RivieraWaves Kernel

Functional Specifications

RW-KERNEL-SW-FS

Version 1.1

2012-05-23

RivieraWaves
http://www.rivierawaves.com/

RivieraWaves confidential. This document is copyrighted and released under CDA or NDA only. Do not copy or distribute without written authorization from RivieraWaves. Please check with RivieraWaves that this document is the latest release.

Revision History

Version	Date	Revision description	Author
1.00	2010-03-01	Initial Release	SBA
1.01	2012-05-23	Updated	SBA

1 Table of contents

2 List of Tables

3 Overview

3.1 Feature List

RivieraWaves Kernel is a small and efficient Real Time Operating System, offering the following features:

- Exchange of messages
- Message saving
- Timer functionality
- The kernel also provides an event functionality used to defer actions

3.2 Source File Structure

File	Description
ke_config.h	Contains all the constants that can be changed in order to tailor the kernel.
ke_env.c, .h	Contains the kernel environment definition.
ke_event.c; .h	Contains the event handling primitives.
ke_mem.c; .h	Implementation of the heap management module.
ke_msg.c, .h	This file contains the scheduler primitives called to create or delete a task. It contains also the scheduler itself.
ke_queue.c; .h	Contains all the functions that handle the different queues (timer queue, save queue, user queue)
ke_task.c; .h	Contains the implementation of the kernel task management.
ke_timer.c, .h	Contains the scheduler primitives called to create or delete a timer task. It contains also the timer scheduler itself.

Table 3.1 : Kernel files List

3.3 Include Files

In order to use the services offered by the kernel the user should include the following files:

- ke_task.h
- ke_timer.h

3.4 Kernel Environment

The kernel environment structure contains the three queues used for the event, timer and message management.

- **evt_field:** Queue of sent messages but not yet delivered to receiver
- **queue_sent:** Queue of sent messages but not yet delivered to receiver.
- **queue_saved:** Queue of messages delivered but not consumed by receiver.
- **queue_timer:** Queue of timer.
- **mblock_first:** Pointer to first element of linked list.

If kernel profiling is enabled, the following fields are added:

- **max_heap_used:** Maximum heap memory used by the kernel.
- **queue_timer:** Queue of messages delivered but not consumed by receiver

4 Messages

4.1 Overview

Message queues provide a mechanism to transmit one or more message to a task. Two queues are defined:

- **queue_sent:** Queue of sent messages but not yet delivered to receiver
- **queue_saved:** Queue of messages delivered but not consumed by receiver

Transmission of messages is done in 3 steps:

- Allocation of a message structure by the sender task
- Filling of the message parameters
- Message structure pushed in the kernel

A message is identified by a unique ID composed of the task type and an increasing number. The following macro builds the first message ID of a task:

#define KE_FIRST_MSG(task) ((ke_msg_id_t)((task) << 10))

A message has a list of parameters that is defined in a structure (see chapter **Message Object**).

4.2 Message Object

The structure of the message contains:

- **hdr:** List header for chaining
- **hci_type:** Type of HCI data (used by the HCI only)
- **hci_off:** Offset of the HCI data in the message (used by the HCI only)
- **hci_len:** Length of the HCI traffic (used by the HCI only)
- **id:** Message identifier.
- **dest_id:** Destination kernel identifier.
- **src_id**: Source kernel identifier.
- **param_len:** Parameter embedded structure length.
- **param:** Parameter embedded structure. Must be word-aligned.

4.3 Message Identifier

- Message Identifier is defined as follow:

  typedef uint16_t ke_msg_id_t;

- The message identifier should be defined by task type in one file only to avoid multiple identical definitions.
  - In xx_task.h for XX task.

4.4 Parameter Management

- During message allocation, the size of the parameter is passed and memory is allocated in the kernel heap. In order to store this data, the pointer on the parameters is returned. The scheduler frees this memory after the transition completion.

```
void *ke_msg_alloc(ke_msg_id_t const id,
                   ke_task_id_t const dest_id,
                   ke_task_id_t const src_id,
                   uint16_t const param_len)
{
   struct ke_msg *msg = (struct ke_msg*) ke_malloc(sizeof(struct ke_msg) +
                                        param_len - sizeof (uint32_t));
...
   return param_ptr;
}
```

4.5 Message Queue Object

A Message queue is defined as a linked list composed of message element:

- ***first:** pointer to first element of the list
- ***last:** pointer to the last element

If kernel profiling is enable those following field are added:

- **cnt:** number of element in the list
- **maxcnt:** max number of element in the list
- **mincnt:** min number of element in the list

4.6 Message Queue Primitives

4.6.1 Message Allocation

Prototype:

```
void *ke_msg_alloc(ke_msg_id_t const id,
                   ke_task_id_t const dest_id,
                   ke_task_id_t const src_id,
                   uint16_t const param_len)
```

Parameters:

Type	Parameters	Description
ke_msg_id_t	id	Message Identifier
ke_task_id_t	dest_id	Destination Task Identifier
ke_task_id_t	src_id	Source Task Identifier
uint16_t	param_len	Length of parameter

Return: Pointer to the parameter member of the ke_msg. If the parameter structure is empty, the pointer will point to the end of the message and should not be used (except to retrieve the message pointer or to send the message).

Description:

This primitive allocates memory for a message that has to be sent. The memory is allocated dynamically on the heap and the length of the variable parameter structure has to be provided in order to allocate the correct size.

4.6.2 Message Send

Prototype:
void ke_msg_send(void const *param_ptr)

Parameters:

Type	Parameters	Description
void const *	param_ptr	Pointer to the parameter member of the message that should be sent

Return: None.

Description:

Send a message previously allocated with any ke_msg_alloc()-like functions. The kernel will take care of freeing the message memory.

Once the function have been called, it is not possible to access its data anymore as the kernel may have copied the message and freed the original memory.

4.6.3 Message Send Basic

Prototype:

void ke_msg_send_basic (ke_msg_id_t const id,
ke_task_id_t const dest_id,
ke_task_id_t const src_id)

Parameters:

Type	Parameters	Description
ke_msg_id_t	id	Message Identifier
ke_task_id_t	dest_id	Destination Task Identifier
ke_task_id_t	src_id	Source Task Identifier

Return: None.

Description:

Send a message that has a zero length parameter member. No allocation is required as it will be done internally.

4.6.4 Message Forward

Prototype:

void ke_msg_forward (void const *param_ptr,
ke_task_id_t const dest_id,
ke_task_id_t const src_id)

Parameters:

Type	Parameters	Description
void const *	param_ptr	Pointer to the parameter member of the message that should be sent
ke_task_id_t	dest_id	Destination Task Identifier
ke_task_id_t	src_id	Source Task Identifier

Return: None.

Description:

Forward a message to another task by changing its destination and source tasks IDs.

4.6.5 Message Free

Prototype:

void ke_msg_free (struct ke_msg const *msg)

Parameters:

Type	Parameters	Description
struct ke_msg const *	msg	Pointer to the message to be freed

Return: None.

Description:

Free allocated message.

5 Scheduler

5.1 Overview

- The scheduler is called in the main loop of the background.
- In background loop, the Kernel checks if the event field is non-null, and executes the event handlers for which the corresponding event bit is set

5.2 Requirements

5.2.1 Priority Management

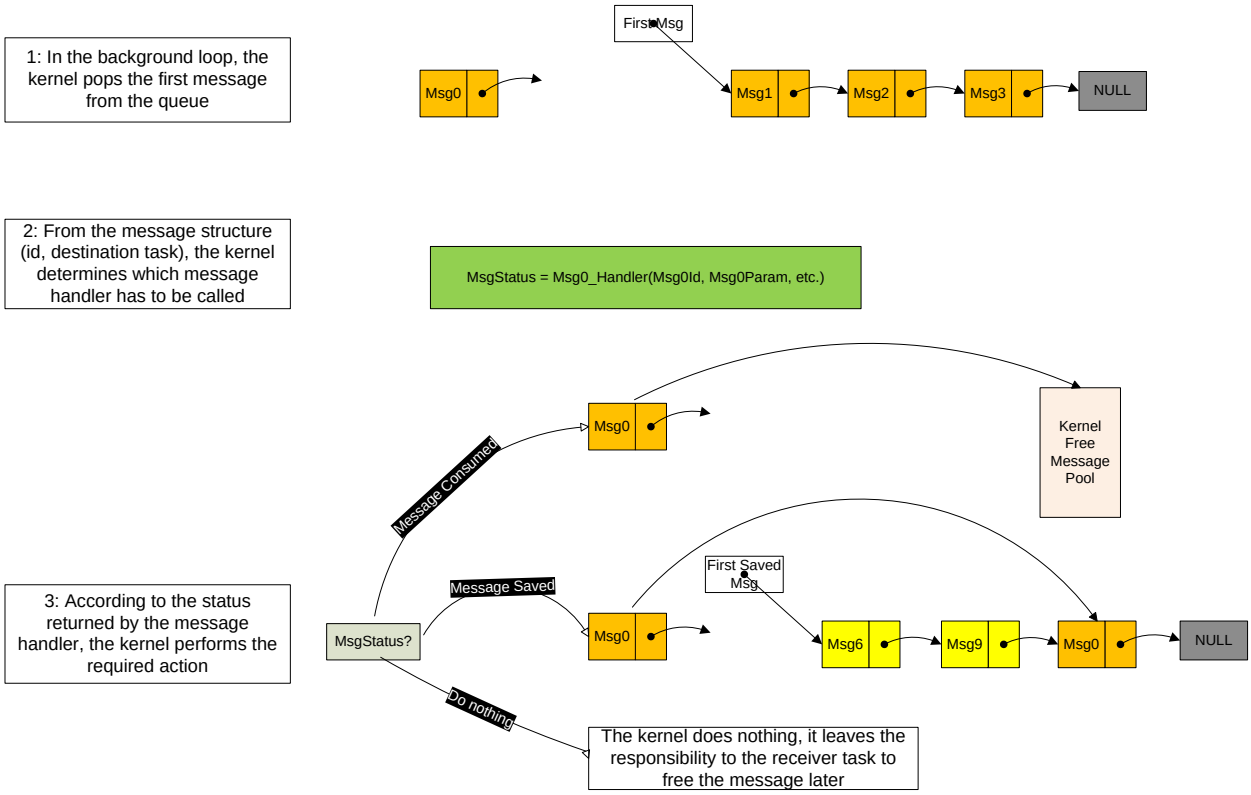
The scheduler checks the event field and gets the one with the highest priority.
The function below is in charge of finding the event marked with the highest priority set:

co_clz(field);

The event priority is defined in the ke_event.h file.
0 has the highest priority and up to the value 31 (which has the lowest priority).

```
enum
{
        KE_HIGHEST_PRIO,
        …
        KE_LOWEST_PRIO,
        KE_EVT_MAX
};
```

5.2.2 Scheduling Algorithm



5.2.3 Save Service

The Save service allows to “SAVE” a message i.e. to store it in memory without being consumed. If the task state change after a message is received the scheduler will try to handle the saved message before scheduling any other signals.

6 Tasks

6.1 Definition

A kernel task is defined by:

- Its task type, i.e. a constant value defined by the kernel, unique for each task
- Its task descriptor, which is a structure containing all the information about the task
    - The messages that it is able to receive in each of its states
    - The messages that it is able to receive in the default state
    - The number of instances of the task
    - The number of states of the task
    - The current state of each instance of the task

The kernel keeps a pointer to each task descriptor, which is used to handle the scheduling of the messages transmitted from a task to another one

7 Kernel Timer

7.1 Overview

- RW Kernel provides a Time reference (absolute time counter),
- RW Kernel provides timer services: Start, Stop timer
- Timers are implemented by the mean of a reserved queue of delayed messages,
- Timer messages do not have parameters.

7.2 Time definition

- Time is defined as duration, the minimum step is 10ms.

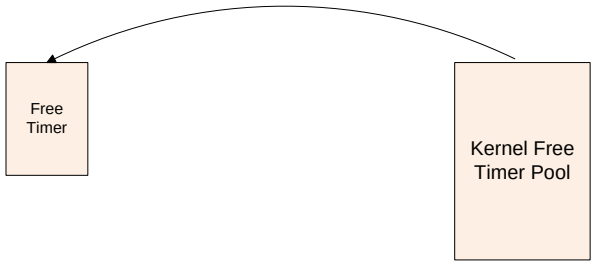
7.3 Timer Object

The structure of the timer message contains:

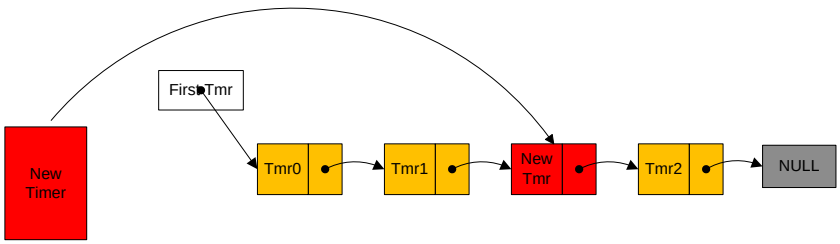
- ***next:** Pointer on the next timer
- **id:** Message identifier.
- **task:** Destination task identifier.
- **time**: Duration.

7.4 Timer Setting

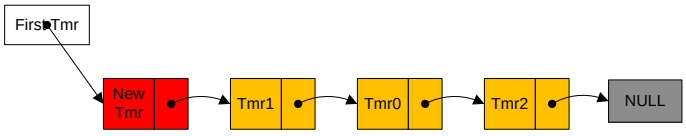
1: When a task requests to program a timer in the future, the kernel allocates one timer structure from the free pool. The kernel then fills the new timer with the parameters passed by the task (time, timer_id, task_id)



2: The kernel pushes the new timer at the right place into the ordered timer list



3: If the new timer is the first of the list, then the kernel programs the HW gross target time to match the target time of the new timer



7.5 Timer Primitives

7.5.1 Timer Set

Start or restart a timer.

Prototype:

void ke_timer_set(ke_msg_id_t const timer_id, ke_task_id_t const task, uint16_t const delay);

Parameters:

Type	Parameters	Description
ke_msg_id_t	timer_id	Timer identifier
ke_task_id_t	task id	Task identifier
uint16_t	delay	Timer duration (multiple of 10ms)

Return: None

Description:

The function first cancel the timer if it exists, then it creates a new one. The timer can be one-shot or periodic, i.e. it will be automatically set again after each trigger.

7.5.2 Timer Clear

Remove a registered timer.

Prototype:

void ke_timer_clear(ke_msg_id_t const timer_id, ke_task_id_t const task);

Parameters:

Type	Parameters	Description
ke_msg_id_t	timer_id	Timer identifier
ke_task_id_t	task id	Task identifier

Return: None

Description:

This function searches for the timer element identified by its timer and task identifiers. If found, it is stopped and freed, otherwise an error message is returned.

7.5.3 Timer Activity

Check if a requested timer is active.

Prototype:

bool ke_timer_active(ke_msg_id_t const timer_id, ke_task_id_t const task);

Parameters:

Type	Parameters	Description
ke_msg_id_t	timer_id	Timer identifier
ke_task_id_t	task id	Task identifier

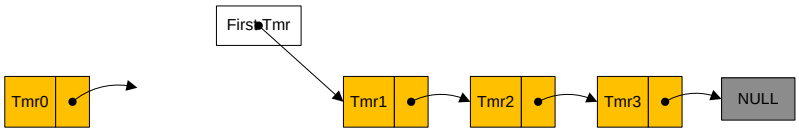
Return: TRUE if the timer identified by Timer Id is active for the Task id, FALSE otherwise

Description:

This function pops the first timer from the timer queue and notifies the appropriate task by sending a kernel message. If the timer is periodic, it is set again; if it is one-shot, the timer is freed. The function checks also the next timers and process them if they have expired or are about to expire.

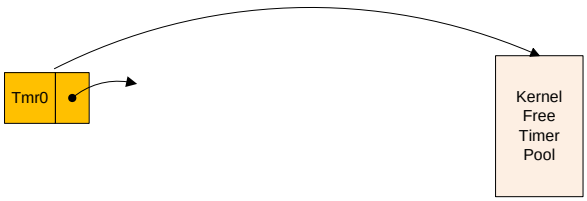
7.5.4 Timer Expiry

1: Upon gross target timer expiry, a kernel event is scheduled in background. In the event scheduler, the kernel pops the first timer from the queue

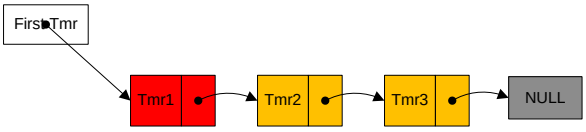


2: From the timer structure (tmr_id, destination task), the kernel sends the corresponding message to the task that programmed the timer

3: The kernel frees the elapsed timer



4: The kernel programs the HW gross target time to match the target time of the next timer



8 Effective Macros

- Builds the task identifier from the type and the index of that task:
  #define KE_BUILD_ID(type, index) ((ke_task_id_t)(((index) << 8)|(type)))

- Retrieves task type from task id:
  #define KE_TYPE_GET(ke_task_id) ((ke_task_id) & 0xFF)

- Retrieves task index number from task id:
  #define KE_IDX_GET(ke_task_id) (((ke_task_id) >> 8) & 0xFF)