mode_t

Cache Coherency mode. Identifies the data buffer cache coherency mode.

5.2.3.3 typedef struct ti_em_config_t_ ti_em_config_t

typedef associated with ti_em_config_t_

5.2.3.4 typedef enum ti_em_counter_type_e ti_em_counter_type_e

The structure describes the counters addressed by the ti_em_counter_get() api.

5.2.3.5 typedef uint32_t ti_em_counter_type_t

Counter type. Identifies the counter type.

5.2.3.6 typedef uint32_t ti_em_counter_value_t

Counter value. Identifies the counter value.

5.2.3.7 typedef uint32_t ti_em_destination_id_t

Destination id. Identifies the destination for a route. It may be a ti_em_process_id_t or a ti_em_device_id_t.

5.2.3.8 typedef uint8_t ti_em_device_id_t

Device id. Identifies the device.

5.2.3.9 typedef uint32_t ti_em_dma_id_t

DMA id. Identifies the DMA.

5.2.3.10 typedef uint32_t ti_em_flow_id_t

flow id. Identifies the flow.

5.2.3.11 typedef void(* ti_em_free_func_t)(void *buffer_ptr, size_t buffer_size)

Free funtion handler.

Parameters

buffer_ptr	buffer pointer
buffer_size	buffer size

5.2.3.12 typedef uint8_t ti_em_interrupt_id_t

Identifies the Interrupt index.

5.2.3.13 typedef uint32_t ti_em_packet_t

Packet. Identifies the packet of the event.

5.2.3.14 typedef uint8_t ti_em_pdsp_id_t

Pdsp id. Identifies the Pdsp.

5.2.3.15 typedef uint8_t ti_em_process_id_t

Openem process id. Identifies the openem process.

5.2.3.16 typedef uint32_t ti_em_process_type_t

Openem process type. Identifies the openem process type.

5.2.3.17 typedef uint32_t ti_em_push_policy_t

Identifies the push policy.

5.2.3.18 typedef uint16_t ti_em_queue_id_t

Event queue id. Identifies the event queue.

5.2.3.19 typedef uint8_t ti_em_queue_mode_t

Identifies the Queue Mode.

5.2.3.20 typedef uint32_t ti_em_sem_id_t

Hardware semaphore id. Identifies the hardware semaphore.

5.2.3.21 typedef uint32_t ti_em_stream_id_t

Stream id. Identifies the stream.

5.2.3.22 typedef em_status_t(* ti_em_trace_handler_t)(ti_em_tscope_t tscope,...)

Trace function handler.

Parameters

tscope	Error scope. Identifies the scope for interpreting the error code and variable arguments
args	Variable number and type of arguments

Returns

The function may not return depending on implementation.

See also

em_register_trace_handler(), em_unregister_trace_handler()

5.2.3.23 typedef uint32_t ti_em_tscope_t

Trace scope. Identifies the scope for interpreting trace codes and variable arguments

5.2.3.24 typedef enum ti_em_xge_rx_miss_type_e ti_em_xge_rx_miss_type_e

The structure describes the XGE chain header miss types checked by the XGE router.

5.2.3.25 typedef uint32_t ti_em_xge_rx_miss_type_t

Rx miss type. Identifies the rx miss type associated with the chaining over xge.

5.2.4 Enumeration Type Documentation

5.2.4.1 enum ti_em_counter_type_e

The structure describes the counters addressed by the ti_em_counter_get() api.

Enumerator:

ti_em_counter_type_XGE_RECEIVE XGE router counter counting the received fragments.

ti_em_counter_type_XGE_WRONG_ETHER_TYPE XGE router counter counting the received fragments with wrong ether type.

ti_em_counter_type_XGE_WRONG_MAC XGE router counter counting the received fragments with wrong MAC address.

ti_em_counter_type_XGE_WRONG_PROCESS XGE router counter counting the received fragments with wrong process index.

ti_em_counter_type_XGE_WRONG_SEQUENCE XGE router counter counting the received fragments with wrong sequence index.

ti_em_counter_type_XGE_WRONG_SERVICE_TYPE XGE router counter counting the received fragments with wrong service type.

5.2.4.2 enum ti_em_xge_rx_miss_type_e

The structure describes the XGE chain header miss types checked by the XGE router.

Enumerator:

ti_em_xge_rx_miss_type_ETHER_TYPE XGE chain header ETHER_TYPE check.

ti_em_xge_rx_miss_type_SERVICE_TYPE XGE chain header SERVICE_TYPE check.

5.2.5 Function Documentation

5.2.5.1 em_event_t ti_em_alloc_local (size_t *size*, em_event_type_t *type*)

Allocate a local event (for post-storing feature).

Memory address of the allocated event is system specific and can depend on given pool id, event size and type. Returned event (handle) may refer to a memory buffer or a HW specific descriptor, i.e. the event structure is system specific.

Use em_event_pointer() to convert an event (handle) to a pointer to the event structure.

EM_EVENT_TYPE_SW with minor type 0 is reserved for direct portability. It is always guaranteed to return a 64-bit aligned contiguous data buffer, that can directly be used by the application up to the given size (no HW specific descriptors etc are visible).

Parameters

size	Event size in octets
type	Event type to allocate

Returns

the allocated event or EM_EVENT_UNDEF on an error

See also

em_free(), em_send(), em_event_pointer(), em_receive_func_t()

5.2.5.2 em_event_t ti_em_alloc_with_buffers (em_event_type_t *event_type*, em_pool_id_t *pool_id*, int32_t *buffer_num*, ti_em_buffer_config_t * *buffer_config_tbl*)

Allocates an event while providing buffer(s) to attach.

It allocates a primary event from pools with zero-buffer descriptors.

It attaches one or more buffers to the event.

The event comes with optional free function and coherency mode for each buffer.

Parameters

event_type	Event type to allocate
pool_id	Event pool id
buffer_num	Event buffers number
buffer_-config_tbl	Data buffers configuration table

Returns

Event handle

See also

ti_em_free_with_buffers()

5.2.5.3 uint32_t ti_em_atomic_processing_locality (void)

Indicates that the atomic processing locality has been achieved by the scheduler.

Returns

Hint if the atomic processing locality has been achieved.

5.2.5.4 size_t ti_em_buffer_size (em_event_t *event*)

Returns the buffer size.

Returns the size of the event attached data buffer.

Parameters

event	Event handle.

Returns

Size of the attached buffer.

5.2.5.5 em_status_t ti_em_chain_rx_flow_open (int *dma_idx*, int *flow_idx*, int *dst_queue_idx*, int *free_queue_idxSizeA*, int *sizeA*, int *free_queue_idxSizeB*, int *sizeB*, int *free_queue_idxSizeC*, int *sizeC*, int *free_queue_idxSizeD*, int *sizeD*, int *free_queue_idxOverflow*, int *error_handling*)

Opens an hard-ware process chaining receive flow on the data-base and configure the hard-ware registers of this receive flow.

Precondition

The hard-ware receive flow shall not be used by another part of the application on the device.
If the hard-ware receive flow is already used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the Cppi_configureRxFlow PDK function.

Parameters

dma_idx	Index of the dma instance. This refers to the Multicore Navigator PktDMA instance.
flow_idx	Index of the receive flow to be open. It ranges from 0 to the maximum Rx Flow index supported by the Multicore Navigator instance.
dst_queue_idx	Index of the destination queue.
free_queue_idxSizeA	Index of the destination free queue for packet length smaller than sizeA. Queue indexes range from 0 to the maximum queue index supported by the Multicore Navigator instance.
sizeA	Maximal size in bytes of the packet length for free queue A
free_queue_idxSizeB	Index of the destination free queue for packet length smaller than sizeB. Queue indexes range from 0 to the maximum queue index supported by the Multicore Navigator instance.
sizeB	Maximal size in bytes of the packet length for free queue B
free_queue_idxSizeC	Index of the destination free queue for packet length smaller than sizeC. Queue indexes range from 0 to the maximum queue index supported by the Multicore Navigator instance.
sizeC	Maximal size in bytes of the packet length for free queue C
free_queue_idxSizeD	Index of the destination free queue for packet length smaller than sizeD. Queue indexes range from 0 to the maximum queue index supported by the Multicore Navigator instance.
sizeD	Maximal size in bytes of the packet length for free queue D
free_queue_idxOverflow	Index of the destination free queue for packet length with overflow. Queue indexes range from 0 to the maximum queue index supported by the Multicore Navigator instance.
error_handling	Receive flow error handling mode when starvation occurs. 0 = Starvation errors result in dropping packet. 1 = Starvation errors result in subsequent re-try.

Postcondition

On success, the RX_FLOW_CONFIG registers shall be correctly configured.

Returns

status, EM_OK on success

5.2.5.6 void* ti_em_claim_local (void)

The function returns the local event buffer pointer for a local event or NULL if there is no local event.

Returns

Pointer to the event buffer.

5.2.5.7 em_event_t ti_em_combine (ti_em_pair_t *event_pair*, ti_em_iterator_t * *iterator_ptr*)

Combines two (2) event handles into one scattered event handle.

When returned, the iterator buffer points to last descriptor of the old pair.head_event.

Parameters

event_pair	pair to combine.
iterator_ptr	Pointer to iterator.

Returns

event handle.

5.2.5.8 em_status_t ti_em_counter_get (ti_em_counter_type_t *type*, ti_em_counter_value_t * *value_ptr*, int *flag*)

Returns the value of the counter specified by the "type" parameter. If the "flag" parameter is set to 1, the counter is read and then reset to 0 in the API call. The returned counter value is written at the "value_ptr" address.

Parameters

type	type
flag	reset flag; if set to 1, the counter is reset to 0.
value_ptr	pointer to the variable where to write the counter value.

Returns

status, EM_OK on success

5.2.5.9 em_status_t ti_em_device_add_rio_route (ti_em_device_id_t *device_idx*, ti_em_device_rio_route_t *device_route*)

Adds an SRIO route to the device router.

Parameters

device_idx	Device index.
device_route	Device route.

Returns

status, EM_OK on success

5.2.5.10 em_status_t ti_em_device_add_xge_route (ti_em_device_id_t *device_idx*, ti_em_device_xge_route_t *device_route*)

Adds an XGE route to the device router.

Parameters

device_idx	Device index.
device_route	Device route.

Returns

status, EM_OK on success

5.2.5.11 em_status_t ti_em_dispatch_once (void)

The function runs the dispatcher once.

The dispatcher code is responsible for calling a execution object receive API. It checks once if a event has been scheduled to its core and calls the receive function.

Returns

returns EM_ERR_NOT_FOUND if no receive function was executed. returns EM_OK when a public event was executed.

5.2.5.12 size_t ti_em_event_size (em_event_t *event*)

Returns the event payload size.

Returns the size of the event attached data buffers.

Parameters

event	Event handle.

Returns

Size of the attached buffers.

5.2.5.13 em_status_t ti_em_exit_global (void)

Global shutdown of EM internals.

Only one core does this and after this call, no other call is allowed.

Returns

status, EM_OK on success

5.2.5.14 void ti_em_flush (em_event_t *event*)

Loops over all data buffers of the event and perform a cache write back invalidate if the data buffer is dirty.

Parameters

event	Event handle.

5.2.5.15 em_event_t ti_em_free_with_buffers (em_event_t *event*, int32_t *buffer_num*, ti_em_buffer_config_t * *buffer_config_tbl*)

Frees an event with buffers.

It returns an event to its free pool(s).

It doesn't call free function, even if provided.

It fills out the configuration for each buffer that has been attached with ti_em_alloc_-with_buffers().

It returns the number of such buffers - error if too many buffer pointers to return (wrt to buffer_num).

Parameters

event	Event to be freed
buffer_num	Event buffers number
buffer_-config_tbl	Data buffers configuration table

Returns

Event handle.

See also

ti_em_alloc_with_buffers()

5.2.5.16 em_event_t ti_em_from_packet (ti_em_packet_t * *packet_ptr*)

Helper function that converts into an event handle from a packet.

Parameters

packet_ptr	Pointer to the packet.

Returns

Event handle.

5.2.5.17 ti_em_queue_id_t ti_em_get_absolute_queue_id (ti_em_dma_id_t *dma_idx*, ti_em_queue_id_t *queue_idx*)

Retrieves the absolute queue index from the associated Multicore Navigator PkDMA engine index and the relative TX queue index.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the CPPI LLD.

Parameters

dma_idx	Index of the dma instance. This refers to the Multicore Navigator PktDMA instance.
queue_idx	Relative index of the queue.

Returns

Absolute index of the queue.

5.2.5.18 ti_em_buf_mode_t ti_em_get_buf_mode (const em_event_t *event*)

Helper function which returns the buffer mode.

Parameters

event	Event handle.

Returns

Buffer mode of the event.

5.2.5.19 ti_em_coh_mode_t ti_em_get_coh_mode (const em_event_t *event*)

Helper function which returns the cache coherency mode.

Parameters

event	Event handle.

Returns

Cache coherency mode of the event.

5.2.5.20 size_t ti_em_get_eo_size_fast (void)

The function returns the needed size for the fast part of the EO descriptor in bytes. It is stored in the .tiEmGlobalFast memory section preferably located in non cached MSMC RAM or DDR3 RAM.

Returns

size for the fast part of a EO structure

5.2.5.21 size_t ti_em_get_eo_size_slow (void)

The function returns the needed size for the slow part of the EO descriptor in bytes. It is stored in the .tiEmGlobalSlow memory section preferably located in non cached MSMC RAM or DDR3 RAM.

Returns

size for the slow part of a EO structure

5.2.5.22 size_t ti_em_get_event_group_size_fast (void)

The function returns the needed size for the fast part of the event group descriptor in bytes. It is stored in the .tiEmGlobalFast memory section preferably located in non cached MSMC RAM or DDR3 RAM.

Returns

size for the fast part of a event group structure

5.2.5.23 size_t ti_em_get_event_group_size_slow (void)

The function returns the needed size for the slow part of the event group descriptor in bytes. It is stored in the .tiEmGlobalSlow memory section preferably located in non cached MSMC RAM or DDR3 RAM.

Returns

size for the slow part of a event group structure

5.2.5.24 size_t ti_em_get_pcb_size (void)

The function returns the needed size for the master control block plus the runtime master control block in bytes. It is stored in the .tiEmGlobalFast memory section.

Returns

size for the Master Control Block structure

5.2.5.25 em_status_t ti_em_get_ps_words (em_event_t *event*, uint32_t * *ps_word_ptr*, size_t *ps_wsize*)

Helper function which gets the ps words from the event.

Parameters

event	Event handle.
ps_word_ptr	Pointer to ps words.
ps_wsize	Number of ps words.

Returns

status, EM_OK on success

5.2.5.26 size_t ti_em_get_ps_wsize (em_event_t *event*)

Helper function which gets the number of ps words from the event.

Parameters

event	Event handle.

Returns

Number of ps words

5.2.5.27 size_t ti_em_get_queue_group_size_fast (void)

The function returns the needed size for the fast part of the queue group descriptor in bytes. It is stored in the .tiEmGlobalFast memory section preferably located in non cached MSMC RAM or DDR3 RAM.

Returns

size for the fast part of a queue group structure

5.2.5.28 size_t ti_em_get_queue_group_size_slow (void)

The function returns the needed size for the slow part of the queue group descriptor in bytes. It is stored in the .tiEmGlobalSlow memory section preferably located in non

cached MSMC RAM or DDR3 RAM.

Returns

size for the slow part of a queue group structure

5.2.5.29 size_t ti_em_get_queue_size_fast (void)

The function returns the needed size for the fast part of the queue descriptor in bytes. It is stored in the .tiEmGlobalFast memory section preferably located in non cached MSMC RAM or DDR3 RAM.

Returns

size for the fast part of a queue structure

5.2.5.30 size_t ti_em_get_queue_size_slow (void)

The function returns the needed size for the slow part of the queue descriptor in bytes. It is stored in the .tiEmGlobalSlow memory section preferably located in non cached MSMC RAM or DDR3 RAM.

Returns

size for the slow part of a queue structure

5.2.5.31 size_t ti_em_get_tcb_size (void)

The function returns the needed size for the dispatcher control block in bytes. It is stored in the .tiEmLobal memory section.

Returns

size for the Dispatcher Control Block structure

5.2.5.32 em_event_type_t ti_em_get_type (const em_event_t *event*)

Helper function which returns the event type.

Parameters

event	Event handle.

Returns

Event type of the descriptor.

5.2.5.33 em_event_type_t ti_em_get_type_preload (em_event_type_t *type*)

Helper function which extracts the preload policy from the event type.

Parameters

type	Event type.

Returns

Preload policy.

5.2.5.34 em_status_t ti_em_hw_queue_close (int *queue_idx*)

Closes an hard-ware queue on the data-base.

Precondition

The hard-ware queue shall already be used by the Open-Em process.
If the hard-ware queue is not used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the Qmss_queueClose PDK function.

Parameters

queue_idx	Index of the queue to be open. It ranges from 0 to the maximum queue index supported by the Multicore Navigator.

Postcondition

None.

Returns

status, EM_OK on success.

5.2.5.35 em_status_t ti_em_hw_queue_open (int *queue_idx*)

Opens an hard-ware queue on the data-base.

Precondition

The hard-ware queue shall not be used by another part of the application on the device.
If the hard-ware queue is already used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c

file using the Qmss_queueOpen PDK function.

Parameters

queue_idx	Index of the queue to be open. It ranges from 0 to the maximum queue index supported by the Multicore Navigator.

Postcondition

None.

Returns

status, EM_OK on success.

5.2.5.36 em_status_t ti_em_init_global (const ti_em_config_t * *config_ptr*)

Global initialization of EM internals.

Only one core does this and this must be called before any other call.

Parameters

config_ptr	Pointer to the hardware configuration structure.

Returns

status, EM_OK on success

5.2.5.37 em_status_t ti_em_init_local (void)

Local initialization of EM internals.

All cores call this and it must be called after em_init_global(), but before any other call. Implementation may be actually empty, but this might be needed later for some core specific initializations, so application startup should call this always.

Returns

status, EM_OK on success

5.2.5.38 em_status_t ti_em_interrupt_disable (void)

Disables the interrupt generation for the attached dispatcher.

Returns

status, EM_OK on success

5.2.5.39 em_status_t ti_em_interrupt_enable (void)

Enables the interrupt generation for the attached dispatcher. The interrupt channel index used is based on the configuration hw_queue_base_idx and the physical core index. If the interrupt channel index is inferior to 32, it is mapped to one of the 32 HI channels in the associated INTD register. Otherwise it is mapped to one of the 16 LOW channels. By default, the interrupt are not enabled.

Returns

status, EM_OK on success

5.2.5.40 em_status_t ti_em_iterator_next (ti_em_iterator_t * *iterator_ptr*)

Moves the iterator to the next buffer.

Parameters

iterator_ptr	Pointer to iterator.

Returns

returns EM_OK when successfull, EM_ERR_BAD_STATE when iterator is in a wrong state. EM_ERR_NOT_FOUND when the iterator points to the last buffer.

5.2.5.41 void* ti_em_iterator_pointer (ti_em_iterator_t * *iterator_ptr*)

Returns a pointer to the iterator attached data buffer.

Parameters

iterator_ptr	Pointer to iterator.

Returns

Pointer to the buffer.

5.2.5.42 em_status_t ti_em_iterator_previous (ti_em_iterator_t * *iterator_ptr*)

Moves the iterator to the previous buffer.

Parameters

iterator_ptr	Pointer to iterator.

Returns

returns EM_OK when successfull, EM_ERR_BAD_STATE when iterator is in a wrong state. EM_ERR_NOT_FOUND when the iterator points to the first buffer.

5.2.5.43 size_t ti_em_iterator_size (ti_em_iterator_t * *iterator_ptr*)

Returns the size of the iterator attached data buffer.

Parameters

iterator_ptr	Pointer to iterator.

Returns

Size of the attached buffer.

5.2.5.44 em_status_t ti_em_iterator_start (em_event_t *event*, ti_em_iterator_t * *iterator_ptr*)

Initializes the iterator to the first buffer. Two iterators may not be started at the same time for a single event.

Parameters

event	Event handle to start.
iterator_ptr	Pointer to iterator.

Returns

returns EM_OK.

5.2.5.45 em_status_t ti_em_iterator_stop (ti_em_iterator_t * *iterator_ptr*)

It registers all the touched buffers.

It is mandatory to perform ti_em_iterator_stop() before em_free(), em_send() or em_event_group_apply().

It is forbidden to use the iterator or any iterator content from this point onwards.

Parameters

iterator_ptr	Pointer to iterator.

Returns

returns EM_OK when successfull, EM_ERR_BAD_STATE when iterator is in a wrong state.

5.2.5.46 void ti_em_packet_restore_free_info (ti_em_packet_t * *packet_ptr*)

Helper function which restores the free information in the packet.

Parameters

packet_ptr	Pointer to the packet.

5.2.5.47 void ti_em_packet_set_buffer_info (ti_em_packet_t * *packet_ptr*, ti_em_buffer_config_t *buffer_config*)

Helper function which sets the buffer information in the packet.

Parameters

packet_ptr	Pointer to the packet.
buffer_config	Structure containing the buffer info.

5.2.5.48 void ti_em_packet_set_default (ti_em_packet_t * *packet_ptr*)

Helper function which sets the 12 first words of an event to their default values.

The CPPI type is set to Host, The EPIB flag is ON and the PS location flag is ON.

Parameters

packet_ptr	Pointer to the packet.

5.2.5.49 void ti_em_packet_set_event_group (ti_em_packet_t * *packet_ptr*, em_event_group_t *event_group_hdl*)

Helper function which sets the event group information in the packet.

Parameters

packet_ptr	Pointer to the packet.
event_group_hdl	Event group handle to set.

5.2.5.50 void ti_em_packet_set_pool_info (ti_em_packet_t * *packet_ptr*, ti_em_pool_config_t *pool_config*)

Helper function which sets the event pool information in the packet.

Parameters

packet_ptr	Pointer to the packet.
pool_config	Structure containing the pool info.

5.2.5.51 void ti_em_packet_set_queue (ti_em_packet_t * *packet_ptr*, em_queue_t *queue_hdl*)

Helper function which sets the queue information in the packet.

Parameters

packet_ptr	Pointer to the packet.
queue_hdl	Queue handle to set.

5.2.5.52 void ti_em_packet_set_type (ti_em_packet_t * *packet_ptr*, em_event_type_t *event_type*)

Helper function which sets the event type in the packet.

Parameters

packet_ptr	Pointer to the packet.
event_type	Event type of the packet.

5.2.5.53 void ti_em_preschedule (void)

Indicates that the scheduler can pre-schedule events to this core.

5.2.5.54 em_status_t ti_em_process_add_route (ti_em_process_route_t *process_route*, ti_em_process_id_t *process_idx*)

Adds a process route to the device router.

Parameters

process_route	Parameters of the process route.
process_idx	Process index.

Returns

status, EM_OK on success

5.2.5.55 em_queue_t ti_em_queue_create_hw (const char * *name*, ti_em_queue_id_t *hw_queue_idx*)

Creates a HW queue.

Parameters

name	Queue name for debugging purposes (optional, NULL ok)
hw_queue_idx	Hw queue index

Returns

new queue id or EM_QUEUE_UNDEF on an error

5.2.5.56 em_status_t ti_em_queue_create_hw_static (const char * *name*, ti_em_queue_id_t *hw_queue_idx*, em_queue_t *queue*)

Creates a static HW queue.

Parameters

name	Queue name for debugging purposes (optional, NULL ok)
hw_queue_idx	Hw queue index
queue	Requested queue id from the static range

Returns

EM_OK on success or EM_QUEUE_UNDEF on an error

5.2.5.57 ti_em_device_id_t ti_em_queue_get_device_id (em_queue_t *queue_hdl*)

Helper function which extracts the device index from the queue handle.

Parameters

queue_hdl	Queue handle.

Returns

Device index.

5.2.5.58 ti_em_queue_mode_t ti_em_queue_get_mode (em_queue_t *queue_hdl*)

Helper function which extracts the queue mode (HW or SD) from the queue handle.

Parameters

queue_hdl	Queue handle.

Returns

Queue mode.

5.2.5.59 ti_em_process_id_t ti_em_queue_get_process_id (em_queue_t *queue_hdl*)

Helper function which extracts the process index from the queue handle.

Parameters

queue_hdl	Queue handle.

Returns

Process index.

5.2.5.60 ti_em_queue_id_t ti_em_queue_get_queue_id (em_queue_t *queue_hdl*)

Helper function which returns the index of the associated Multicore Navigator queue.

Parameters

queue_hdl	Queue handle.

Returns

Device index.

5.2.5.61 em_queue_t ti_em_queue_make_global (em_queue_t *queue_hdl*, ti_em_device_id_t *device_idx*, ti_em_process_id_t *process_idx*)

Helper function which generates a global queue handle.

Parameters

queue_hdl	Local queue handle.
device_idx	Device index.
process_idx	Process index.

Returns

Global queue handle.

5.2.5.62 em_event_t ti_em_receive (em_queue_t *queue*)

Receives an event from a queue.

Event must have been allocated with em_alloc().

Parameters

queue	Receiving queue

Returns

the received event or EM_EVENT_UNDEF if the queue is empty.

See also

em_alloc()

5.2.5.63 em_status_t ti_em_register_trace_handler (ti_em_trace_handler_t *handler*)

Registers the trace handler.

Parameters

handler	Trace handler.

Returns

Success code.

5.2.5.64 em_status_t ti_em_rio_rx_flow_open (int *dma_idx*, int *flow_idx*, int *dst_queue_idx*, int *free_queue_idxSizeA*, int *sizeA*, int *free_queue_idxSizeB*, int *sizeB*, int *free_queue_idxSizeC*, int *sizeC*, int *free_queue_idxSizeD*, int *sizeD*, int *free_queue_idxOverflow*, int *deviceIdx*, int *processIdx*, int *error_handling*)

Opens an hard-ware process RIO receive flow on the data-base and configure the hard-ware registers of this receive flow. It also creates the mapping between the Type9 message parameters and the opened flow.

Precondition

The hard-ware receive flow shall not be used by another part of the application on the device.
If the hard-ware receive flow is already used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_-hal.c file using the Cppi_configureRxFlow PDK function and the CSL_SRIO_-MapType9MessageToQueue CSL API.

Parameters

dma_idx	Index of the dma instance. This refers to the Multicore Navigator PktDMA instance.
flow_idx	Index of the receive flow to be open. It ranges from 0 to the maximum Rx Flow index supported by the Multicore Navigator instance.
dst_queue_-idx	Index of the destination queue.
free_-queue_-idxSizeA	Index of the destination free queue for packet length smaller than sizeA. Queue indexes range from 0 to the maximum queue index supported by the Multicore Navigator instance.
sizeA	Maximal size in bytes of the packet length for free queue A
free_-queue_-idxSizeB	Index of the destination free queue for packet length smaller than sizeB. Queue indexes range from 0 to the maximum queue index supported by the Multicore Navigator instance.
sizeB	Maximal size in bytes of the packet length for free queue B
free_-queue_-idxSizeC	Index of the destination free queue for packet length smaller than sizeC. Queue indexes range from 0 to the maximum queue index supported by the Multicore Navigator instance.
sizeC	Maximal size in bytes of the packet length for free queue C
free_-queue_-idxSizeD	Index of the destination free queue for packet length smaller than sizeD. Queue indexes range from 0 to the maximum queue index supported by the Multicore Navigator instance.
sizeD	Maximal size in bytes of the packet length for free queue D

free_queue_idxOverflow	Index of the destination free queue for packet length with overflow. Queue indexes range from 0 to the maximum queue index supported by the Multicore Navigator instance.
deviceIdx	Index of the current device
processIdx	Index of the current process
error_handling	Receive flow error handling mode when starvation occurs. 0 = Starvation errors result in dropping packet. 1 = Starvation errors result in subsequent re-try.

Postcondition

On success, the RX_FLOW_CONFIG registers shall be correctly configured.

Returns

status, EM_OK on success

5.2.5.65 em_status_t ti_em_rio_tx_channel_open (int *dma_idx*)

Opens an hard-ware SRIO transmit channel on the data-base and configure the hard-ware registers of this transmit channel.

Precondition

The hard-ware transmit channel shall not be used by another part of the application on the device.
If the hard-ware transmit channel is already used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the Cppi_txChannelOpen PDK function.

Parameters

dma_idx	Index of the dma instance. This refers to the Multicore Navigator PktDMA instance.

Postcondition

On success, the TX_CHANNEL_GLOBAL_CONFIG_REG_A register shall be enabled.
The TX_CHANNEL_SCHEDULER_CONFIG_REG_PRIORITY register shall be configured to the desired value.

Returns

status, EM_OK on success

5.2.5.66 ti_em_queue_id_t ti_em_rio_tx_queue_open (int *rio_tx_queue_idx*)

Opens an hard-ware SRIO transmit queue on the data-base and configure the hard-ware registers of this transmit queue.

Precondition

The hard-ware transmit queue shall not be used by another part of the application on the device.
If the hard-ware transmit queue is already used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the ti_em_hw_queue_open function.

Parameters

rio_tx_queue_idx	RIO queue index

Returns

index of the RIO transmit queue or error

5.2.5.67 em_status_t ti_em_rx_channel_close (int *dma_idx,* int *channel_idx*)

Closes an hard-ware receive channel on the data-base.

Precondition

The hard-ware receive channel shall be used by the Open-EM process.
If the hard-ware receive channel is not used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the Cppi_channelClose PDK function.

Parameters

dma_idx	Index of the dma instance. This refers to the Multicore Navigator PktDMA instance.
channel_idx	Index of the receive channel to be closed. It ranges from 0 to the maximum Rx Channel index supported by the Multicore Navigator instance.

Returns

status, EM_OK on success

5.2.5.68 em_status_t ti_em_rx_channel_open (int *dma_idx*, int *channel_idx*)

Opens an hard-ware receive channel on the data-base and configure the hard-ware registers of this receive channel.

Precondition

The hard-ware receive channel shall not be used by another part of the application on the device.
If the hard-ware receive channel is already used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the Cppi_rxChannelOpen PDK function.

Parameters

dma_idx	Index of the dma instance. This refers to the Multicore Navigator PktDMA instance.
channel_idx	Index of the receive channel to be open. It ranges from 0 to the maximum Rx Channel index supported by the Multicore Navigator instance.

Postcondition

On success, the RX_CHANNEL_GLOBAL_CONFIG_REG register shall be enabled.

Returns

status, EM_OK on success

5.2.5.69 em_status_t ti_em_rx_flow_close (int *dma_idx*, int *flow_idx*)

Closes an hard-ware receive flow on the data-base.

Precondition

The hard-ware receive flow shall be used by the Open-EM process.
If the hard-ware receive flow is not used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the Cppi_closeRxFlow PDK function.

Parameters

dma_idx	Index of the dma instance. This refers to the Multicore Navigator PktDMA instance.
flow_idx	Index of the receive flow to be closed. It ranges from 0 to the maximum Rx Flow index supported by the Multicore Navigator instance.

Returns

status, EM_OK on success

5.2.5.70 em_status_t ti_em_rx_flow_open (int *dma_idx*, int *flow_idx*, int *dst_queue_idx*, int *free_queue_idx*, int *error_handling*)

Opens an hard-ware receive flow on the data-base and configure the hard-ware registers of this receive flow.

Precondition

The hard-ware receive flow shall not be used by another part of the application on the device.
If the hard-ware receive flow is already used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the Cppi_configureRxFlow PDK function.

Parameters

dma_idx	Index of the dma instance. This refers to the Multicore Navigator PktDMA instance.
flow_idx	Index of the receive flow to be open. It ranges from 0 to the maximum Rx Flow index supported by the Multicore Navigator instance.
dst_queue_idx	Index of the destination queue.
free_queue_idx	Index of the destination free queue. Queue indexes range from 0 to the maximum queue index supported by the Multicore Navigator instance.
error_handling	Receive flow error handling mode when starvation occurs. 0 = Starvation errors result in dropping packet. 1 = Starvation errors result in subsequent re-try.

Postcondition

On success, the RX_FLOW_CONFIG registers shall be correctly configured.

Returns

status, EM_OK on success

5.2.5.71 em_status_t ti_em_set_ps_words (em_event_t *event*, uint32_t * *ps_word_ptr*, size_t *ps_wsize*)

Helper function which sets the ps words in the event.

Parameters

event	Event handle.
ps_word_ptr	Pointer to ps words.
ps_wsize	Number of ps words.

Returns

status, EM_OK on success

5.2.5.72 em_status_t ti_em_set_ps_wsize (em_event_t *event*, size_t *ps_wsize*)

Helper function which sets the number of ps words in the event.

Parameters

event	Event handle.
ps_wsize	Number of ps words.

Returns

status, EM_OK on success

5.2.5.73 void ti_em_set_queue (em_event_t *event*, em_queue_t *queue_hdl*, em_event_group_t *event_group_hdl*)

Helper function which sets the queue and event group information in the event.

Parameters

event	Event handle.
queue_hdl	Queue handle to set.
event_group_hdl	Event group handle to set.

5.2.5.74 void ti_em_set_type (em_event_t *event*, em_event_type_t *event_type*)

Helper function which sets the event type in the event.

Parameters

event	Event handle.
event_type	Event type to set.

5.2.5.75 ti_em_pair_t ti_em_split (em_event_t *event*, ti_em_iterator_t * *iterator_ptr*)

Splits one scattered event into two events.

when returned, the iterator buffer points to the last buffer of pair.head_event (can be recombined right away).

Second and following buffers of pair.tail_event are cache coherent.

Parameters

event	Event to split.
iterator_ptr	Pointer to iterator where the split starts.

Returns

Events pair.

5.2.5.76 void ti_em_tag_set_queue (uint32_t * *tag_ptr,* em_queue_t *queue_hdl*)

Helper function which sets the queue information in the event tag location.

Parameters

tag_ptr	Pointer to tag location.
queue_hdl	Queue handle to set.

5.2.5.77 void ti_em_tag_set_type (uint32_t * *tag_ptr,* em_event_type_t *event_type*)

Helper function which sets the event type in the tag location.

Parameters

tag_ptr	Pointer to tag location.
event_type	Event type to set.

5.2.5.78 ti_em_packet_t* ti_em_to_packet (em_event_t *event*)

Helper function that converts an event handle into a packet.

Parameters

event	Event handle.

Returns

Pointer to the packet.

5.2.5.79 em_status_t ti_em_tx_channel_close (int *dma_idx,* int *channel_idx*)

Closes an hard-ware transmit channel on the data-base.

Precondition

The hard-ware transmit channel shall be used by the Open-EM process.
If the hard-ware transmit channel is not used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the Cppi_channelClose PDK function.

Parameters

dma_idx	Index of the dma instance. This refers to the Multicore Navigator PktDMA instance.
channel_idx	Index of the transmit channel to be closed. It ranges from 0 to the maximum Tx Channel index supported by the Multicore Navigator instance.

Returns

status, EM_OK on success

5.2.5.80 em_status_t ti_em_tx_channel_open (int *dma_idx*, int *channel_idx*)

Opens an hard-ware transmit channel on the data-base and configure the hard-ware registers of this transmit channel.

Precondition

The hard-ware transmit channel shall not be used by another part of the application on the device.
If the hard-ware transmit channel is already used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the Cppi_txChannelOpen PDK function.

Parameters

dma_idx	Index of the dma instance. This refers to the Multicore Navigator PktDMA instance.
channel_idx	Index of the transmit channel to be open. It ranges from 0 to the maximum Tx Channel index supported by the Multicore Navigator instance.

Postcondition

On success, the TX_CHANNEL_GLOBAL_CONFIG_REG_A register shall be enabled.
The TX_CHANNEL_SCHEDULER_CONFIG_REG_PRIORITY register shall be configured to the desired value.

Returns

status, EM_OK on success

5.2.5.81 em_status_t ti_em_unregister_trace_handler (void)

Unregisters the trace handler.

Returns

Success code is 0,

5.2.5.82 em_status_t ti_em_xge_rx_channel_close (int *dma_idx,* int *channel_idx*)

Closes an hard-ware XGE receive channel on the data-base.

Precondition

The hard-ware receive channel shall be used by the Open-EM process.
If the hard-ware receive channel is not used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the Cppi_channelClose PDK function.

Parameters

dma_idx	Index of the dma instance. This refers to the Multicore Navigator PktDMA instance.
channel_idx	Index of the receive channel to be closed. It ranges from 0 to the maximum Rx Channel index supported by the Multicore Navigator instance.

Returns

status, EM_OK on success

5.2.5.83 em_status_t ti_em_xge_rx_channel_open (int *dma_idx,* int *channel_idx*)

Opens an hard-ware xge receive channel on the data-base and configure the hard-ware registers of this receive channel.

Precondition

The hard-ware receive channel shall not be used by another part of the application on the device.
If the hard-ware receive channel is already used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the Cppi_rxChannelOpen PDK function.

Parameters

dma_idx	Index of the dma instance. This refers to the Multicore Navigator PktDMA instance.
channel_idx	Index of the receive channel to be open. It ranges from 0 to the maximum Rx Channel index supported by the Multicore Navigator instance.

Postcondition

On success, the RX_CHANNEL_GLOBAL_CONFIG_REG register shall be enabled.

Returns

status, EM_OK on success

5.2.5.84 em_status_t ti_em_xge_rx_flow_close (int *dma_idx*, int *flow_idx*)

Closes an hard-ware XGE receive flow on the data-base.

Precondition

The hard-ware receive flow shall be used by the Open-EM process.
If the hard-ware receive flow is not used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the Cppi_closeRxFlow PDK function.

Parameters

dma_idx	Index of the dma instance. This refers to the Multicore Navigator PktDMA instance.
flow_idx	Index of the receive flow to be closed. It ranges from 0 to the maximum Rx Flow index supported by the Multicore Navigator instance.

Returns

status, EM_OK on success

5.2.5.85 em_status_t ti_em_xge_rx_flow_open (int *dma_idx*, int *flow_idx*, int *dst_queue_idx*, int *free_queue_idx0*, int *free_queue_idx1*, int *error_handling*)

Opens an hard-ware xge receive flow on the data-base and configure the hard-ware registers of this receive flow.

Precondition

The hard-ware receive flow shall not be used by another part of the application on the device.

If the hard-ware receive flow is already used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the Cppi_configureRxFlow PDK function.

Parameters

dma_idx	Index of the dma instance. This refers to the Multicore Navigator PktDMA instance.
flow_idx	Index of the receive flow to be open. It ranges from 0 to the maximum Rx Flow index supported by the Multicore Navigator instance.
dst_queue_idx	Index of the destination queue.
free_queue_idx0	Index of the destination free queue. Queue indexes range from 0 to the maximum queue index supported by the Multicore Navigator instance.
free_queue_idx1	Index of the destination free queue. Queue indexes range from 0 to the maximum queue index supported by the Multicore Navigator instance.
error_handling	Receive flow error handling mode when starvation occurs. 0 = Starvation errors result in dropping packet. 1 = Starvation errors result in subsequent re-try.

Postcondition

On success, the RX_FLOW_CONFIG registers shall be correctly configured.

Returns

status, EM_OK on success

5.2.5.86 em_status_t ti_em_xge_rx_miss_disable (ti_em_xge_rx_miss_type_t *miss_type*)

Disable miss redirection for Ethertype/service type.

Parameters

miss_type	miss type disabled.

Returns

status, EM_OK on success

5.2.5.87 em_status_t ti_em_xge_rx_miss_enable (ti_em_xge_rx_miss_type_t *miss_type*)

Enable miss redirection for Ethertype/service type.

Parameters

miss_type	miss type enabled.

Returns

status, EM_OK on success

5.2.5.88 em_status_t ti_em_xge_tx_channel_close (int *dma_idx*, int *channel_idx*)

Closes an hard-ware XGE transmit channel on the data-base.

Precondition

The hard-ware transmit channel shall be used by the Open-EM process.
If the hard-ware transmit channel is not used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the Cppi_channelClose PDK function.

Parameters

dma_idx	Index of the dma instance. This refers to the Multicore Navigator PktDMA instance.
channel_idx	Index of the transmit channel to be closed. It ranges from 0 to the maximum Tx Channel index supported by the Multicore Navigator instance.

Returns

status, EM_OK on success

5.2.5.89 em_status_t ti_em_xge_tx_channel_open (int *dma_idx*, int *channel_idx*)

Opens an hard-ware xge transmit channel on the data-base and configure the hard-ware registers of this transmit channel.

Precondition

The hard-ware transmit channel shall not be used by another part of the application on the device.
If the hard-ware transmit channel is already used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the Cppi_txChannelOpen PDK function.

Parameters

dma_idx	Index of the dma instance. This refers to the Multicore Navigator PktDMA instance.
channel_idx	Index of the transmit channel to be open. It ranges from 0 to the maximum Tx Channel index supported by the Multicore Navigator instance.

Postcondition

On success, the TX_CHANNEL_GLOBAL_CONFIG_REG_A register shall be enabled.
The TX_CHANNEL_SCHEDULER_CONFIG_REG_PRIORITY register shall be configured to the desired value.

Returns

status, EM_OK on success

5.2.5.90 ti_em_queue_id_t ti_em_xge_tx_queue_base_idx_get (void)

Returns the first hard-ware xge transmit queue among TI_EM_XGE_VLAN_PRIO_-NUM (8).

Precondition

The hard-ware transmit queue shall not be used by another part of the application on the device.
If the hard-ware transmit queue is already used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the ti_em_xge_tx_queue_base_idx_get function.

Parameters

.	

Returns

index of the first XGE transmit queue

5.2.5.91 ti_em_queue_id_t ti_em_xge_tx_queue_open (ti_em_queue_id_t *queue_base_idx*, int *vlan_priority*)

Opens an hard-ware xge transmit queue on the data-base and configure the hard-ware registers of this transmit queue.

Precondition

The hard-ware transmit queue shall not be used by another part of the application on the device.

If the hard-ware transmit queue is already used, the function is not successful.

Remarks

The implementation is left to the user. An example is provided in em_pdk_hal.c file using the ti_em_hw_queue_open function.

Parameters

queue_base_idx	Index of the 1st XGE tx queue
vlan_priority	VLAN priority.

Returns

index of the XGE transmit queue

Chapter 6

Data Structure Documentation

6.1 em_notif_t Struct Reference

Data Fields

- em_event_t event
- em_queue_t queue

6.1.1 Detailed Description

Notification structure allows user to define a notification event and destination queue pair. EM will notify user by sending the event into the queue.

6.1.2 Field Documentation

6.1.2.1 em_event_t em_notif_t::event

User defined notification event

6.1.2.2 em_queue_t em_notif_t::queue

Destination queue

6.2 ti_em_buffer_config_t_ Struct Reference

Data Fields

- void * buffer_ptr
- size_t buffer_size

- ti_em_coh_mode_t coh_mode
- ti_em_free_func_t free_func
- ti_em_push_policy_t free_push_policy
- uint32_t orig_buffer_pool
- void * orig_buffer_ptr
- size_t orig_buffer_size

6.2.1 Detailed Description

Buffer parameters.

This describes the buffer configurable parameters which are required.

6.2.2 Field Documentation

6.2.2.1 void* ti_em_buffer_config_t_::buffer_ptr

Buffer pointer

6.2.2.2 size_t ti_em_buffer_config_t_::buffer_size

Buffer size

6.2.2.3 ti_em_coh_mode_t ti_em_buffer_config_t_::coh_mode

Buffer coherency mode

6.2.2.4 ti_em_free_func_t ti_em_buffer_config_t_::free_func

Buffer free function

6.2.2.5 ti_em_push_policy_t ti_em_buffer_config_t_::free_push_policy

Event machine pool free push policy

6.2.2.6 uint32_t ti_em_buffer_config_t_::orig_buffer_pool

Original buffer pool id

6.2.2.7 void* ti_em_buffer_config_t_::orig_buffer_ptr

Original buffer pointer

6.2.2.8 size_t ti_em_buffer_config_t_::orig_buffer_size

Original buffer size

6.3 ti_em_chain_config_t_ Struct Reference

Data Fields

- ti_em_dma_id_t dma_idx
- ti_em_queue_id_t dma_tx_queue_base_idx
- em_pool_id_t pool_idx_overflow
- em_pool_id_t pool_idx_size_a
- em_pool_id_t pool_idx_size_b
- em_pool_id_t pool_idx_size_c
- em_pool_id_t pool_idx_size_d
- ti_em_poststore_config_t * poststore_config_ptr
- ti_em_chain_rio_config_t * rio_config_ptr
- ti_em_chain_xge_config_t * xge_config_ptr

6.3.1 Detailed Description

Event machine chaining configuration.

This describes the EM configurable parameters which are required for the chaining.

6.3.2 Field Documentation

6.3.2.1 ti_em_dma_id_t ti_em_chain_config_t_::dma_idx

Index for Packet DMA flows configuration. It has to map to a Multicore Navigator infrastructure Packet DMA.

6.3.2.2 ti_em_queue_id_t ti_em_chain_config_t_::dma_tx_queue_base_idx

Identifies the first relative DMA TX queue index used for the chaining flows.

6.3.2.3 em_pool_id_t ti_em_chain_config_t_::pool_idx_overflow

Identifies the pool used for allocating even bigger events (size D <= event size). The received event uses one buffer (first) from the pool 'pool_idx_size_d'. The received event uses one or more buffers (second and next) from the pool 'pool_idx_overflow'.

6.3.2.4 em_pool_id_t ti_em_chain_config_t_::pool_idx_size_a

Identifies the pool used for allocating very small events (event size <= size A). Size A corresponds to the buffer size of the pool. The received event uses one buffer from the pool.

6.3.2.5 em_pool_id_t ti_em_chain_config_t_::pool_idx_size_b

Identifies the pool used for allocating small events (size A <= event size <= size B). Size B corresponds to the buffer size of the pool. The received event uses one buffer from the pool.

6.3.2.6 em_pool_id_t ti_em_chain_config_t_::pool_idx_size_c

Identifies the pool used for allocating big events (size B <= event size <= size C). Size C corresponds to the buffer size of the pool. The received event uses one buffer from the pool.

6.3.2.7 em_pool_id_t ti_em_chain_config_t_::pool_idx_size_d

Identifies the pool used for allocating very big events (size C <= event size <= size D). Size D corresponds to the buffer size of the pool. The received event uses one buffer from the pool.

6.3.2.8 ti_em_poststore_config_t* ti_em_chain_config_t_::poststore_config_ptr

Pointer to the post-storing configuration.

6.3.2.9 ti_em_chain_rio_config_t* ti_em_chain_config_t_::rio_config_ptr

Pointer to the SRIO Configuration.

6.3.2.10 ti_em_chain_xge_config_t* ti_em_chain_config_t_::xge_config_ptr

Pointer to the 10 Gigabit Configuration.

6.4 ti_em_chain_rio_config_t_ Struct Reference

Data Fields

- ti_em_dma_id_t dma_idx
- ti_em_queue_id_t dma_tx_queue_idx
- uint8_t my_node_idx [2]
- uint8_t service_class

6.4.1 Detailed Description

Event machine chaining configuration for SRIO.

This describes the EM configurable parameters which are required for the chaining over SRIO.

6.4.2 Field Documentation

6.4.2.1 ti_em_dma_id_t ti_em_chain_rio_config_t_::dma_idx

Index for Packet DMA flows configuration. It has to map to a Multicore Navigator infrastructure Packet DMA.

6.4.2.2 ti_em_queue_id_t ti_em_chain_rio_config_t_::dma_tx_queue_idx

Index for SRIO tx queue.

6.4.2.3 uint8_t ti_em_chain_rio_config_t_::my_node_idx[2]

Specifies the SRIO node index

6.4.2.4 uint8_t ti_em_chain_rio_config_t_::service_class

Specifies the Service Class

6.5 ti_em_chain_xge_config_t_ Struct Reference

Data Fields

- ti_em_dma_id_t dma_idx
- uint8_t ether_type [2]
- uint8_t my_mac_address [6]
- ti_em_pdsp_id_t pdsp_idx
- ti_em_queue_id_t rx_fragment_free_queue_idx
- ti_em_queue_id_t rx_header_free_queue_idx
- ti_em_queue_id_t rx_miss_queue_base_idx
- uint8_t service_type
- ti_em_queue_id_t tx_divert_queue_base_idx
- ti_em_queue_id_t tx_fragment_free_queue_idx
- ti_em_queue_id_t tx_header_free_queue_idx
- uint8_t vlan_prio_mask

6.5.1 Detailed Description

Event machine chaining configuration for xge.

This describes the EM configurable parameters which are required for the chaining over xge.

6.5.2 Field Documentation

6.5.2.1 ti_em_dma_id_t ti_em_chain_xge_config_t_::dma_idx

Index for Packet DMA flows configuration. It has to map to a Multicore Navigator infrastructure Packet DMA.

6.5.2.2 uint8_t ti_em_chain_xge_config_t_::ether_type[2]

Specifies the Ethernet type.

6.5.2.3 uint8_t ti_em_chain_xge_config_t_::my_mac_address[6]

Specifies the MAC address of the device on which this OpenEM instance is running.

6.5.2.4 ti_em_pdsp_id_t ti_em_chain_xge_config_t_::pdsp_idx

Identifies a PDSP running the chaining firmware.

6.5.2.5 ti_em_queue_id_t ti_em_chain_xge_config_t_::rx_fragment_free_queue_idx

Identifies the free queue for RX fragments.

6.5.2.6 ti_em_queue_id_t ti_em_chain_xge_config_t_::rx_header_free_queue_idx

Identifies the free queue for RX headers.

6.5.2.7 ti_em_queue_id_t ti_em_chain_xge_config_t_::rx_miss_queue_base_idx

Identifies the first of TI_EM_XGE_VLAN_PRIO_NUM XGE RX miss queues

6.5.2.8 uint8_t ti_em_chain_xge_config_t_::service_type

Specifies the service type.

6.5.2.9 ti_em_queue_id_t ti_em_chain_xge_config_t_::tx_divert_queue_base_idx

Identifies the base index for the OpenEM instance to allocate the XGE TX divert queue.

6.5.2.10 ti_em_queue_id_t ti_em_chain_xge_config_t_::tx_fragment_free_queue_idx

Identifies the free queue for TX fragments (without buffers).

6.5.2.11 ti_em_queue_id_t ti_em_chain_xge_config_t_::tx_header_free_queue_idx

Identifies the free queue for TX headers

6.5.2.12 uint8_t ti_em_chain_xge_config_t_::vlan_prio_mask

Specifies the VLAN priority Mask.

6.6 ti_em_config_t_ Struct Reference

Data Fields

- ti_em_queue_id_t ap_region_queue_idx
- ti_em_queue_id_t cd_region_queue_idx
- ti_em_chain_config_t * chain_config_ptr
- ti_em_dma_id_t dma_idx
- uint16_t dma_queue_base_idx
- ti_em_interrupt_id_t hw_interrupt_base_idx
- uint16_t hw_queue_base_idx
- uint32_t hw_sem_idx
- ti_em_device_id_t my_device_idx

  Identifies the device on which this OpenEM instance is running.

- ti_em_process_id_t my_process_idx

  Identifies the process in which this OpenEM instance is running.

- ti_em_pdsp_id_t pdsp_idx_tbl [TI_EM_SCHEDULER_THREAD_NUM]
- uint8_t pdsp_num
- ti_em_pool_config_t pool_config_tbl [TI_EM_POOL_NUM]
- uint32_t pool_num
- ti_em_preload_config_t * preload_config_ptr

6.6.1 Detailed Description

Event Machine hardware configuration.

This describes the EM configurable parameters which are required.

6.6.2 Field Documentation

6.6.2.1 ti_em_queue_id_t ti_em_config_t_::ap_region_queue_idx

Hardware queue containing the private events used for Atomic Processing. This queue shall contain at least 256 private events.

6.6.2.2 ti_em_queue_id_t ti_em_config_t_::cd_region_queue_idx

Hardware queue containing the private events used for Command Processing. This queue shall contain at least 32 private events. This queue may be the same as ap_-region_queue_idx.

6.6.2.3 ti_em_chain_config_t* ti_em_config_t_::chain_config_ptr

Pointer to the chaining configuration.

6.6.2.4 ti_em_dma_id_t ti_em_config_t_::dma_idx

Index for Packet DMA flows configuration. It has to map to a Multicore Navigator infrastructure Packet DMA.

6.6.2.5 uint16_t ti_em_config_t_::dma_queue_base_idx

Index of the first DMA queue that will be used by the TI EM.

6.6.2.6 ti_em_interrupt_id_t ti_em_config_t_::hw_interrupt_base_idx

Identifies the interrupt base index on which the scheduler will notify dispatching.

6.6.2.7 uint16_t ti_em_config_t_::hw_queue_base_idx

Index of the 1st hardware queue that will be used by the TI EM.

6.6.2.8 uint32_t ti_em_config_t_::hw_sem_idx

Hardware Semaphore used to protect internal data structures.

6.6.2.9 ti_em_pdsp_id_t ti_em_config_t_::pdsp_idx_tbl[TI_EM_SCHEDULER_THREAD_NUM]

Indexes of the pdsp threads running the FW scheduler.

6.6.2.10 uint8_t ti_em_config_t_::pdsp_num

Number of pdsp instances running the FW scheduler.

6.6.2.11 ti_em_pool_config_t ti_em_config_t_::pool_config_tbl[TI_EM_POOL_NUM]

pool configuration descriptors.

6.6.2.12 uint32_t ti_em_config_t_::pool_num

Number of pools

6.6.2.13 ti_em_preload_config_t* ti_em_config_t_::preload_config_ptr

Pointer to the preload configuration.

6.7 ti_em_device_rio_route_t_ Struct Reference

Data Fields

- uint8_t node_idx [2]

6.7.1 Detailed Description

...
This describes the parameters required to configure a route over the Serial Rapid IO.

6.7.2 Field Documentation

6.7.2.1 uint8_t ti_em_device_rio_route_t_::node_idx[2]

Specifies the node index.

6.8 ti_em_device_xge_route_t_ Struct Reference

Data Fields

- size_t fragmentSize
- uint8_t mac_address [6]
- uint8_t vlan_tag [4]

6.8.1 Detailed Description

...

This describes the parameters required to configure a route over the 10 Gigabit Ethernet (XGE).

6.8.2 Field Documentation

6.8.2.1 size_t ti_em_device_xge_route_t_::fragmentSize

Specifies the size of an event fragment.

6.8.2.2 uint8_t ti_em_device_xge_route_t_::mac_address[6]

Specifies the MAC address.

6.8.2.3 uint8_t ti_em_device_xge_route_t_::vlan_tag[4]

Specifies the VLAN tag. Must match the VLAN priority .

6.9 ti_em_iterator_t_ Struct Reference

Data Fields

- uint32_t body [TI_EM_ITERATOR_WSIZE]

6.9.1 Detailed Description

Identifies the iterator type.

This describes the array of iterator fields.

6.9.2 Field Documentation

6.9.2.1 uint32_t ti_em_iterator_t_::body[TI_EM_ITERATOR_WSIZE]

array of iterator fields

6.10 ti_em_pair_t_ Struct Reference

Data Fields

- em_event_t head_event_hdl
- em_event_t tail_event_hdl

6.10.1 Detailed Description

Event handle pair.

This describes the event handle pair when combining/splitting events.

6.10.2 Field Documentation

6.10.2.1 em_event_t ti_em_pair_t_::head_event_hdl

Event machine head event in the pair.

6.10.2.2 em_event_t ti_em_pair_t_::tail_event_hdl

Event machine tail event in the pair.

6.11 ti_em_pool_config_t_ Struct Reference

Data Fields

- ti_em_buf_mode_t buf_mode
- size_t buf_size
- size_t dsc_wsize
- uint16_t free_queue_idx

6.11.1 Detailed Description

Data buffer pool configuration.

This describes the data buffer pool configurable parameters which are required.

6.11.2 Field Documentation

6.11.2.1 ti_em_buf_mode_t ti_em_pool_config_t_::buf_mode

Event machine pool buffer mode

6.11.2.2 size_t ti_em_pool_config_t_::buf_size

Number of bytes used for storing a buffer

6.11.2.3 size_t ti_em_pool_config_t_::dsc_wsize

Number of words used for storing a descriptor

6.11.2.4 uint16_t ti_em_pool_config_t_::free_queue_idx

Event machine pool free queues

6.12 ti_em_poststore_config_t_ Struct Reference

Data Fields

- ti_em_queue_id_t local_free_queue_idx_tbl [2 ∗TI_EM_CORE_NUM]
- void ∗ local_heap_ptr_tbl [2 ∗TI_EM_CORE_NUM]
- size_t poststore_size_max

6.12.1 Detailed Description

Event machine post-storing configuration.

This describes the EM configurable parameters which are required for the post-storing.

6.12.2 Field Documentation

6.12.2.1 ti_em_queue_id_t ti_em_poststore_config_t_::local_free_queue_idx_tbl[2 ∗TI_EM_CORE_NUM]

Identifies the post-storing free queues. Array entry allocations per core are (core index ∗ 2) + 0 and (core index ∗ 2) + 1.

6.12.2.2 void∗ ti_em_poststore_config_t_::local_heap_ptr_tbl[2 ∗TI_EM_CORE_NUM]

Identifies the post-storing heap pointers. Array entry allocations per core are (core index ∗ 2) + 0 and (core index ∗ 2) + 1.

6.12.2.3 size_t ti_em_poststore_config_t_::poststore_size_max

Specifies the maximum number of bytes to post-store.

6.13 ti_em_preload_config_t_ Struct Reference

Data Fields

- ti_em_queue_id_t local_free_queue_idx_tbl [TI_EM_CORE_NUM]
- size_t preload_size_a
- size_t preload_size_b
- size_t preload_size_c

6.13.1 Detailed Description

Event machine preload configuration.

This describes the EM configurable parameters which are required for the preload.

6.13.2 Field Documentation

6.13.2.1 ti_em_queue_id_t ti_em_preload_config_t_::local_free_queue_idx_tbl[TI_EM_CORE_NUM]

Identifies the preload free queues.

6.13.2.2 size_t ti_em_preload_config_t_::preload_size_a

Specifies the maximum number of bytes to preload if event type matches TI_EM_EVENT_TYPE_PRELOAD_ON_SIZE_A. Must be more than 0 and less than or equal to 8Kbytes.

6.13.2.3 size_t ti_em_preload_config_t_::preload_size_b

Specifies the maximum number of bytes to preload if event type matches TI_EM_EVENT_TYPE_PRELOAD_ON_SIZE_B. Must be more than 0 and less than or equal to 8Kbytes.

6.13.2.4 size_t ti_em_preload_config_t_::preload_size_c

Specifies the maximum number of bytes to preload if event type matches TI_EM_EVENT_TYPE_PRELOAD_ON_SIZE_B. Must be more than 0 and less than or equal to 1Mbytes.

6.14 ti_em_process_route_t_ Struct Reference

Data Fields

- ti_em_dma_id_t dma_idx
- ti_em_queue_id_t dma_tx_queue_idx

6.14.1 Detailed Description

Event handle pair.

This describes the parameters required to configure a route to another process.

6.14.2 Field Documentation

6.14.2.1 ti_em_dma_id_t ti_em_process_route_t_::dma_idx

Index for Packet DMA flows configuration. It has to map to infrastructure Packet DMA.

6.14.2.2 ti_em_queue_id_t ti_em_process_route_t_::dma_tx_queue_idx

Identifies the relative DMA TX queue index used for the chaining flow.