Regensburg University of Applied Sciences
Regensburg University of Applied Sciences
Department of Computing

Implementation and Test of EDF and LLREF Schedulers in FreeRTOS

Enrico Carraro

Submitted in part fulfilment of the requirements for the degree of
(change in icthesis.sty) in Computing of the University of London and
the Diploma of Imperial College, April 2016

Universitá degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

Implementation and Test of EDF and LLREFSchedulers in FreeRTOS

Laureando:
Enrico Carraro

Relatori:
Prof. Michele Moro
Prof. Jürgen Mottok

Anno accademico 2015/2016

Abstract

L'evoluzione tecnologica dei sistemi embedded ha reso possibile l'esecuzione di applicazioni sempre piú complesse, rendendo sempre piú preferibile, se non necessario, l'adozione di un sistema operativo a cui demandare la gestione dell'interazione tra task e la loro schedulazione. Tali compiti sono tanto pi importanti nel caso di sistemi in tempo reale. In questa tesi viene preso in oggetto FreeRTOS, un sistema operativo real time appositamente sviluppato per piccoli sistemi embedded. Dopo una approfondita descrizione dello scheduler a priorit adottato da FreeRTOS, vengono proposti due nuovi scheduler: il primo basato sul noto algoritmo Erliest Deadline First (EDF), il secondo basato su un algoritmo recentemente proposto, Largest Local Remaining Execution time First (LLREF), originariamente sviluppato per sistemi multiprocessore. Di ciascuno scheduler proposto ne viene descritta l'implementazione in FreeRTOS, e quindi ne viene validata la correttezza attraverso una fase di test.

Abstract

Embedded systems technological evolution has made the execution of increasingly complex applications possible. Due to this increasing complexity, the adoption of an operating system to manage the interaction between tasks and their scheduling is becoming preferable and even necessary, also for little embedded systems. This thesis examines FreeRTOS scheduler. FreeRTOS is a real time operating system specially developed for small embedded systems. After an in-depth description of the priority-based scheduler adopted by FreeRTOS, two new schedulers are proposed: the first one is based on the well-known Earliest Deadline First algorithm (EDF), the second one is based on a new algorithm, Largest Local Remaining First Execution time (LLREF), originally developed for multiprocessor systems. For each proposed scheduler, an implementation description on FreeRTOS is given. Then, their correctness is verified by a test phase.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Real time Systems

Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. If the timing constraints of the system are violated, system failure occurs or a punishment is incurred for the violation. Hence, it is essential that the timing constraints of the system are guaranteed to be met. It is also desirable that the system attain a high degree of utilization while satisfying the timing constraints of the system[1].

Real-time systems span a broad spectrum of complexity from very simple micro controllers to highly sophisticated, complex and distributed systems. Some examples of real-time systems include process control systems, flight control systems, manufacturing applications, robotics, intelligent highway systems, and high speed and multimedia communication systems [2]. For instance, the objective of a computer controller might be to command the robots to move parts from machines without colliding with other objects. If the computer controlling a robot does not command it to stop or turn in time, the robot might collide with another object on the factory floor.

A real-time system will usually have to meet many demands within a limited time. The importance of the demands may vary with their nature (e.g. a safety-related demand may be more important than a simple data-log) or with the time available for a response. Thus, the allocation of the system resources needs to be planned so that all demands are met by the time of their respective deadlines. This is usually done using a scheduler which implements a scheduling policy that determines how the resources of the system are allocated to the demands.

A real-time application is normally composed of multiple tasks with different levels of criticality. We can formally define a real-time system as follows[1]: let's consider a system consisting of a set of tasks, $T = \tau_1, \tau_2, ..., \tau_n$, where the finishing time of each task $\tau_i T$ is F_i. The system is said to be real-time if there exists at least one task $\tau_i T$, which falls into one of the following categories:

- Task τ_i is a $hardreal - time$ task - the execution of the task τ_i must be completed by a given deadline d_i;
- Task τ_i is a $softreal - timetask$ - the later the task τ_i finishes its computation after a given deadline d_i, the more penalty it pays. A penalty function $G(\tau_i)$ is defined for the task. If $F_i d_i$, the penalty function $G(\tau_i)$ is zero. Otherwise $G(\tau_i) > 0$.

1.2 Real Time Scheduling

Basically, the scheduling problem for a real-time system is to determine a schedule for the execution of the tasks in order to satisfy their timing constraints. For scheduling a real-time system, we need to have enough information, such as the deadline, release time and execution time of each task. Also, it is required to know the importance of the task as compared with the other tasks and its precedence relation. A majority of systems assume that much of this information is available a priori.
A Real Time Scheduler Algorithm can be classified according to several properties:
-**preemptive/non preemptive behaviour**: In some real-time scheduling algorithms, a task can be preempted if another task of higher priority becomes ready. In contrast, the execution of a non preemptive task should be completed without interruption once it is started;

-**periodic/sporadic tasks management**: Periodic real-time tasks are released regularly at fixed rates (periods). A majority of sensory processing is periodic in nature. For example, a a digital thermometer that measures the temperature in an industrial tank produces data at a fixed rate; sporadic real-time tasks are activated irregularly with some known bounded rate;
-**static/dynamic priority scheduling**: In priority driven scheduling, a priority is assigned to each task. Assigning the priorities can be done statically or dynamically while the system is running.

1.2.1 FreeRTOS

FreeRTOS is a Real Time Operating System (RTOS) that is designed to be small enough to run on a microcontroller - although its use is not limited to microcontroller applications. FreeRTOS provides the core real time scheduling functionality, inter-task communication, timing and synchronisation primitives. Additional functionality, such as a command console interface, or networking stacks, can be then be included with add-on components[1]. FreeRTOS scheduler is preemptive and fixed-priority based. At the initialization of a task, a priority is assigned to it. If multiple tasks have equal priority, it uses round-robin scheduling among them.

1.3 Aims of the project

As we saw, FreeRTOS uses a static scheduler where tasks are given a fixed priority.
The goal of this project is to implement two new dynamic priority scheduler algorithms for FreeRTOS: the first one is based on the well-known Earliest Deadline First algorithm (EDF)[3], the second one is based on a new algorithm, Largest Local Remaining First Execution time (LLREF)[4], originally developed for multiprocessor systems. For each proposed scheduler, an implementation description on FreeRTOS is given. Then, their correctness is verified by a test phase.

1.4 Environment used

In this section we will illustrate the environment used for this project:

- **Target Board**: ST STM32F429Discovery Board - ARM Cortex M4 180MHz CPU;

  - 2 Mbytes of Flash memory;

  - 256 Kbytes of RAM;

  - 2.4" QVGA TFT LCD;

  - USB port;

- **Real Time OS:** FreeRTOS 8.2.2;

- **PC Host:** Quad Core Intel i5 With Windows 10 - GCC compiler; - STM32 ST-LINK USB board driver; - CooCox CoIde ARM Cortex Development tool-chain;

The project folder is structured as follows:

FREERTOSscheduler is the main folder; it contains:
-*cmsis*, *cmsis_boot*, *cmsis_lib*, *stm32f429* folders contain hardware dependent code for the board management;
-*freertos* folder contains the FreeRTOS files code; -*semihosting* folder contains files to enable semihosting debug mode for CoIde;
-*main.c* and *main.h* files, the demo application to test.

1.5 Organisation of the paper

The rest of this thesis is organized as follows. Chapter 2 contains a detailed description of the FreeRTOS scheduler: the system structures and functions involved are introduced and a complete schedule example is given.

In Chapter 3 we will present our EDF scheduler implementation for FreeRTOS: Earliest Deadline First algorithm is presented, then design choices are discussed. The final implementation is described with code samples, and test results are shown to ensure the correctness of the algorithm. LLREF scheduler implementation for FreerTOS is presented in Chapter 4. Like in Chapter 3, LLREF algorithm is presented, design choices are discussed and the final implementation is described. A test phase then shows the correctness of the algorithm implementation. Chapter 5 summarises the project, taking a view of the work done.

Chapter 2

FreeRTOS Task Scheduling

2.1 Introduction to FreeRTOS

FreeRTOS is an open-source Real Time Operating System designed for embedded systems. The FreeRTOS project started in 2002 and is under active development. Its official support to 35 embedded system architectures and different compiler tool-chains, its simple and full documented API, and its open-source license contributed to diffuse it among the embedded market, while the user base grows year after year[5].

2.2 Kernel structure

Since FreeRTOS works in embedded environments, it is designed to minimize the memory usage and is also suitable for low clock frequency microcontrollers: the FreeRTOS minimum kernel consists of only three source files, for less than 9000 line of code. In order to be compatible with all the supported architectures and tool-chains, FreeRTOS kernel is composed by a hardware dependent layer, customized for every supported architectures, and a hardware independent layer, common to all the ports. Figure 2.1 shows the FreeRTOS layers.

The 3 source files that compose the minimal kernel (alongside a handful of header files)provide these functions:

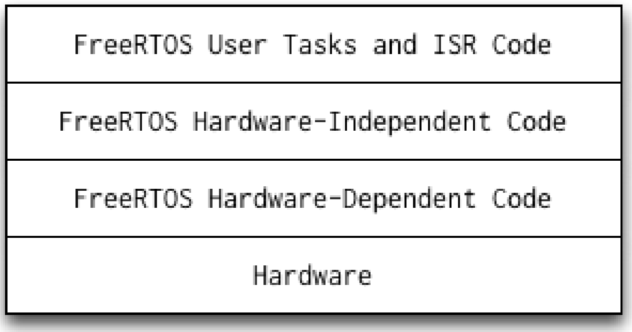

Figure 2.1: layers

- **task.c** : the task function is defined, and its life cycle is managed. Scheduling functions are also defined here.
- **queue.c** : In this file the structures used for task communication and synchronisation are described- tasks and interrupts communicate witch each other using queues to exchange messages; semaphores and mutexes are used to synchronize the sharing of critical resources.
- **list.c** : the list data structure and its maintaining functions are defined. Lists are used both by task functions and queues.

2.3 The Task

2.3.1 Task structure

Tasks are implemented as C functions. Every single task created is a small program on its own right, at witch a priority is assigned. Each task executes within its own context, without dependency on other task's context. At each instant the OS selects the task that will be executed, according to its priority. At each task, FreeRTOS associates a proper data structure called Task Control Block (TCB) that contains the following parameters:

```
typedef struct tskTaskControlBlock
{
```

```
    volatile StackType_t *pxTopOfStack; /*< Points to the location of
                                            the last item placed on the
                                            tasks stack.*/
    ListItem_t      xGenericListItem; /*< The list that the state list item of a task
                                            is reference from denotes the state of
                                            that task (Ready, Blocked, Suspended ).
                                               */
    ListItem_t      xEventListItem;    /*< Used to reference a task from an event
        list. */
    UBaseType_t     uxPriority;      /*< The priority of the task.
                                            0 is the lowest priority. */
    StackType_t     *pxStack;        /*< Points to the start of the stack. */
    char         pcTaskName[ configMAX_TASK_NAME_LEN ];/*< Descriptive name given to
                                                            the task when created.
                                                            Facilitates debugging
                                                               only. */
    StackType_t   *pxEndOfStack;     /*< Points to the end of the stack on
        architectures
                                           where the stack grows up from low memory.
                                              */
    UBaseType_t   uxBasePriority;       /*< The priority last assigned to the task
                                            - used by the priority inheritance
                                               mechanism. */
} tskTCB;
```

The TCB contains general information characterizing the task:

-stack pointers: ***pxStack** points to the beginning of task stack beginning, ***pxTopOfStack** points the current top of the stack, and a third pointer, ***pxEndOfStack**, used for stack overflow checking, points to the end of the stack;

-**uxPriority** variable contains the task priority, while **uxBasePriority** contains the latest assigned priority (used by the priority inheritance mechanism);

-*ListIteam* objects **xGenericListItem** and **xEventListItem**: when a task is inserted in a list (for example, the Ready List, as we will see), the list contains not a simple pointer to the Task Control Block, but a pointer to an object *ListIteam*. The usage of *ListIteam* elements guarantees lists to be more intelligent and to compute operations with less computational complexity;

-**pcTaskName** is a char vector containing the task name.

2.3.2 Task states

As shown in Figure 2.2, a task can exist in one of the following states:

- **Running** : the task pointed by $*pcCurrentTCB$ system variable is said to be in Running state. It is currently utilising the processor. Only one task can be executed at one time;
- **Ready** : tasks that are ready to be executed and are waiting for being scheduled, but are not executing because another task with equal or higher priority is in Running state.
- **Blocked** : a task in Blocked state cannot be scheduled, because it is waiting for an external event or a temporal event. For example a running task calling the method $vTaskDelay()$ will block itself being placed in the Blocked state, waiting for a delay period, or another task could block waiting for queue and semaphore events.
- **Suspended** : a task can reach or leave the Suspended state only by explicitly calling the $vTaskSuspend()$ and $xTaskResume()$ method respectively. Suspended tasks are not available for scheduling.

The Task Control Block does not contain a variable that represents the task state: instead, FreeRTOS manages lists containing tasks for each state -Ready, Blocked and Suspended- so, task state is tracked implicitly by putting tasks in the proper list.

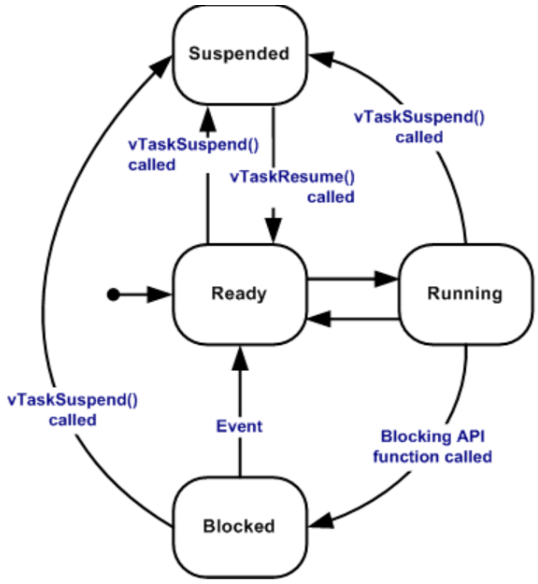

Figure 2.2: Task valid states and transitions

- **Ready tsks** :

```
PRIVILEGED_DATA static List_t pxReadyTasksLists[ configMAX_PRIORITIES ];/*<
    Prioritised ready tasks. */
```

-*pxReadyTasksLists*[] is an array of lists, containing as many list as the maximum number of priority selected. The i-th position of the array contains the list of the tasks having the i-th priority.

- **Blocked tasks** :

```
PRIVILEGED_DATA static List_t xDelayedTaskList1;           /*< Delayed tasks.
    */
PRIVILEGED_DATA static List_t xDelayedTaskList2;           /*< Delayed tasks
    (two lists are used - one for delays that have overflowed the current tick
    count. */
PRIVILEGED_DATA static List_t * volatile pxDelayedTaskList;    /*< Points to
    the delayed task list currently being used. */
PRIVILEGED_DATA static List_t * volatile pxOverflowDelayedTaskList;   /*<
    Points to the delayed task list currently being used to hold tasks that
    have overflowed the current tick count. */
```

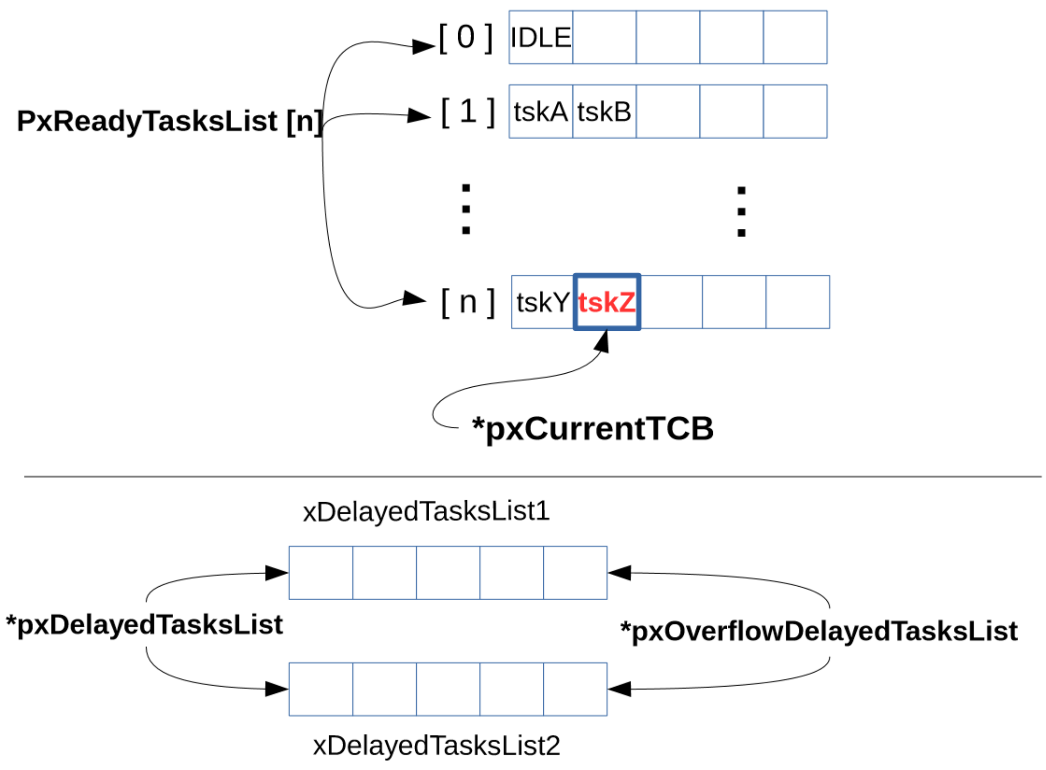

Figure 2.3: graphic representation of Ready List and Delayed List

-two lists are used to manage blocked tasks: $xDelayedTaskList1$ and $xDelayedTaskList2$. One list contains tasks which awakeness time has overflowed the current tick count. At each moment, $*pxDelayedTaskList$ points at the Delayed list currently used, while $pxOverflowDelayedTaskList$ points the other one. When the tick count overflows, then the pointers switch each other.

- **Suspended tasks** :

```
PRIVILEGED_DATA static List_t xSuspendedTaskList;  /*< Tasks that are
    currently suspended. */
```

-a simple list is used to contain suspended tasks.

2.3.3 Task initialization

A task is created invocating the task.c method $xTaskCreate()$:

```
BaseType_t xTaskCreate(
                    TaskFunction_t pvTaskCode,
                    const char * const pcName,
                    uint16_t usStackDepth,
                    void *pvParameters,
                    UBaseType_t uxPriority,
                    TaskHandle_t *pvCreatedTask
                 );
```

xTaskCreate() create a new task with an assigned priority, and add it to the Ready Task set. In details a task is create in these steps:

```
 /* Allocate the memory required by the TCB and stack for the new task,
checking if the allocation was successful. */
prvAllocateTCBAndStack( usStackDepth, puxStackBuffer );
```

-memory space is allocated for a new TCB struct and for a new stack (if enough memory is available);

```
 /* Setup the newly allocated TCB with the initial state of the task. */
prvInitialiseTCBVariables( pxNewTCB, pcName, uxPriority, xRegions, usStackDepth
    );
```

-TCB variables are initialized;

```
 /* Initialize the TCB stack to look as if the task was already running,
but had been interrupted by the scheduler. The return address is set
to the start of the task function. Once the stack has been initialised
the  top of stack variable is updated. */
 pxPortInitialiseStack( pxTopOfStack, pxTaskCode, pvParameters );
```

-stack is initialized as well: a new task stack is initialized in a way that it looks like a stack of a task suspended by the scheduler. In this way, the scheduler does not need special case code to manage new tasks, since they look the same as old tasks already switched off. *pxPortInitialiseStack*() is a hardware-dependent function implemented in the port.c file. As we will see, when a task is interrupted, all task context is saved on the task stack. So the new stack created is modified and looks as though the registers have been pushed, even if the task has not used them yet;

```
	prvAddTaskToReadyList( pxNewTCB );
```

-the created task is added to the Ready set. As said, *pxReadyTasksLists*[] array contains one ready list for each possible priority level (level 0 is the lowest one). A priority p task will be placed in the corresponding *pxReadyTasksLists*[p] list. *prvAddTaskToReadyList*() is defined in this way:

```
	#define prvAddTaskToReadyList( pxTCB )                                  \
	                                                   \
	  taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority );
	      \
	  vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( ( pxTCB
	      )->xGenericListItem ) )
```

basically, first the system variable $UBaseType_t uxTopReadyPriority$, representing in every moment the priority of the running task, is compared with the new task priority, and if the new priority is higher, *uxTopReadyPriority* is updated. Then, the task's *xGenericListItem* is inserted at the end of the proper Ready list of the *pxReadyTasksLists*[] array.

The task is now in the Ready list waiting for be executed by the scheduler.

2.3.4 Delay a Task

We saw how a task is created and initialized to the Ready state. Now we will see how a task can reach the Blocked state by calling the $vTaskDelayUntil()$: function. This function defines a frequency at which the task is periodically executed, so it can be used to implement periodic tasks. As we will see, FreeRTOS measures time by periodically increasing the tick count variable. $vTaskDelayUntil()$: moves the invoking task to the Waiting list, where it waits for a chosen time interval before being moved to the Ready list again, periodically.

```
* @param pxPreviousWakeTime: Pointer to a variable that holds the time at which the
* task was last unblocked. The variable must be initialised with the current time
* prior to its first use (see the example below). Following this the variable is
* automatically updated within vTaskDelayUntil ().
*
* @param xTimeIncrement: The cycle time period. The task will be unblocked at
* time *pxPreviousWakeTime + xTimeIncrement. Calling vTaskDelayUntil with the
* same xTimeIncrement parameter value will cause the task to execute with
* a fixed interface period.
```

The function works in this way:

```
    uxListRemove( &( pxCurrentTCB->xGenericListItem ) )
```

- $pxCurrentTCB$ is pointing at the running task that called $vTaskDelayUntil()$:, so its $xGenericListItem$ is removed from the Ready list in witch it was stored;

```
    /* The list item will be inserted in wake time order. */
  listSET_LIST_ITEM_VALUE( &( pxCurrentTCB->xGenericListItem ), xTimeToWake );
```

- $xGenericListItem$ now contains the value of the tick at witch the task will be unblock;

```
    vListInsert( pxDelayedTaskList, &( pxCurrentTCB->xGenericListItem ) );
```

- the *xGenericListItem* is inserted in the *DelayedTaskList*. *DelayedTaskList* contains the *xGenericListItem* of the other tasks in blocked state, sorted by the unblock time value. So the top of the list contains the *xGenericListItem* of the task closer to being unblocked. *vListInsert*() function insert the new item in the list maintaining it sorted.

```
/* If the task entering the blocked state was placed at the head of the
 list of blocked tasks then xNextTaskUnblockTime needs to be updated
 too. */
 if( xTimeToWake < xNextTaskUnblockTime )
 {
    xNextTaskUnblockTime = xTimeToWake;
 }
```

-finally, the system variable *xNextTaskUnblockTime*, containing the time at witch the next task unblock will occur, is updated if needed.

at this point, a context switch is needed. The hardware-dependent function *portYIELD_WITHIN_API*() is called, and the highest priority task in the Ready List is selected to execute. In the next paragraph we will see how a context switch works in FreeRTOS.

2.4 Context Switch

Context Switch must occur in a transparent way with respect to the tasks involved: in fact a task does not know when it is going to get suspended or resumed by the system, it might just continue its execution flow as if no context switch have occurred. The OS is in charge to do that: when the running task is switched out, the execution context is saved in its stack, ready to be restored when the task will execute again.

Figure 2.4 shows a representation of a task execution context: the Stack Pointer (SP) register points to the running task stack, the Program Counter (PC) register points to the next instruction in the task's code, and the CPU registers are used by the task.

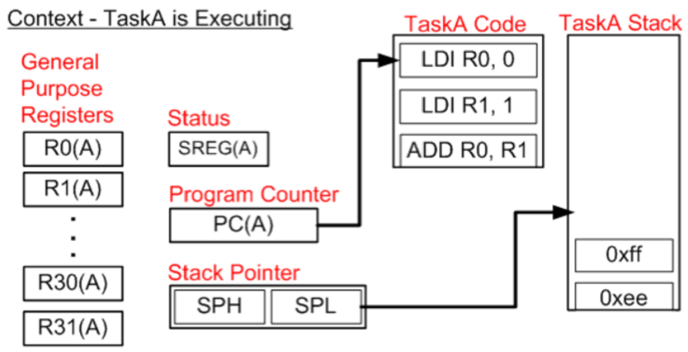

Figure 2.4: Task execution context [from freertos.org]

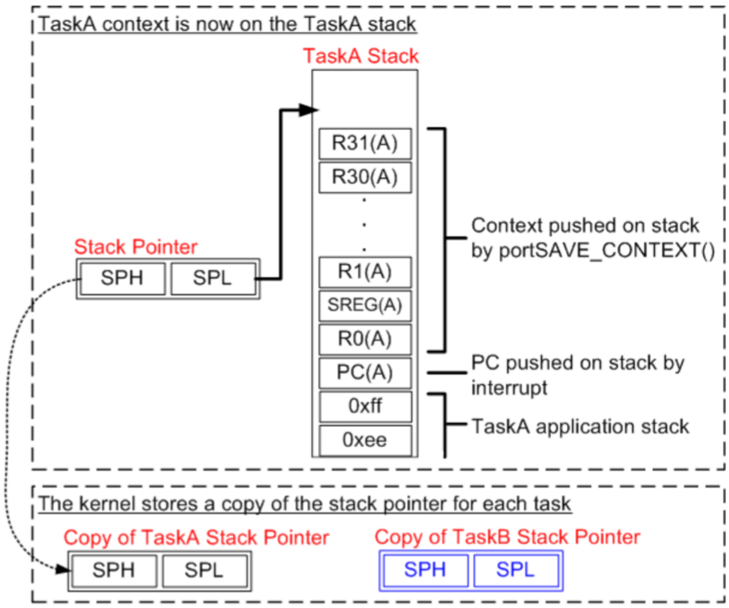

Figure 2.5: Task stack after saving context [from freertos.org]

$portSAVE\_CONTEXT()$ is an hardware based function in charge for saving the execution context: the PC and SP registers, along with the other general purpose registers are pushed on the task stack. Figure 2.5 shows the task stack after the execution context got saved. A copy of the Stack Pointer is saved by the kernel: the OS stores the stack pointers of all suspended tasks in order to retrieve them when tasks are resumed.

A task context is restored by the $portRESTORE\_CONTEXT()$ function: The kernel retrieves the task stack pointer that have beeb previously stored, then POP's the saved execution context back into the correct processor registers.

2.5 The Tick System

We saw that when $xTaskDelayUntill()$ function is called, the calling task will specify a time after which it requires waking. FreeRTOS measures time using a tick count system variable. The tick interrupt activates an Interrupt Service Routine (ISR) that increments the tick count with strict temporal accuracy, allowing the real time kernel to measure time to a resolution of the chosen timer interrupt frequency. Each time the tick count is incremented the OS must check if it is now time to wake a task. It is possible that a task woken during the tick ISR will have a priority higher than that of the interrupted task. If this happens, the tick ISR should return to the newly woken task. A context switch forced by the system in this way is said *preemptive*. Below will be descripted the ISR tick function, from the port.c file:

```
    void vPortYieldFromTick( void )
{
    portSAVE_CONTEXT();

    if(xTaskIncrementTick() != pdTRUE)
    {
        vTaskSwitchContext();
    }

    portRESTORE_CONTEXT();

    asm volatile ( "ret" );
}
```

the first thing that $vPortYieldFromTick$ does is saving the execution context with $portSAVE\_CONTEXT$ then two functions from the hardware independent layer are called:

- xTaskIncrementTick() increments the tick count variable and check if it is time to wake up tasks from the blocked state to the ready state: if so, then that tasks are removed from the $BlockedTaskList$ and are putted in the proper Ready List. The function returns true if same tasks got awaken, in order to let the IRS to know if a context switch is needed;

-vTaskSwitchContext() sets $*pxCurrentTCB$ to the TCB of the highest priority task staying in Ready List.

-finally $portRESTORE\_CONTEXT()$ function restores the context from the stack of the task pointed by $*pxCurrentTCB$.

2.6 Scheduling Example

In this section two scheduling example are shown: the first describes a preemptive context switch, where the kernel interrupts the execution flow of the running task and assign the CPU to another task; the second example shows a non-preemptive context switch, where a task calls $xTaskDelayUntill()$ function and another task in the Ready List is executed. The preemptive example is shown in Figure ??: When tick=3, tskA is the running task and tskB is waiting to be awakened; Then, a tick interrupt occurs and $vPortYeldFromTick()$ ISR is called. The interrupt service routine saves the running task context (tskA), and calls thexTaskIncrementTick() method:
-tickCount variable is incremented (tickCount= 4);
-tskA TCB is removed form $xDelayedTaskList$;
-tskA $GenericListIteam$ is insered in $pxReadyTasksList[2]$, since tskA priority is 2; because at least one task has been awakened,vTaskSwitchContext() method is called, so $*pxCurrentTCB$ points to tskA;
portRESTORECONTEXT() restores the context of the task pointed by $*pxCurrentTCB$, so from now tskA is executing.

For the non-preemptive example let's consider the Figure ??. At tickCount=26 the situation is described in Figure ??-a: tskA is running; As shown in Figure ??-b, at tickCount=12 tskA

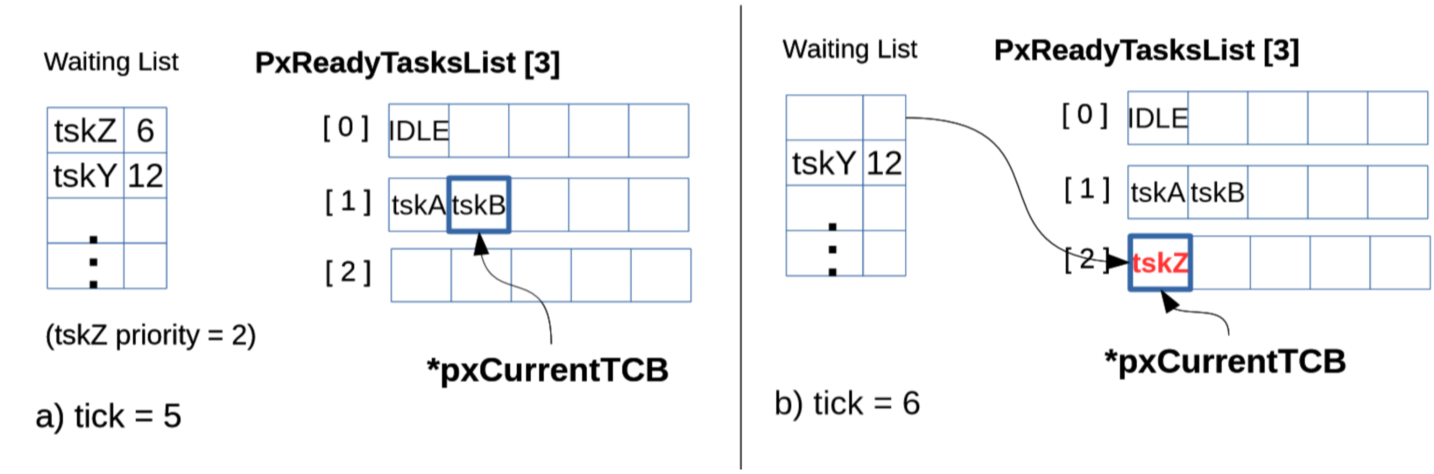

Figure 2.6: preemptive context switch - a))description of the Waiting List and Ready List at tick=26; b) when tick=27, tskA moves in the Ready List and goes in Running state

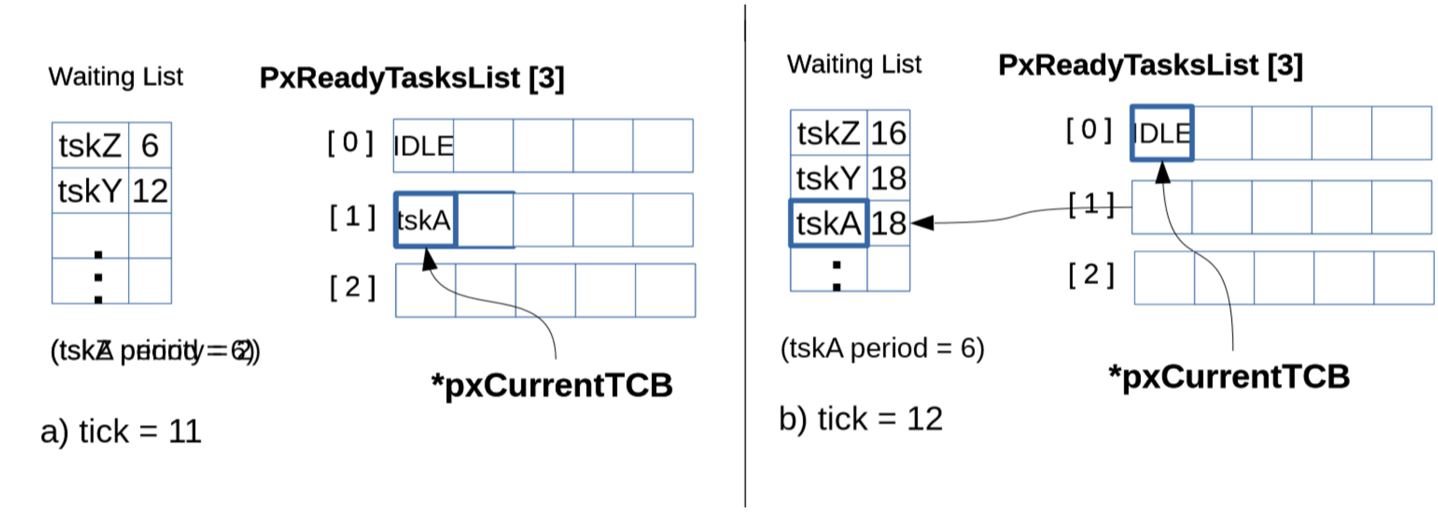

Figure 2.7: graphic representation of Ready List and Delayed List

finishes its execution by calling $delayTaskUntill()$ function:

-tskA TCB is removed from $xReadyTaskList[1]$;

-tskA $GenericListIteam$ is set to the next awake time;

-tskA TCB is inserted in $xDelayedTaskList$;

-$portYELD\_WITH\_API()$ function is called: this method force a context switch, so tskA execution context is saved ($portSAVE\_CONTEXT()$), $vTaskSwitchContext()$ makes $*pxCurrentTCB$ pointing to the TCB of the highest priority task in Ready List, i.e. IDLE; then $portRESTORE\_CONTEXT()$ restores IDLE task context.

[6]

Chapter 3

EDF Scheduler

3.1 Earliest Deadline First Algorithm

The first scheduler we will implement is based on the Earliest Deadline First algorithm (EDF)[3]. EDF adopts a dynamic priority-based preemptive scheduling policy, meaning that the priority of a task can change during its execution, and the processing of any task is interrupted by a request for any higher priority task.
The algorithm assigns priorities to tasks in a simple way: the priority of a task is inversely proportional to its absolute deadline; In other words, the highest priority is the one with the earliest deadline. In case of two or more tasks with the same absolute deadline, the highest priority task among them is chosen random.
The algorithm is suited to work in an environment where these assumptions applies[3]:

- **(A1)** The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests.
- **(A2)** Deadlines consist of run-ability constraints only, i.e. each task must be completed before the next requests for it occurs.
- **(A3)** The tasks are independent in that requests for a certain task do not depend on the initialization or the completion of requests for other tasks.

- **(A4)** Run-time for each task is constant for that task and does not vary with time. Run-time refers to the time which is taken by a processor to execute the task without interruption.

- **(A5)** Any non-periodic tasks in the system are special; they are initialization or failure-recovery routines; they displace periodic tasks while they themselves are being run, and do not themselves have hard, critical deadlines.

Due to these assumption, we can characterize a task using only two parameters: its period and its run-time. We shall use $\tau_1, \tau_2, \ldots, \tau_m$ to denote m periodic tasks, with their request periods being $T_1, T_2, \ldots, T_m$ and their run-times being $C_1, C_2, \ldots, C_m$, respectively. So, task τ_i is released every T_i time units and must be able to consume at most C_i units of CPU time before reaching its deadline, T_i time units after release ($C_i \leq T_i$).
The following theorem about the schedulability of a task set with EDF can be proven:

Theorem 3.1 *A task set of periodic tasks is schedulable by EDF if and only if:*

$$U = \sum_{i=1}^{N} \frac{Ci}{Ti} \leq 1$$

Let's consider two tasks, A and B, such described in table 3.1.

$$U = \frac{2}{5} + \frac{2}{8} = 0,65 < 1.$$

According with theorem 3.1, EDF algorithm can schedule them without missing any deadline. Diagram in Figure 3.9 shows how A and B are scheduled in a common period. Both tasks start at $t = 0$, and task A is scheduled since its next deadline is closer. At time $t = 2$ task A completes, so task B executes, and so on. At $t = 25$ a preemption occurs: task B is executing at $t = 24$, when at $t = 25$ task A starts a new period and requires to be executed. Since task A new deadline ($t_A = 30$) comes first then task B deadline ($t_B = 32$), Task A becomes the executing task. At $t = 27$ task A completes, so task B can execute and finish its remaining

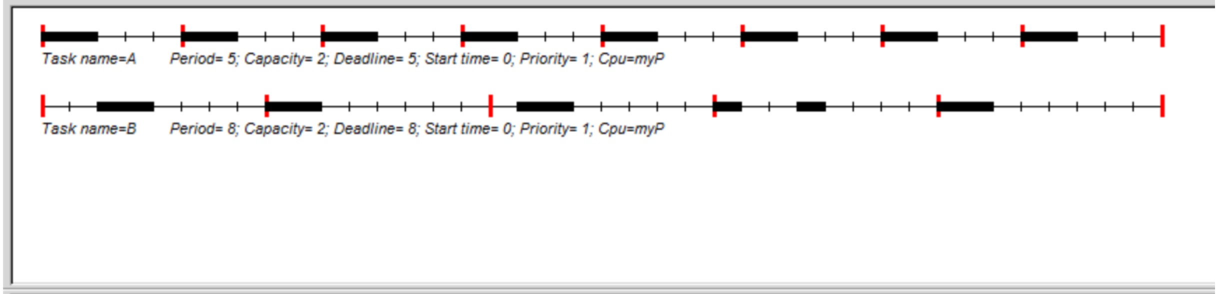

Figure 3.1: EDF schduling of task A and task B

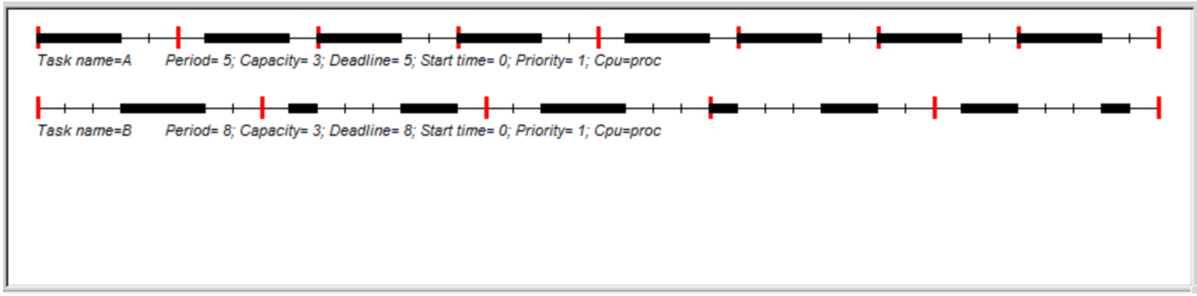

Figure 3.2: EDF scheduling of task A (T=5, C=3) and task B (T=8, C=3). In this case $U = \frac{3}{5} + \frac{3}{8} = 0,975 < 1$, so the tasks are still schedulable

run-time for this period. Figure 3.11 shows how task A and B are scheduled with $C = 3$ instead of $C = 2$.

Table 3.1: Tasks example:

	T	C
Task A	5	2
Task B	8	2

3.2 Implementation in FreeRTOS

3.2.1 General scheme

As shown previously, FreeRTOS uses a scheduler based on static priority policy. The aim of this chapter is to describe how to implement an EDF scheduler, using the existing structures that FreeRTOS offers and creating new ones. The general idea is to create a new Ready List (Figure 3.3), able to menage a dynamic task priority behaviour: it will contain tasks ordered by increasing deadline time, where positions in the list represent the tasks priorities, with the

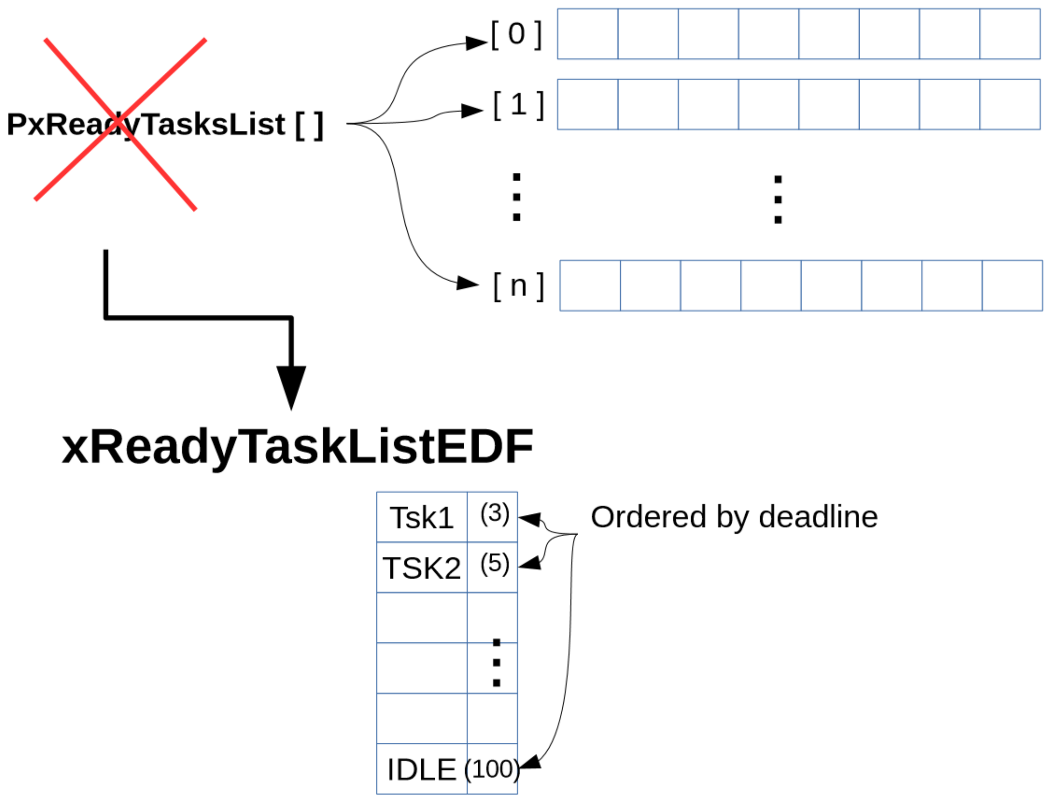

Figure 3.3: The new Ready List contains tasks ordered by increasing deadline time. The head of the list contains the task with the closest deadline

head of the list containing the running task. The rest of FreeRTOS architecture and structures, as the Waiting List and the clock mechanism are maintained with marginal changes.

The example in Figure 3.4 shows how the new Ready List works, and its interaction whit the Waiting List.

3.2.2 Implementation

This section contains implementation details of the proposed EDF scheduler. Example code will be shown, and same architectural project choice will be explained. Every time FreeRTOS code is reported, it refers to 8.2.2 version. According to what said in Cap. 3.1, these assumption still works:

- Periodic tasks only;
- task deadline equal to task period;

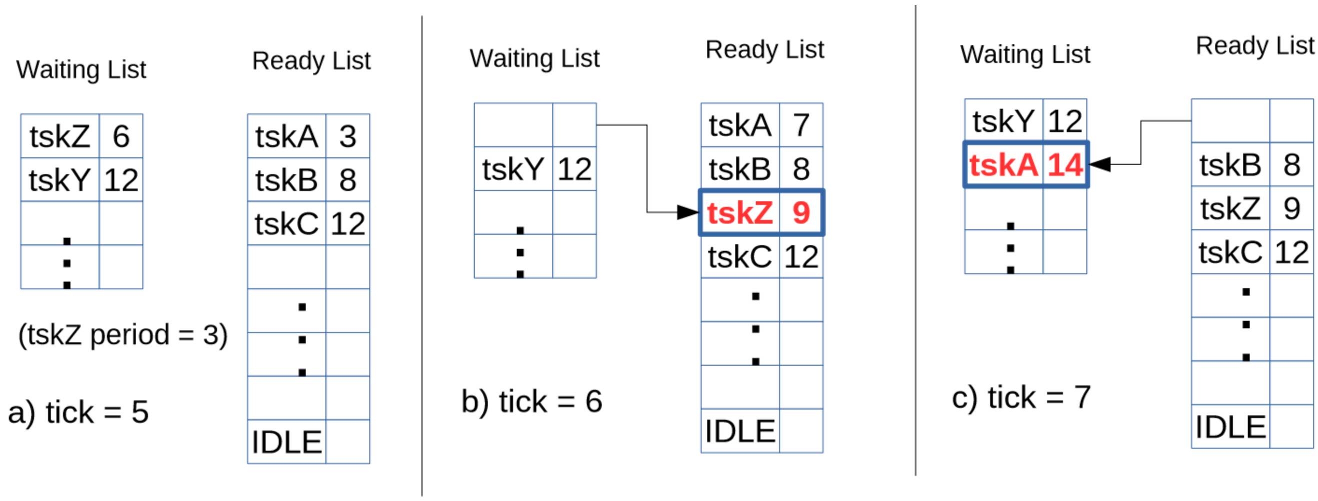

Figure 3.4: a) tskA is the running task, its deadline is at tick = 6; b) tick mechanism wakes tskZ removing it from Waiting List and adding it to Ready List, in the right position according to its new deadline: $Z_{deadline} = tick + Z_{period} = 9$; c) tskA ends its execution and is moved to the Waiting List. tskB now is on the head of the Ready List.

- only schedulable tasks set;
- independent tasks only (no shared resources and no sync issues).

All changes that will be illustrated refer to *tasks.c* file, since scheduler structures and methods are contained there. According with the FreeRTOS style guideline, a configuration variable, *configUSE_EDF_SCHEDULER*, is added to the *FreeRTOS.h* config file. When *configUSE_EDF_SCHEDULER* is set to 1, EDF scheduler is used, elsewhere the OS uses the original scheduler.

First of all, the new Ready List is declared: *xReadyTasksListEDF* is a simple list structure.

```
/* E.C. : the new RedyList */
#if ( configUSE_EDF_SCHEDULER == 1 )
  PRIVILEGED_DATA static List_t xReadyTasksListEDF;        /*< Ready tasks ordered
      by their deadline. */
#endif
```

Then, the *prvInitialiseTaskLists*() method, that initialize all the task lists at the creation of the first task, is modified adding the initialization of *xReadyTasksListEDF*:

```
static void prvInitialiseTaskLists( void )
{
    ...

   /* E.C. */
   #if ( configUSE_EDF_SCHEDULER == 1 )
   {
      vListInitialise( &xReadyTasksListEDF );
   }
   #endif

    ...

}
```

prvAddTaskToReadyList() method that adds a task to the Ready List is then modified as follows:

```
    /*
 * Place the task represented by pxTCB into the appropriate ready list for
 * the task. It is inserted at the end of the list.
 */
#if configUSE_EDF_SCHEDULER == 0 /* E.C. : */
   #define prvAddTaskToReadyList( pxTCB )                                    \
                                  \
      vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( ( pxTCB
          )->xGenericListItem ) )
#else
   #define prvAddTaskToReadyList( pxTCB ) /*xGenericListIteam must contain the
       deadline value */  \
      vListInsert( &(xReadyTasksListEDF), &( ( pxTCB )->xGenericListItem ) )
```

```
#endif
```

vListInsert() method is called to insert in *xReadyTasksListEDF* the task TCB pointer. The item will be inserted into the list in a position determined by its item value *xGenericListItem* (descending item value order). So it is assumed that *xGenericListItem* contains the next task deadline.

The second change introduced refers to the task structure. As shown in the example of Figure 3.4, when a task moves to the Ready List, the knowledge of its next deadline is needed in order to insert it in the correct position. The deadline is calculated as: $TASK_{deadline} = tick_{cur} + TASK_{period}$, so every task needs to store its period value. A new variable is added in the tskTaskControlBlock structure (TCB):

```
  /* E.C. : the period of a task */
  #if ( configUSE_EDF_SCHEDULER == 1 )
    TickType_t xTaskPeriod;       /*< Stores the period in tick of the task. > */
  #endif
```

Accordingly, a new initialization task method is created. *xTaskPeriodicCreate*() is a modified version of the standard method *xTaskGenericCreate*() shown in Cap.2, that receives the task period as additional input parameter and set the *xTaskPeriod* variable in the task TCB structure. Before adding the new task to the Ready List by calling *prvAddTaskToReadyList*(), the task's *xGenericListItem* is initialized to the value of the next task deadline.

```
/*E.C. : */
BaseType_t xTaskPeriodicCreate( < param > , TickType_t period )
{
    ...

    /*E.C. : initialize the period */
  pxNewTCB->xTaskPeriod = period;

```

```
    ...

  /*E.C. : insert the period value in the generic list iteam before to add the
     task in RL: */
  listSET_LIST_ITEM_VALUE( &( ( pxNewTCB )->xGenericListItem ), ( pxNewTCB
     )->xTaskPeriod + currentTick);

  prvAddTaskToReadyList( pxNewTCB );

    ...

}
```

The IDLE task management is modified as well. The initialization of the IDLE task happens in the $vTaskStartScheduler()$ method, that starts the real time kernel tick processing and initialize all the scheduler structures. Since FreeRTOS specifications want a task in execution at every instant, a correct management of the IDLE task is fundamental. With the standard FreeRTOS scheduler, the IDLE task is a simple task initialized at the lowest priority. In this way it would be scheduled only when no other tasks are in the ready state. With the EDF scheduler, the lowest priority behaviour can be simulated by a task having the farest deadline. $vTaskStartScheduler()$ method initializes the IDLE task and inserts it into the Ready List. The method is modified as follow:

```
       /*E.C. : */
       #if (configUSE_EDF_SCHEDULER == 1)
      {
         tickType initIDLEPeriod = 100;
        xReturn = xTaskCreatePeriodic( prvIdleTask, "IDLE", tskIDLE_STACK_SIZE, (
           void * ) NULL, ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ), NULL,
           initIDLEPeriod );
      }
```

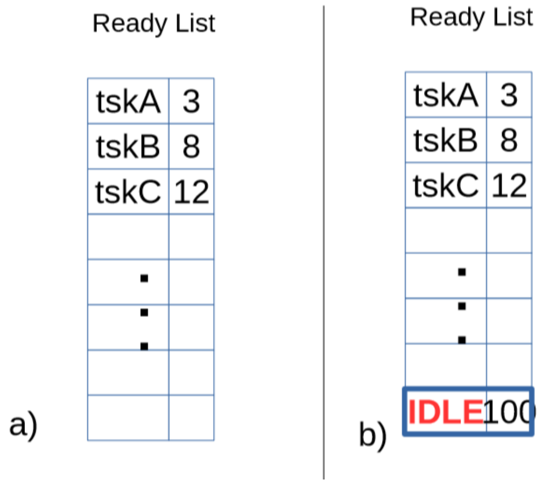

Figure 3.5: a) tskA, tskB and tskC are created before the sceduler is started; b) $vTaskStartScheduler()$ method is called, the real time kernel tick processing starts and the IDLE task is added at the last position of the Rady List. It will execute only when no other tasks are in ready state.

```
        #else
          /* Create the idle task without storing its handle. */
          xReturn = xTaskCreate( prvIdleTask, "IDLE", tskIDLE_STACK_SIZE, ( void * )
              NULL, ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ), NULL );
//      #endif
```

The IDLE task is initialized with a period of $initIDLEPeriod = 100$. We assume that no task can have a period greater than $initIDLEPeriod$: in this way, when the IDLE task is added to the Ready List, it will be at the last position of the list, since its deadline will be greater than any other task ($TASK_{deadline} = tick_{cur} + TASK_{period}$, with $tick_{cur} = 0$ and $IDLE_{period} = initIDLEPeriod$ greater than any other task period). Every time IDLE task executes (i.e. no other tasks are in the Ready List), it calls a method that increments its deadline in order to guarantee that IDLE task will remain in the last position of the Ready List.

Last change needed involves the switch context mechanism. Every time the running task is suspended, or a suspended task with an higher priority than the running task awakes, a switch context occurs. $vTaskSwitchContext()$ method is in charge to update the $*pxCurrentTCB$

pointer to the new running task:

```
void vTaskSwitchContext( void ){
    ...

    /* E.C. : */
    #if (configUSE_EDF_SCHEDULER == 0)
    {
      taskSELECT_HIGHEST_PRIORITY_TASK();
    }
    #else
    {
      pxCurrentTCB = (TCB_t * ) listGET_OWNER_OF_HEAD_ENTRY( &(
          xReadyTasksListEDF ) );
    }
    #endif
    ...
}
```

taskSELECT_HIGHEST_PRIORITY_TASK() method is replaced in order to assign to *pxCurrentTCB* the task at the first place of the new Ready List.

Now we have all the pieces to get the new EDF scheduler work. In the next section will be analyzed a scheduling example in order to show how these changes work all together.

3.2.3 Scheduling Example

Two scheduling example are shown: the first refers to a preemptive behaviour where the OS interrupts the execution flow of the running task and assign the CPU to another task; the second example instead shows a cooperative situation where the running task finishes its execution and leaves the running state.

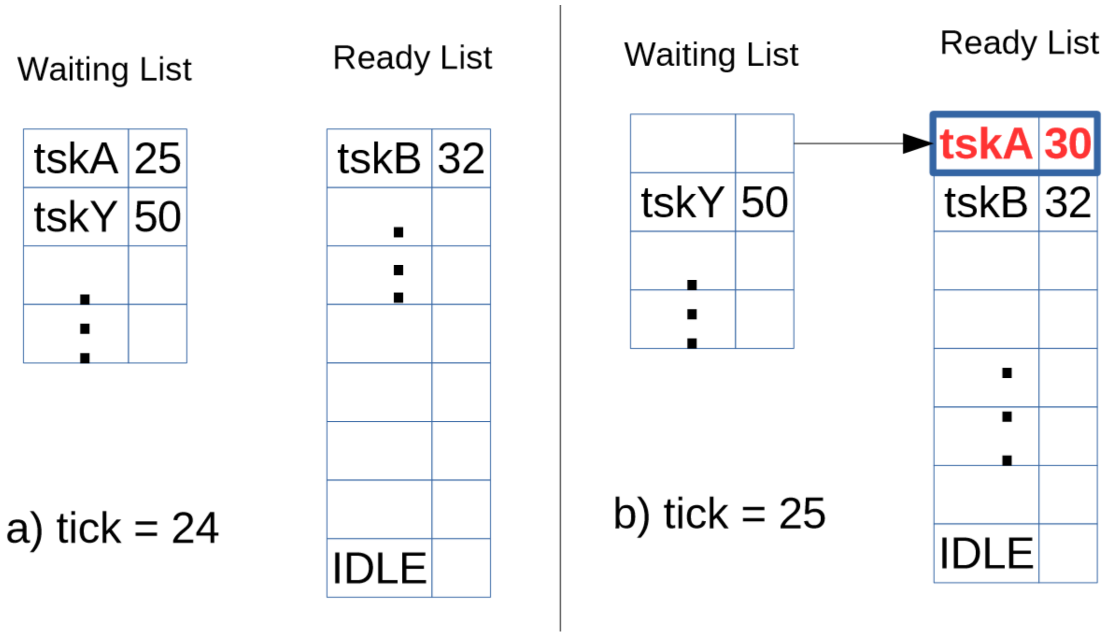

Figure 3.6: a) tskA will be awaken at the next tick. tskB has the nearest deadline among the tasks in ReadyList, so it's in the head position. IDLE is the task with the farest deadline. b) After the tick interrupt happened, tskA moves to the Ready List: $tskA_{deadline} = currentTick + tskA_{period} = 25 + 5 = 30$

For the preemptive example, let tskA and tskB be tasks of capacity 2 and period 5 and 8 respectively. Let's consider the situation described in Figure 3.6-a: $tickCount = 24$, tskB is executing and tskA is waiting tick 25 to wake up. When the tick interrupt occurs, the ISR $vPortYeldFromTick()$ is called. As shown in Cap.1, this interrupt service routine saves the running task context, calls the $xTaskIncrementTick()$ method, and if same tasks are wake up from the Waiting List, performs a context switch and restores the context of the new running task. In Figure 3.7-b the tick interrupt has occurred. tskB context is saved ($portSAVE\_CONTEXT()$), then $XTaskIncrementTick()$ method is called:

-tickCount variable is incremented ($tickCount = 25$);

-tskA TCB is removed form xDelayedTaskList;

-tskA's GenericListIteam is set to tskA's new deadline (currentTick + tskA period);

-tskA is insered in $xReadyTasksListEDF$ by callyng the $addTaskToReadyList()$ method (since tskA sdeadline is closer than tskB deadline, tskA is added at the head of the Ready List);

because at least one task has been awakened, $vTaskSwitchContext()$ method is called, so the $*pxCurrentTCB$ pointer points to tskA; $portRESTORE\_CONTEXT()$ restores the context

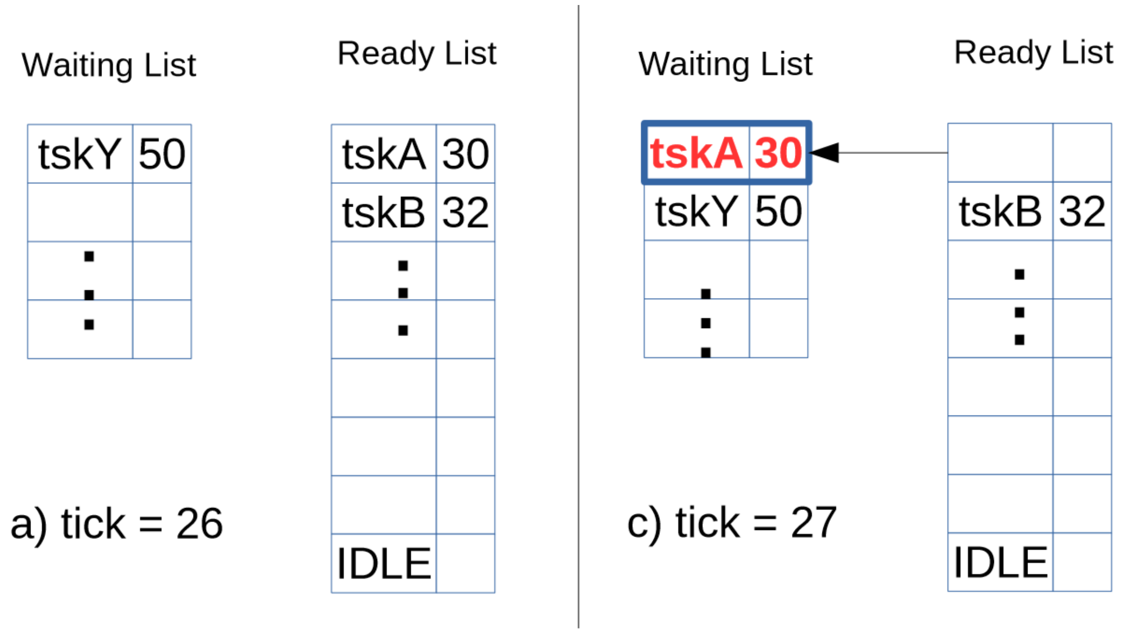

Figure 3.7: Collaborative example- a)description of the Waiting List and Ready List at tick=26 b) when tick=27, tskA moves in the Waiting List and tskB became the executing task

of the task pointed by $*pxCurrentTCB$, so from now tskA is executing.

For the collaborative example let's consider the same tasks upon. At $tickCount = 26$ the situation is as described in Figure 3.7-a: tskA is running and tskB is in ready state. As shown in Figure 3.7-b, at $tickCount = 27$ tskA finishes its execution and goes in waiting state till its next periodic awakening by calling $delayTaskUntill()$ method, as shown in Cap.2:

-tskA TCB is removed from $xReadyTaskListEDF$;

-tskA GenericListIteam is set to the next awake time;

-tskA TCB is inserted in $xDelayedTaskList$;

-$portYELD\_WITH\_API()$ method is called: this method force a context switch, so tskA context is saved ($portSAVE\_CONTEXT()$), $vTaskSwitchContext()$ makes $*pxCurrentTCB$ pointing the task TCB on the head of $xReadyTaskListEDF$, i.e. tskB, then $portRESTORE\_CONTEXT$ restore tskB context.

Figure 3.6 and Figure 3.7

3.3 Demo application description

In this section we will describe the demo application used to test the EDF scheduler. The application creates two tasks, task A (A_{period}, $A_{capacity}$), and task B (B_{period}, $B_{capacity}$). The job of the two tasks is to keep the CPU utilization for $A_{capacity}$ and $B_{capacity}$ system tick every A and B period respectively.

```

/* Standard includes. */
#include "stdio.h"
#include "main.h"

//-----------------------------------------------
// Tasks Protopies
//-----------------------------------------------

void TSK_A (void *pvParameters);
void TSK_B (void *pvParameters);

//-----------------------------------------------
// Global Variables
//-----------------------------------------------

#define CAPACITY 3    //cpu time in tick
#define A_PERIOD 5    //task A period
#define B_PERIOD 8    //task B period
```

First, task prototypes are declared, and CAPACITY, A_PERIOD, and B_PERIOD variables are defined.

```

//---------------------------------------------
```

```
// Start point
//-------------------------------------------------

int main(void)
{

  SystemInit();

  xTaskPeriodicCreate( TSK_A, ( const char * ) "A",
                       configMINIMAL_STACK_SIZE, NULL,
                       1, NULL, A_PERIOD );
  xTaskPeriodicCreate( TSK_B, ( const char * ) "B",
                       configMINIMAL_STACK_SIZE, NULL,
                       1, NULL, B_PERIOD );

  // FreeRTOS Scheduler starten
  vTaskStartScheduler();

  // wird nie erreicht!!
  while(1)
  {

  }
}
```

SystemInit() is a method from the native layer of FreeRTOS, and is needed to initialize the board. Then, task A and B are created calling *xTaskPeriodicCreate*(), and A_PERIOD and B_PERIOD are set as task A and B periods. *vTaskStartScheduler*() activates the EDF scheduler: from now the Ready List contains three tasks ready to be skeduled: A, B and IDLE.

```
//--------------------------------------------------------------
// Task A:
//--------------------------------------------------------------

void TSK_A (void *pvParameters)
{
  TickType_t xLastWakeTimeA;
  const TickType_t xFrequency = A_PERIOD; //tsk A frequency
  volatile int count = CAPACITY;      //tsk A capacity

  // Initialise the xLastWakeTime variable with the current time.
  xLastWakeTimeA = 0;

  while(1)
  {
    TickType_t xTime = xTaskGetTickCount ();

    TickType_t x;
    while(count != 0)
    {
      if(( x = xTaskGetTickCount () ) > xTime)
      {
        xTime = x;
      }
    }

    count = CAPACITY;


    // Wait for the next cycle.
    vTaskDelayUntil( &xLastWakeTimeA, xFrequency );
```

```


    }

}
```

Task A code is shown upon: it simulates the utilization of the CPU for t=CAPACITY system ticks (it enters in a while loop until count variable, initialized at CAPACITY value, reaches zero) then calls the $vTaskDelayUntil()$ method and goes in $xDelayedTaskList$, where waits A_PERIOD system ticks before be awakened. Task B works in the same way.

In the next section the test method will be described.

3.4 Tests and Results

3.4.1 Trace macros

To test the correctness of the implemented EDF scheduler, we execute two tasks whose EDF scheduling sequence is known, and match the run-time scheduling sequence with the expected one. To obtain correct EDF scheduling sequences to match, we used Cheddar[7]: it is a free real-time scheduling tool developed by University of Brest that performs scheduler simulation. To monitor the run-time scheduling sequence, FreeRTOS offers special trace functions. As reported in the official guide:

"Trace macros are a very powerful feature that permit you to collect data on how your embedded application is behaving. Key points of interest within the FreeRTOS source code contain empty macros that an application can re-define for the purpose of providing application specific trace facilities. The application need only implement those macros of particular interest - with unused macros remaining empty and therefore not impacting the application timing."

We implemented three trace macros:

-*traceTASK_SWITCHED_OUT*() is called every time a task switch out;

-in the same way, *traceTASK_SWITCHED_IN*() is called every time a task switches in;

-*traceTASK_DELAY_UNTIL*() is called when the running task suspends itself by calling *delayTaskUntill*();

monitoring context switch events, we know which task is running at every time, and we see if a context switch occurred because a task suspended itself or because the system suspended it in a preemptive way. Below the implementation of these trace macros is shown (they are defined in the FreeRTOSConfig.h file):

```

//E.C. : MACROS
#define traceTASK_SWITCHED_OUT() { char name[20];
                                   getTaskName(name);
                                   printf("Task Out: %s\n", name );
                                   }
#define traceTASK_SWITCHED_IN() { char name[20];
                                  getTaskName(name);
                                  printf("Task IN: %s\n", name );
                                  }
#define traceTASK_DELAY_UNTIL() { char name[20]; getTaskName(name);
                                  printf("Task Delay: %s, ", name );
                                   }
```

The implemented trace methods print a string containing the event occurred in the output buffer (we will see how configure it in the ext section).

Another information we need is the tick time at witch these events occur. like trace macros, FreeRTOS makes available a callback function called every time the tick interrupt executes. *vApplicationTickHook*() function is defined in the Defaults_IDLE.c file:

```

//-------------------------------------------------------------
```

```
// TICK
//--------------------------------------------------------------
void vApplicationTickHook( void )
{
    /* vApplicationTickHook() will only be called if configUSE_TICK_HOOK is set
    to 1 in FreeRTOSConfig.h. It is a hook function that will get called during
    each FreeRTOS tick interrupt. Note that vApplicationTickHook() is called
    from an interrupt context. */
   printf("TICK : %d\n",(int)xTaskGetTickCount());
}
```

It prints the tick number just occurred to the output buffer.

3.4.2 Semiosting

In order to let the board print debug messages to the IDE console we use Semihost technique. As described in the ARM Software Development Tools Guide[8]:

"Semihosting is a mechanism for ARM targets to communicate input/output requests from application code to a host computer running a debugger. This mechanism could be used, for example, to enable functions in the C library, such as printf() and scanf(), to use the screen and keyboard of the host rather than having a screen and keyboard on the target system. This is useful because development hardware often does not have all the input and output facilities of the final system. Semihosting enables the host computer to provide these facilities. Semihosting is implemented by a set of defined software instructions (SVCs) that generate exceptions from program control. The application invokes the appropriate semihosting call and the debug agent then handles the exception. The debug agent provides the required communication with the host." (Figure 3.8)

CooCox IDE implements Semiosting[9]: the main project folder contains two sub-folders, *semihosting* and *stdio* that contains all files needed. file *printf.c* contained in *stdio* folder

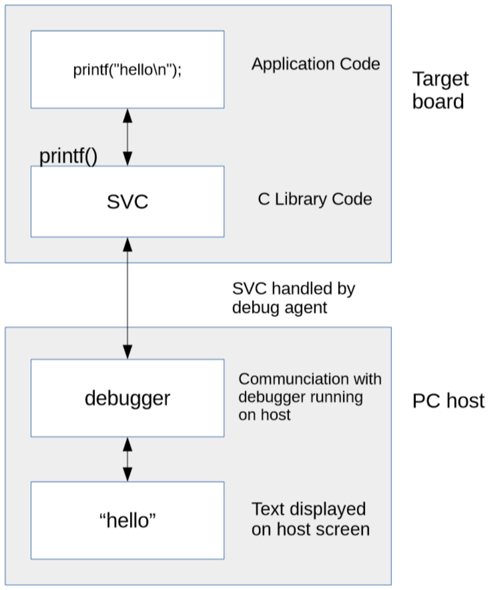

Figure 3.8: Semihosting structure

gives a custom implementation of printf() method that reduces the memory footprint of the binary, compared to the libc implementation. *semihosting.h* file contained in *semihost* folder must be included in the FreeRTOS.h configuration file.

Now we are ready to start the test: the test application executes a couple of tasks, and the IDE console shows the run-time Log information we set.

3.4.3 Test and Results

In the first test we consider task A and task B reported in the next table:

	T	C
Task A	5	2
Task B	8	2

$U = \frac{2}{5} + \frac{2}{8} = 0,65 < 1$. According with theorem 3.1, EDF algorithm can schedule them without missing any deadline. both tasks starts from tick=0. Figure 3.9 is obtained with Cheddar software and describes the correct EDF schedule of task A and B for a single processor preemptive system.

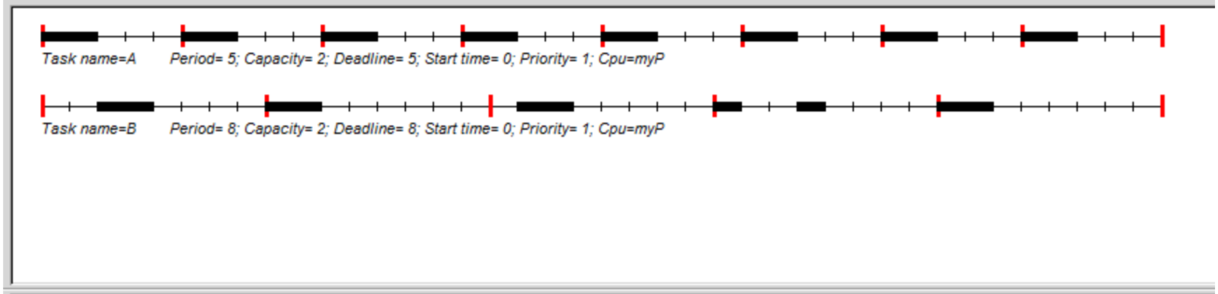

Figure 3.9: EDF schduling of task A and task B

```
///////////////////////////////
TICK : 1
TICK : 2
Task Delay: A, Task Out: A
Task IN: B
TICK : 3
TICK : 4
Task Delay: B, Task Out: B
Task IN: IDLE
TICK : 5
Task Out: IDLE
Task IN: A
TICK : 6
TICK : 7
Task Delay: A, Task Out: A
Task IN: IDLE
TICK : 8
Task Out: IDLE
Task IN: B
TICK : 9
TICK : 10
Task Out: B
Task IN: A
TICK : 11
TICK : 12
Task Delay: A, Task Out: A
Task IN: B
Task Delay: B, Task Out: B
Task IN: IDLE
TICK : 13
TICK : 14
TICK : 15
Task Out: IDLE
Task IN: A
TICK : 16
Task Out: A
Task IN: A
TICK : 17
Task Delay: A, Task Out: A
Task IN: B
TICK : 18
TICK : 19
Task Delay: B, Task Out: B
Task IN: IDLE
TICK : 20
Task Out: IDLE
Task IN: A
TICK : 21
TICK : 22
Task Delay: A, Task Out: A
Task IN: IDLE
TICK : 23
TICK : 24
Task Out: IDLE
Task IN: B
TICK : 25
Task Out: B
Task IN: A
TICK : 26
TICK : 27
Task Delay: A, Task Out: A
Task IN: B
TICK : 28
Task Delay: B, Task Out: B
Task IN: IDLE
TICK : 29
TICK : 30
Task Out: IDLE
Task IN: A
TICK : 31
TICK : 32
Task Out: A
Task IN: A
Task Delay: A, Task Out: A
Task IN: B
TICK : 33
TICK : 34
Task Delay: B, Task Out: B
Task IN: IDLE
TICK : 35
Task Out: IDLE
Task IN: A
TICK : 36
TICK : 37
Task Delay: A, Task Out: A
Task IN: IDLE
TICK : 38
TICK : 39
TICK : 40
```

Figure 3.10: EDF Log of task A and task B

Figure 3.10 shows the Log file obtained by the execution of the demo application, where $A\_PERIOD = 5$, $B\_PERIOD = 8$, and $CAPACITY = 2$; The schedule sequence of task A, task B and IDLE reflects correctly the EDF schedule sequence of Figure 3.9. Only one preemptive context switch occurs (tick=25), and the algorithm is able to handle it properly.

In the second test we consider task A and B described in the table:

	T	C
Task A	5	3
Task B	8	3

$U = \frac{3}{5} + \frac{3}{8} = 0,975 < 1$, the CPU load is higher than first example, but still under the schedulable limit, so no deadline should be missed. Both tasks starts from $tick = 0$. As the first example, Figure 3.11 is obtained with Cheddar software and describes the correct EDF schedule for task A and B. The demo application is set whith the followinf params: $A\_PERIOD = 5$, $B\_PERIOD = 8$, and $CAPACITY = 3$; Respect to the previous example test, the new task configuration needs more preemptive context switch, but the scheduler works as expected.

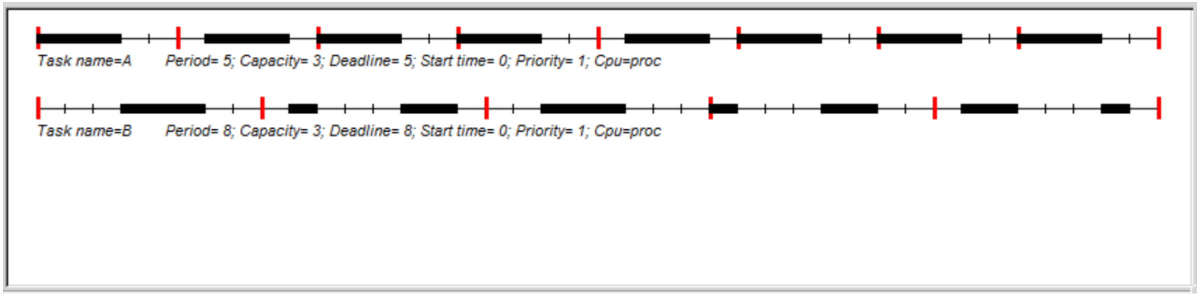

Figure 3.11: EDF scheduling of task A (T=5, C=3) and task B (T=8, C=3).

```
///////////////////////////////
TICK : 1
TICK : 2
TICK : 3
Task Delay: A, Task Out: A
Task IN: B
TICK : 4
TICK : 5
Task Out: B
Task IN: B
TICK : 6
Task Delay: B, Task Out: B
Task IN: A
TICK : 7
TICK : 8
Task Out: A
Task IN: A
TICK : 9
Task Delay: A, Task Out: A
Task IN: B
TICK : 10
Task Out: B
Task IN: A
TICK : 11
TICK : 12
TICK : 13
Task Delay: A, Task Out: A
Task IN: B
TICK : 14
TICK : 15
Task Out: B
Task IN: B
Task Delay: B, Task Out: B
Task IN: A
TICK : 16
Task Out: A
Task IN: A
TICK : 17
TICK : 18
Task Delay: A, Task Out: A
Task IN: B
TICK : 19
TICK : 20
Task Out: B
Task IN: B
TICK : 21
Task Delay: B, Task Out: B
Task IN: A
TICK : 22
TICK : 23
TICK : 24
Task Out: A
Task IN: A
Task Delay: A, Task Out: A
Task IN: B
TICK : 25
Task Out: B
Task IN: A
TICK : 26
TICK : 27
TICK : 28
Task Delay: A, Task Out: A
Task IN: B
Task Out: B
Task IN: B
Task Delay: B, Task Out: B
Task IN: A
TICK : 31
TICK : 32
Task Out: A
Task IN: A
TICK : 33
Task Delay: A, Task Out: A
Task IN: B
TICK : 34
TICK : 35
Task Out: B
Task IN: B
TICK : 36
Task Delay: B, Task Out: B
Task IN: A
TICK : 37
TICK : 38
TICK : 39
Task Delay: A, Task Out: A
Task IN: IDLE
TICK : 40
```

Figure 3.12: EDF Log of task A and task B

Chapter 4

LLREF Scheduler

4.1 LLREF Algorithm

LLREF (Largest Local Remaining Execution First)[4] is an algorithm which is used to schedule periodic task sets in multiprocessor preemptive systems. Full migration across processors is required: jobs are allowed to arbitrarily migrate across processors during their execution, as long as the same task is not executed parallelly on more than one processor[10].

We consider a set of periodic tasks, denoted $\tau = (T_1, T_2, ..., T_N)$, and a set of m symmetric processors available in the system. Tasks are assumed to arrive periodically at their release times r_i. Each task T_i has an execution time c_i, and a deadline d_i which is the same as its period p_i. The utilization u_i of a task T_i is dened as c_i/d_i and is assumed to be less than 1.

We assume that tasks may be preempted at any time, and are independent, i.e., they do not share resources or have any precedences. We consider a non-work conserving scheduling policy: thus processors may be idle even when tasks are present in the ready queue.

LLREF can be prooven to be an optimal schedule algorithm. All tasks meet their deadlines when the total utilization demand is smaller or equal with the utilization capacity of the platform:

$$U = \sum_{i=1}^{N} u_i \leq \quad m.$$

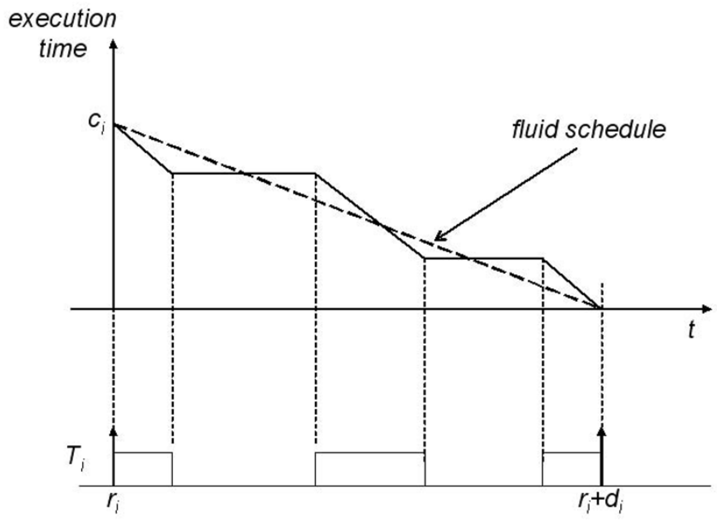

Figure 4.1: Task fluid diagram

LLREF algorithm is based on an abstraction which is known as Time and Local Plane (T-L Plane). This abstraction determines when a task must be scheduled in order to meet its deadline.

4.1.1 T-L Plane

Figure 4.1 illustrates the fundamental idea behind the T-L plane. For a task T_i the figure shows a plane: the x-axis represents the time, and the y-axis represents the tasks remaining execution time. When T_i runs like in Figure 4.1, for example, its execution can be represented as a broken line between (0,c_i) and (d_i,0). In the plane, task execution is represented as a line whose slope is -1, since x and y axes are in the same scale, while the non-execution is represented as a zero slope line.

Figure 4.2 shows how to construct fluid schedules for N tasks. For each task let's consider the right isosceles triangle found between every two scheduling events; then, let's overlap the N triangles between every two consecutive scheduling events (one for each task). We call this as the T-L Plane TL^k, where k is simply increasing over time. Figure 4.3 analyzes in detail the generic TL^k plane. The bottom side of the triangle represents time. The vertical side represents

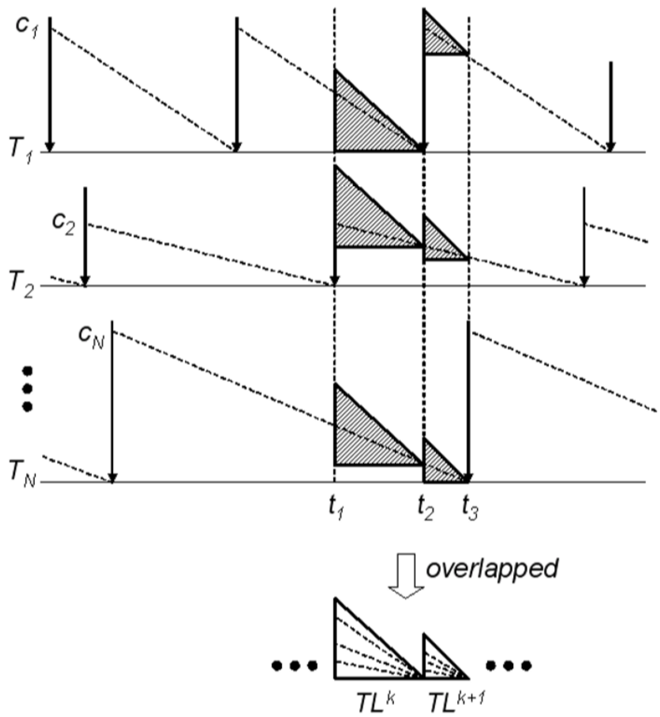

Figure 4.2: T-L plane

the axis of the tasks remaining execution time, which is called *local remaining execution time* l_i, which is supposed to be consumed before each TL_k plane ends. The status of each task is represented as a token in the TL plane. the x coordinate of the token describes the current time, while the y coordinate describes the tasks local remaining execution time l_i (it is the execution time the task must consume until the time ${t_k}^f$, and not the tasks deadline).

Each tasks token moves in the T-L plane. Tokens are only allowed to move in two directions: when the task is executing, the respective token moves diagonally down, as T_N moves in Figure 4.3; otherwise, it moves horizontally, as T_1 moves. In a m processors system, no more than m tokens can move diagonally down together. The scheduling objective in the k^{th} T-L plane is to make all tokens arrive at the rightmost vertex of the T-L plane, with all tasks having $l_i = 0$ before ${t_k}^f$. If all tokens are made locally feasible at each T-L plane, they are possible to be scheduled throughout every consecutive T-L planes over time. An important parameter for the tasks in the T-L plane is their *local laxity*, defined for the generic task T_i as: ${t_f}^k - t_{curr} - l_i$. The oblique syde of the T-L plane has an important meaning: when a token hits that side, it

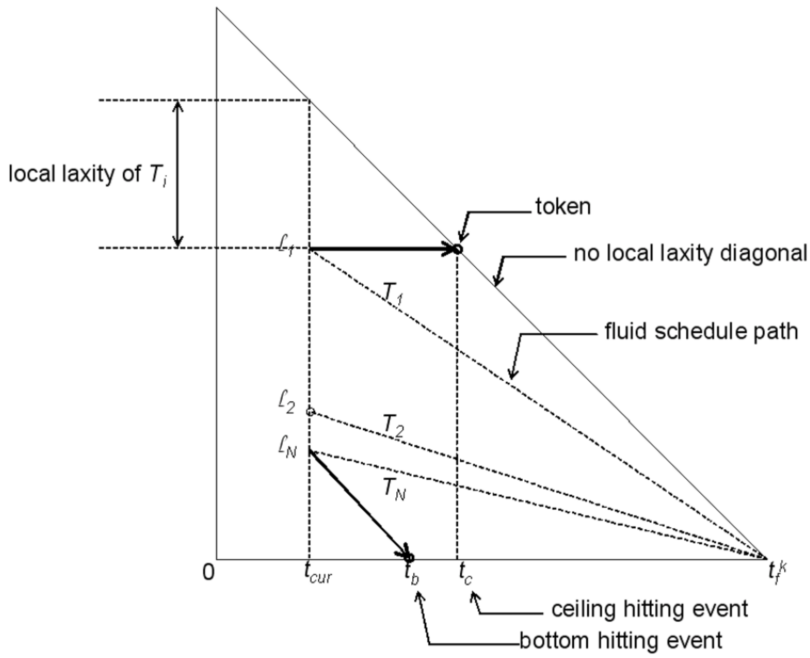

Figure 4.3: Multiple T-L plane

implies that the task does not have any local laxity: thus, if it is not executed immediately, it will not be able to satisfy the scheduling objective of local feasibility.

4.1.2 Scheduling in T-L planes

LLREF is based on two types of scheduling events:

-**bottom-hitting event B** - if a token hits the horizontal line, it means that the task is already executed as long as necessary for this T-L plane, so it is turn of another task to select instead;

-**celling hitting event C** - when a task has zero remaining local laxity, the token hits the diagonal line which means that the task needs to be selected immediately to meet the local deadline.

LLREF pseudo-code function algorithm is shown in Figure 4.4, and it is called every time a schedule event occurs (B, C events and when a task is added to the Ready list).l_i of each task is assumed to be updated before the algorithm starts. When it is invoked, $sortByLLREF$ sorts tokens in the order of largest local remaining execution time and selects m tasks to dispatch to processors.

Algorithm 1: LLREF

Input : **T**=$\{T_1,...,T_N\}$, ζ_r: Ready queue, M:# of processors
Output : array of dispatched tasks to processors

```
sortByLLRET( ζr );
{T1, ..., TM} = selectTasks( ζr );
return {T1, ..., TM};
```

Figure 4.4: LLREF pseudo-code algorithm

4.2 Implementation in FreeRTOS

4.2.1 General Idea

As for the EDF algorithm, our LLREF implementation in FreeRTOS uses the existing structures that the OS already offers and brand new structures specially created. In this section we describe the algorithm implementation, from the design till the code.
Since we work on a single processor CPU, our LLREF implementation concerns the special case where $m = 1$.

The general idea is to implement a new Ready List, where tasks are ordered by their local remaining execution time $l_i = u_i * \Delta_k$. The task in the head of the list is the running task. When a task finishes its execution, moves to the Waiting List where remain until its next awake time.
Two new functions control the T-L planes management:

- **funcNewTLPlane()** - every time a task moves to Ready List, a new T-L plane starts: for ach ready task the local remain execution time is updated, and the Ready List is sorted;

```
funcNewTLPlane(){
    -calc next time arrival;
    -initialize tasks local remain execution time;
    -Ready List sorting;
```

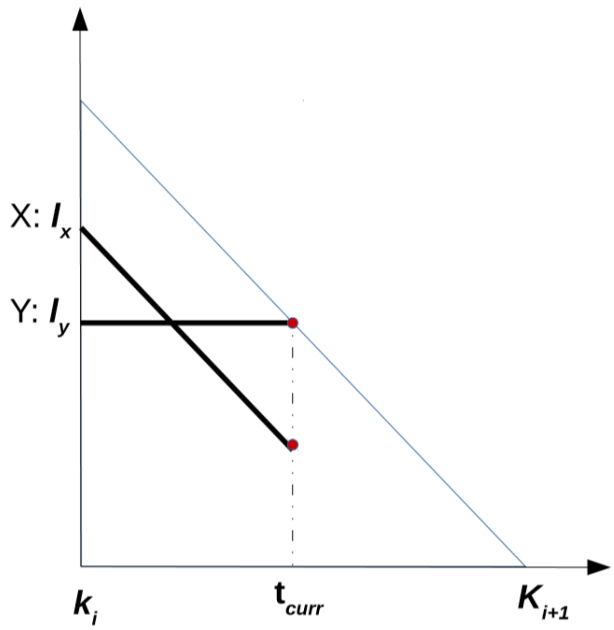

Figure 4.5: T-L plane: $m = 1$

```
        -calc next B/C event time;

        -context switch;


    }
```

- **funcEventHandle()** - every time a schedule B or C event occurs, l_i is updated and the Ready List sorted;

```
    funcEventHandle(){

        -update tasks local remain execution time;

        -Ready List sorting;

        -calc next B/C event time;

        -context switch;


    }
```

Since $m = 1$, in the T-L plane only one token per time will move diagonally down. All the other token will move horizontally. Let's consider the situation in Figure 4.5:

at the beginning of the T-L plane X is the token with the highest local remaining execution time, and Y is the token having the second highest one; B events can only be generated by the running task (X in the example) hitting the bottom line, otherwise C events can only occurs because of the second highest local remaining execution token (Y in the example) hitting the diagonal line. But for $m = 1$ happens as stated in the theorem below:

Theorem 4.1 *If $m = 1$, C events can not occur in a feasible task set.*

Indeed, when a C event occurs, the task executing get suspended, and the task whose token hit the diagonal line obtains the CPU usage till the T-L plane end- that is, the suspended task will never be able to finish its local execution in the T-L plane.

So, every time a new T-L plane starts, the Ready List is sorted and the task in the head of the list is executed, and the next B event time t_B is calculated: the running task will execute until t_B, then, a context switch will occur. But how to calculate t_B?

$$t_B = l_i = \mu_X * \Delta_K,$$

with X being the running task and $\Delta_K = k_{i+1} - k_i$; it means that when a T-L plane starts at time k_i, we have already to know the time the next T-L plane will start- that is- we have to know the next time a task will be inserted to the Ready List from the Waiting List. The awake time of the task in the head of the Waiting List is not sufficient, as the example in Figure 4.6 shows: task B period is two times task A period, so when B is inserted in the Ready List, the next insertion in the Ready List will be still a task B insertion; so, we have to consider both the next wake time w_{time} from the Waiting list and the running task A next release time:

$$k_{i+1} = min w_{time}, k_i + X_{period}.$$

Now we have all the information initialize a T-L plane. The $funcEventHandle()$ function is in charge to update tasks local remain execution time. How to do that? First, we observe that

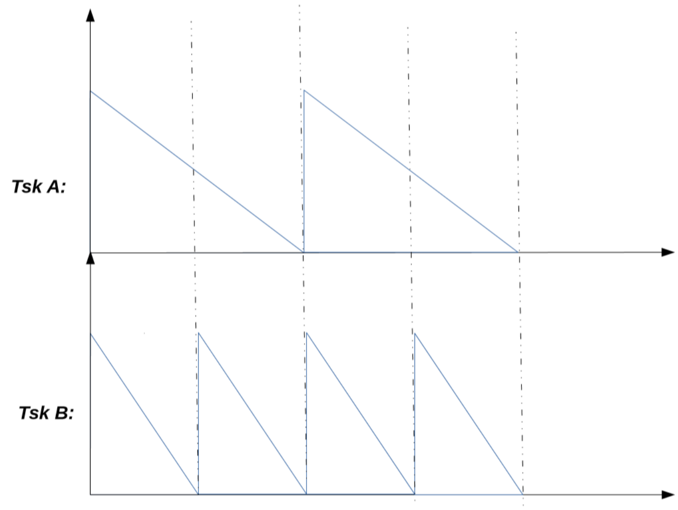

Figure 4.6: Task A period is two times task B period.

only the running task will decrease its local remain execution time. So, it suffices to update only the local remain execution time of the running task A. How to do that?

$$l_A = l_A - \Delta,$$

where Δ is the time spent since the previous event time $t_{previousEvent}$: $\Delta = currentTime - t_{previousEvent}$.

So, we have to calculate the time the next event will occur, and we have to memorize the time the last event occurred.

4.2.2 IDLE Task management

4.3 Code implementation

This section contains implementation details of the proposed EDF scheduler. Example code will be shown, and same architectural project choice will be explained. Every time FreeRTOS code is reported, it refers to 8.2.2 version.

All changes that will be illustrated refer to *tasks.c* file, since scheduler structures and methods are contained there. According with the FreeRTOS style guideline, a configuration variable, *configUSE_LLREF_SCHEDULER*, is added to the *FreeRTOS.h* config file. When *configUSE_LLREF_SCHEDULER* is set to 1, EDF scheduler is used, elsewhere the OS uses the original scheduler.

First, a new set of system variables are introduced:

```
/* Other file private variables. --------------------------------*/
PRIVILEGED_DATA static volatile TickType_t xTLPlaneStart = ( TickType_t ) 0U;
PRIVILEGED_DATA static volatile TickType_t xTLPlaneEnd = ( TickType_t ) 0U;
PRIVILEGED_DATA static volatile TickType_t nextEvetnTick = ( TickType_t ) 0U;
PRIVILEGED_DATA static volatile TickType_t lastEvetnTick = ( TickType_t ) 0U;
```

-**xTLPlaneStart** saves the tick time at which a new T-L plane starts;

-**xTLPlaneEnd** saves the tick time at which a T-L plane ends;

-**nextEventTick** stores the tick time at which the next B event will occurs; -**lastEventTick** stores the tick time at which the last schedule event occurred;

As we described in the previous section, in the Ready List tasks are sorted by their local remaining execution time l_i; at the beginning of the generic K h T-L plane, it is initialized as: $l_i = \mu_i * \Delta_k$ for all the ready tasks. So, each task must memorize its period p and its capacity c, so $c/p = \mu$ can be calculated. *xTaskPeriod* and *xTaskCapacity* variables are added to the Task Control Block structure:

```
  /* E.C. : the period of a task */
  #if ( configUSE_LLREF_SCHEDULER == 1 )
    TickType_t xTaskPeriod;      /*< Stores the period in tick of the task. > */
    TickType_t xTaskCapacity;    /*< Stores the capacity in tick of the task. > */
  #endif
```

In order to initialize *xTaskPeriod* and *xTaskCapacity*, a new task initialization function is created as well:

```
/*E.C. : */
```

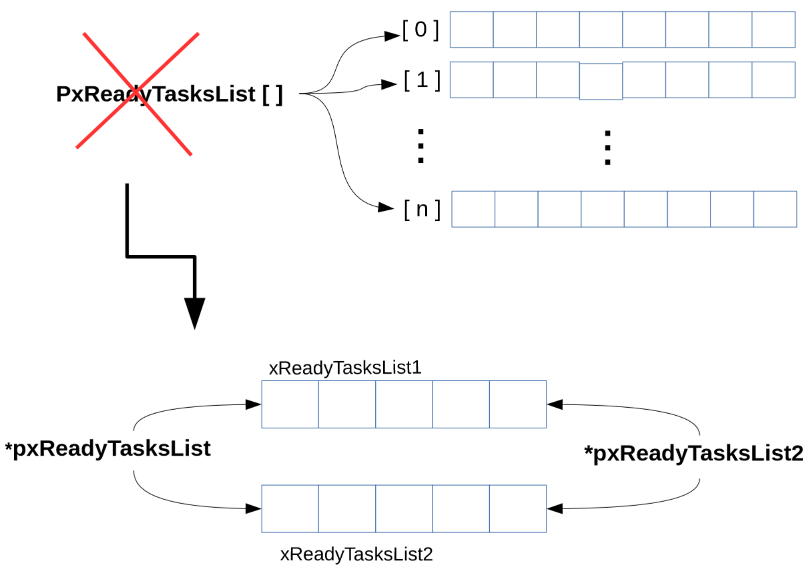

Figure 4.7: The new Ready List implementation.

```
BaseType_t xTaskLLREFCreate( < param > ,TickType_t period, TickType_t capacity )
{ ...
    /*E.C. : initialize the period */
   pxNewTCB->xTaskPeriod = period;
   /*E.C. : initialize the capacity */
   pxNewTCB->xTascCapacity = capacity;
    ...
   /*E.C. : generic list iteam is initializated as 0: */
   listSET_LIST_ITEM_VALUE( &( ( pxNewTCB )->xGenericListItem ), ( pxNewTCB )->0);
   prvAddTaskToReadyList( pxNewTCB );
    ...
}
```

The new Ready List is implemented as a simple list ordered by tasks local remain execution time.Figure 4.7 shows the new Ready List implementation. For implementation reasons, as we will see, two lists are used: when a new T-L plane starts, the tasks inserted in the Ready List in use are removed from it, their local remain execution time is initialized and then are added to the second Ready List, sorted by their local remain execution time. $*pxReadyTaskListLLREF$ pointer refers o the Ready list in use, $*pxReadyTaskListLLREF2$ refers to the Ready List that will be used in the next T-L plane:

```
/* E.C. : the new RedyList */
#if ( configUSE_LLREF_SCHEDULER == 1 )
   PRIVILEGED_DATA static List_t xReadyTasksListLLREF1;
   PRIVILEGED_DATA static List_t xReadyTasksListLLREF2;
   PRIVILEGED_DATA static List_t * volatile pxReadyTaskListLLREF;
    PRIVILEGED_DATA static List_t * volatile pxReadyTaskListLLREF2;
#endif
```

Then, the *prvInitialiseTaskLists*() function, that initialize all task lists at the creation of the first task, is modified adding the initialization of *xReadyTasksListLLREF*:

```
static void prvInitialiseTaskLists( void )
{    ...
   /* E.C. */
   #if ( configUSE_LLREF_SCHEDULER == 1 )
   {
      vListInitialise( &xReadyTasksListLLREF1 );
      vListInitialise( &xReadyTasksListLLREF2 );

      /* Start with pxReadyTaskListLLREF using list1 and the pxReadyTaskList2 using
          list2. */
       pxReadyTaskListLLREF = &xReadyTasksListLLREF1;
       pxReadyTaskListLLREF2 = &xReadyTasksListLLREF2;
   }
   #endif
    ...
}
```

prvAddTaskToReadyList() method that adds a task to the Ready List is then modified as follows (it is assumed that *xGenericListItem* contains the local remain execution time):

```
    /*
 * Place the task represented by pxTCB into the appropriate ready list for
 * the task. It is inserted at the end of the list.
 */
#if configUSE_EDF_SCHEDULER == 0 /* E.C. : */
   #define prvAddTaskToReadyList( pxTCB )                                     \
                                \
      vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( ( pxTCB
          )->xGenericListItem ) )
#else
   #define prvAddTaskToReadyList( pxTCB ) /*xGenericListIteam must contain the
       local remain execution time */  \
      vListInsert( &(pxReadyTaskListLLREF), &( ( pxTCB )->xGenericListItem ) )
#endif
```

two new function are created: $functNewTLPlane()$ and $funcEventHandle()$, as described in the previus section; $functNewTLPlane()$ implementation is here described:

```
/*
 * Called at every xTLPlaneStart tick
 */
void functNewTLPlane()
{
    /*calc the T-L plane end:*/
    xTLPlaneEnd = min( xTLPlaneStart + ( pxCurrentTCB )->xTaskPeriod, /*the next
        arrival time of the running task*/
                        xNextTaskUnblockTime );

    /*initialize local execution time left for the tasks in Ready List and sort
        it:*/
    initializeAndSortLLREF();
```

```

    /*update last event time:*/
    lastEventTick = nextEventTime;

    /*calc next event B time:*/
    TCB_t pxTCB = ( TCB_t * ) listGET_OWNER_OF_HEAD_ENTRY( xReadyTasksListLLREF );
    TickType_t u = (TickType_t )listGET_LIST_ITEM_VALUE( &( pxTCB->xGenericListItem
        ) );
    nextEvetnTick = xTLPlaneStart + ( u * (xTLPlaneEnd-xTLPlaneStart) );

    /*force context switch:*/
    portYELD_WITH_CONTEXT();
}
```

initializeAndSortLLREF() function used above initializes local remaining execution time for the tasks in the actual Ready list, and remove the from it and then add them in the other Ready List: *insertTasktoReadyList*() function preserves the sorted order of the list in which task are inserted:

```
void initializeAndSortLLREF()
{
    List_t *pxTemp;

   /* The delayed tasks list should be empty when the lists are switched. */
   configASSERT( ( listLIST_IS_EMPTY( pxReadyTaskListLLREF2 ) ) );
   while( listLIST_IS_EMPTY( pxReadyTaskListLLREF )
   {
       TCB_t *pxTCB;
       uxListRemove( &( pxTCB->xGenericListItem );

       /*update the task local remaining execution time:*/
```

```
        int tmp = ( ( pxtTCB )->xTaskPeriod / ( pxtTCB )->xTaskCapacity ) * (
            lastEventTick - getCurrTick() );
        listSET_LIST_ITEM_VALUE( &( ( pxCurrentTCB )->xGenericListItem ), tmp);

      /*add task to the other ready list (sort order):*/
      vListInsert( &pxReadyTaskListLLREF2, &( pxTCB->xGenericListItem ) );

   }

   /*invert the pointers to the two ready lists:*/
   pxTemp = pxReadyTaskListLLREF;
   pxReadyTaskListLLREF = pxReadyTaskListLLREF2;
   pxReadyTaskListLLREF = pxTemp;

}
```

and here is $funcEventHandle()$ function implementation:

```
/*
 * Called at every nextEventTick tick
 *
void funcEventHandle()
{
    /*update the running task local remaining execution time:*/
    int tmp = ( pxCurrentTCB )->xGenericListItem - ( lastEventTick - getCurrTick()
        );
    listSET_LIST_ITEM_VALUE( &( ( pxCurrentTCB )->xGenericListItem ), tmp);

    /*sort Ready list bu POP and PUSH the running task: */
    uxListRemove( &( pxCurrentTCB->xGenericListItem );
    prvAddTaskToReadyList( pxCurrentTCB );

```

```
    /*update last event time:*/
    lastEventTick = nextEventTime;

    /*calc next event B time:*/
    TCB_t pxTCB = ( TCB_t * ) listGET_OWNER_OF_HEAD_ENTRY( xReadyTasksListLLREF );
    TickType_t u = (TickType_t )listGET_LIST_ITEM_VALUE( &( pxTCB->xGenericListItem
        ) );
    nextEventTick = xTLPlaneStart + ( u * (xTLPlaneEnd-xTLPlaneStart) );

    /*force context switch:*/
    portYELD_WITH_CONTEXT();
}
```

$xTaskIncrementTick()$ function is modified as well, in order to execute $funcEventHandle()$ and $functNewTLPlane()$ functions at hte right time, at each $xTLPlaneStart$ and $nextEventTick$ tick respectively:

```
BaseType_t xTaskIncrementTick( void )
{
...
    if( currentTick == xTLPlaneStart )
    {
        functNewTLPlane();
    }
...
    if( currentTick == xTLPlaneStart )
    {
        funcEventHandle();
    }
...
}
```

4.3.1 IDLE task management

The IDLE task management is also modified. As we saw for the EDF algorithm, $vTaskStartScheduler()$ function initializes the IDLE task. The IDLE task management is foundamental, since FreeRTOS requires one task in running state at each rime, and IDLE task should run only when no other tasks are in Ready List. In the LLREF scheduler, this IDLE behaviour can be performed by a task that occupy the last position in the Ready List every time. This behaviour can be implemented as a task which local remain execution time is always zero: $l_{IDLE} = \mu_{IDLE} * \Delta$ - that is, $\mu_{IDLE} = c/p = 0$; so, if we set IDLE task period to zero, IDLE task will occupy the last position of Ready List and will be scheduled only if no other tasks are ready.

When a B event occurs, the running task finishes its local remain execution time goes to zero. Then the $funcEventHandle()$ function is called and the running task local remain execution time is updated to zero, and the task is putted at the bottom of the Ready List, behind the IDLE task. If no other tasks are in the Ready List, then IDLE task will be executed till the end of the current T-L plane. $vTaskStartScheduler()$ is modified as follow:

```
        /*E.C. : */
        #if (configUSE_LLREF_SCHEDULER == 1)
      {
          tickType initIDLEPeriod = 0;
          tickType initIDLECapacity = 1;
         xReturn = xTaskLLREFCreate( prvIdleTask, "IDLE", tskIDLE_STACK_SIZE, ( void
             * ) NULL, ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ), NULL,
             initIDLEPeriod, initIDLECapacity );
      }
      #else
         /* Create the idle task without storing its handle. */
         xReturn = xTaskCreate( prvIdleTask, "IDLE", tskIDLE_STACK_SIZE, ( void * )
             NULL, ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ), NULL );
//    #endif
```

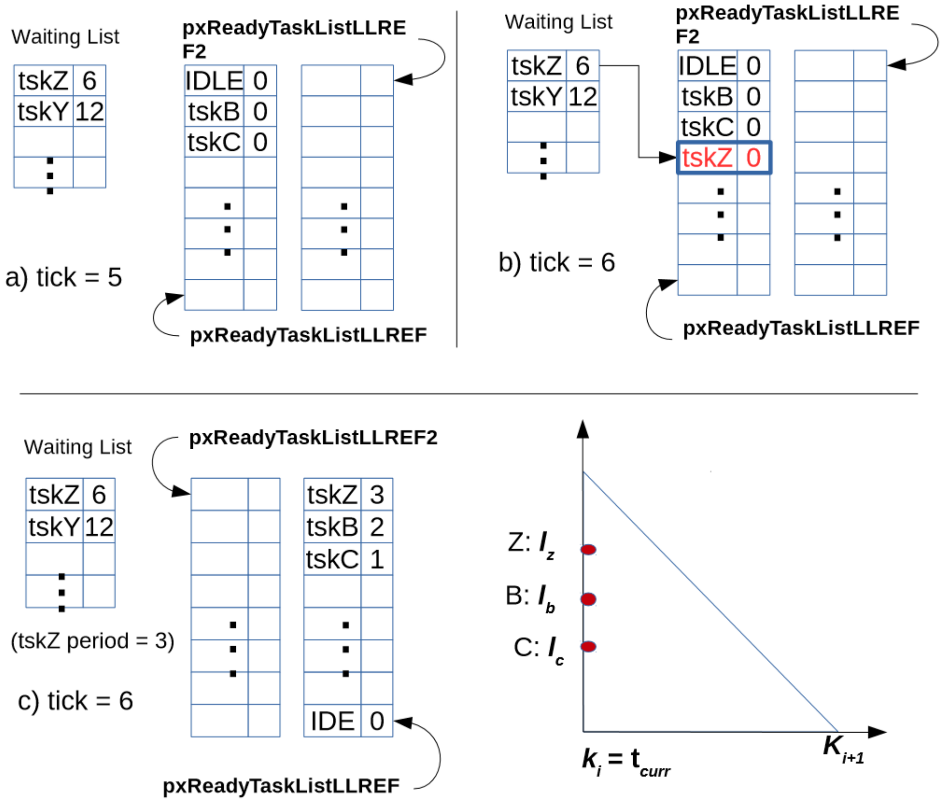

Figure 4.8: Ready List management during a B event

4.3.2 Scheduling example

4.7

Two scheduling example are illustrated: the first shows what happens to the Ready list when a new T-L plane starts and $funcNewTLPlane()$ function is called; the second example shows the Ready List behaviour after the execution of $funcEventHandle()$, following a B event execution.

The first example is shown in Figure 4.8. When tick count is 5, the Ready List situation is illustrated in Figure 4.8-a: the current T-L plane will finish at tick=6 when tskZ will awaken, and IDLE task is running since all tasks have already finished their local remaining execution time. In Figure 4.8-b the tick interrupt occurred and the ISR $vPortYeldFromTick()$ is called:
-running task context is saved ($portSave\_CONTEXT()$);
-$xTAskIncrementTick()$ function is called, so tick is incremented to 6, and tskZ is removed from the Waiting List, its *genericListIteam* is set to zero, and is added to the list pointed by

$*pxReadyTaskListLLREF$ (tskZ is added at the last position of the list since all the other ready tasks have their local remaining execution time equal to zero);

-since $xTLPlaneStart = 6$, $funcNewTLPlane()$ is called: local remaining execution time is initialized for all the task in Ready List, then one by one are removed from the Ready List and added to the List pointed by $*pxReadyTaskListLLREF2$, where now are sorted; then $*pxReadyTaskListLLREF2$ and $*pxReadyTaskListLLREF$ are switched;

-tskZ is now on the top of the Ready List and is pointed by $*pxCurrentTCB$: $portRESTORE\_CONTEXT()$ function will restore tskZ context, and will be executed.

The second example is shown in Figure 4.9. In a) is shown the Ready List at the beginning of the current T-L plane: tskB is running, and $nextEventTick = 8$. In b) tick=8, and $funcEventHandle()$ function is called:

-tskB local execution remain time is updated to zero, then the task is removed and putted at the last position of the Ready List, behind the IDLE task. Now tskC is the task with the higher local remain execution time, and is pointed by $*pxCurrentTCB$; then a context switch is forced and $portRESTORE\_CONTEXT()$ function restores tskZ context, which will be executed.

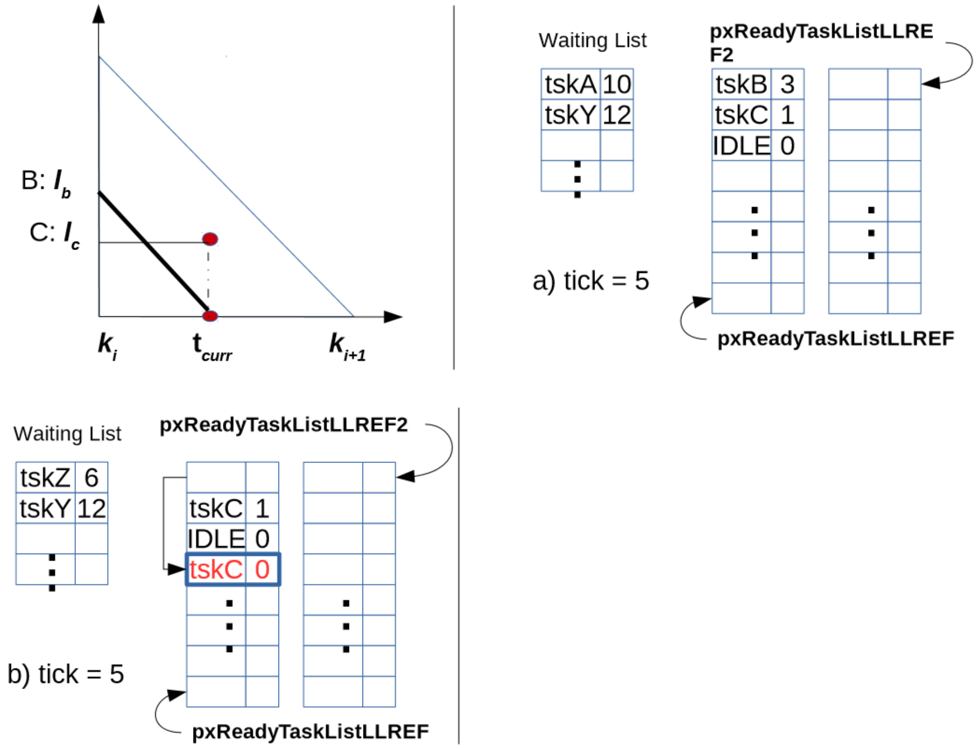

Figure 4.9: Ready List during a T-L plane initialization

4.4 Tests and Results

To test the correctness of the implemented LLREF scheduler, we execute two tasks whose LLREF scheduling sequence is known, and match the run-time scheduling sequence with the expected one. To monitor the run-time scheduling sequence we used the same trace macros unctions introducted for the EDF scheduler. The two tasks chosen are: A(p=5, c=2), B(P=8, c=2), as the test example

First, some changes to the demo application are needed:

-since B scheduling events can occur not only in integer number of tick time, we have to choose carefully the period and capacity parameters of the tasks: the two tasks chosen are: A(p=50, c=20), B(P=80, c=20); - the new *xTaskLLREFCreate*() function is called to initialize the tasks;

```
//-----------------------------------------------
// Global Variables
//-----------------------------------------------
```

```c

#define CAPACITY 20   //cpu time in tick
#define A_PERIOD 50   //task A period
#define B_PERIOD 80   //task B period

//-----------------------------------------------
// Start point
//---------------------------------------------------

int main(void)
{

  SystemInit();

  xTaskLLREFCreate( TSK_A, ( const char * ) "A",
                     configMINIMAL_STACK_SIZE, NULL,
                     1, NULL, A_PERIOD, CAPACITY );
  xTaskLLREFCreate( TSK_B, ( const char * ) "B",
                     configMINIMAL_STACK_SIZE, NULL,
                     1, NULL, B_PERIOD, CAPACITY );

  // FreeRTOS Scheduler starten
  vTaskStartScheduler();

  // wird nie erreicht!!
  while(1)
  {

  }
}
```

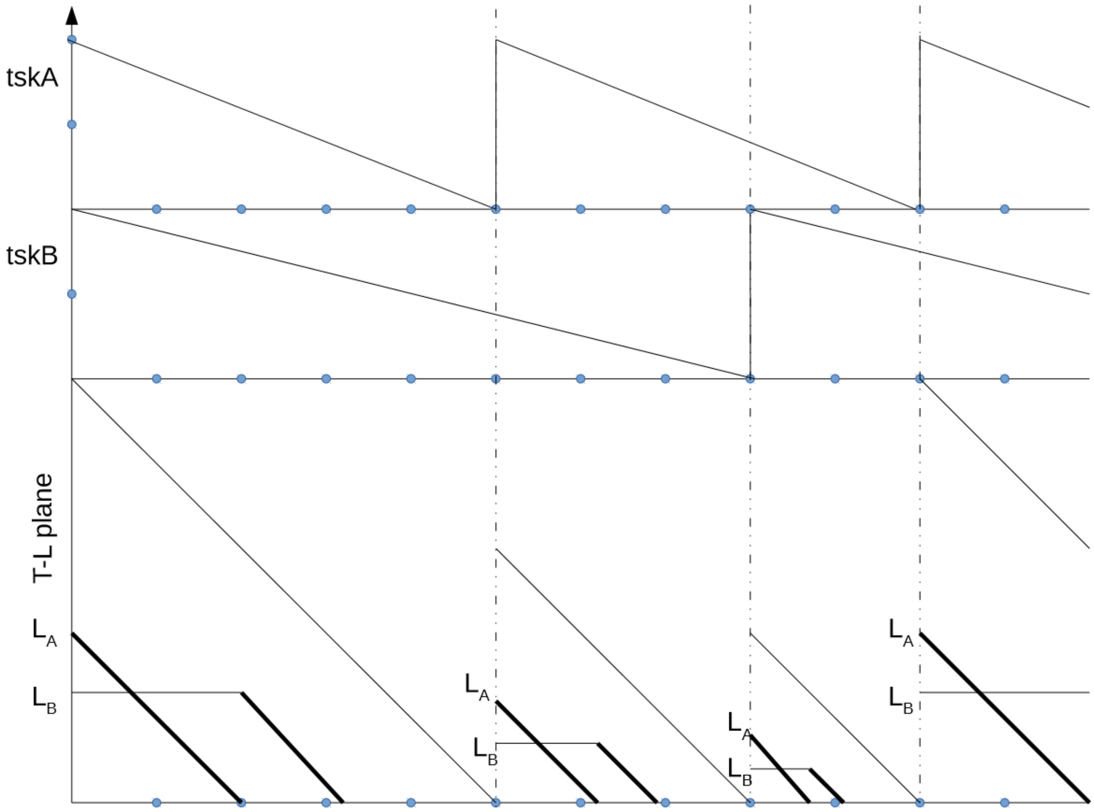

Figure 4.10: T.L plane construction for task A (p=5, c=2) and task B (p=8, c=2); For T-L plane 1: $L_A = 2$, $L_B = 1.2$; For T-L plane 2: $L_A = 1.2$, $L_B = 0.7$; For T-L plane 3: $L_A = 0.8$, $L_B = 0.5$; For T-L plane 4: $L_A = 2$, $L_B = 1.2$;

Then, a correct LLREF scheduling sequence is calculated. Figure 4.10 shows how T-L planes are obtained, then Figure 4.11 shows the final scheduling sequence from the T-L plane sequence.

Figure 4.12 shows the printed console output for tick=1 to 400 (the greatest common period): the obtained schedule sequence is correct, matching it with the scheduling sequence previously calculated. [11]

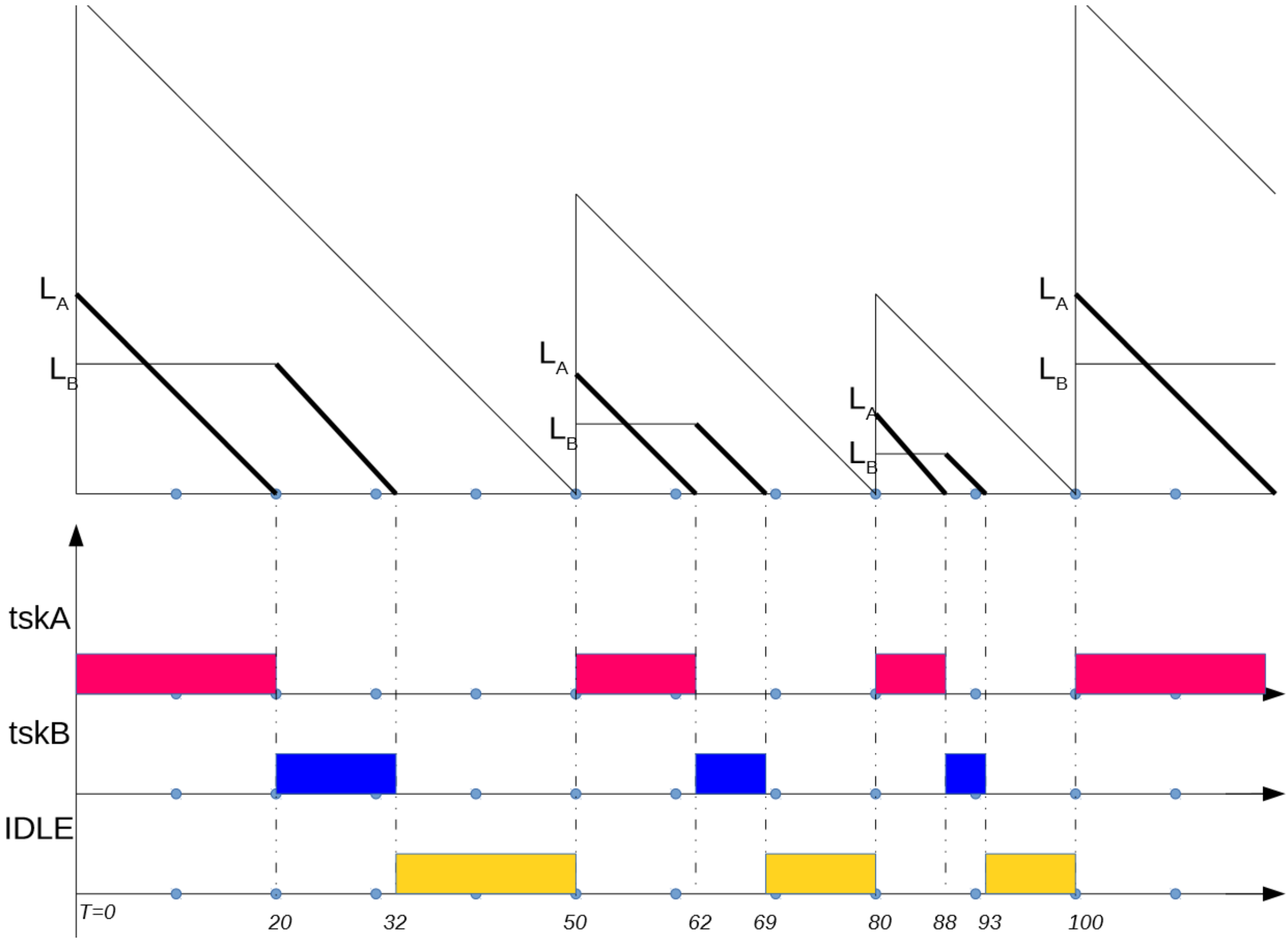

Figure 4.11: LLREF scheduling for tskA and tskB: t=0 to t=120

```
////////////////////////////
TICK : 1
.
.
.
TICK : 20
Task Delay: A, Task Out: A
Task IN: B
TICK : 21
.
.
.
TICK : 32
Task Out: B
Task IN: IDLE
TICK : 33
.
.
.
TICK : 50
Task Out: IDLE
Task IN: A
TICK : 51
.
.
.
TICK : 62
Task Out: A
Task IN: B
TICK : 63
.
.
.
TICK : 69
Task Delay: B, Task Out: B
Task IN: IDLE
TICK : 70
.
.
.
TICK : 80
Task Out: IDLE
Task IN: A
TICK : 81
.
.
.
TICK : 88
Task Delay: A, Task Out: A
Task IN: B
TICK : 89
.
.
.
TICK : 93
Task Delay: B, Task Out: B
Task IN: IDLE
TICK : 94
.
.
.
TICK : 100
Task Out: IDLE
Task IN: A
TICK : 101
.
.
.
TICK : 120
Task Delay: A, Task Out: A

Task Delay: A, Task Out: A
Task IN: B
TICK : 121
.
.
.
TICK : 132
Task Out: B
Task IN: IDLE
TICK : 133
.
.
TICK : 150
Task Out: IDLE
Task IN: A
TICK : 151
TICK : 152
TICK : 153
TICK : 154
Task Out: A
Task IN: B
TICK : 155
TICK : 156
Task Delay: B, Task Out: B
Task IN: IDLE
TICK : 157
TICK : 158
TICK : 159
TICK : 160
Task Out: IDLE
Task IN: A
TICK : 161
.
.
.
TICK : 176
Task Delay: A, Task Out: A
Task IN: B
TICK : 177
TICK : 178
TICK : 179
TICK : 180
Task Out: B
Task IN: IDLE
TICK : 181
.
.
.
TICK : 200
Task Out: IDLE
Task IN: A
TICK : 201
.
.
.
TICK : 216
Task Out: A
Task IN: B
TICK : 217
.
.
.
TICK : 226
Task Delay: B, Task Out: B
Task IN: IDLE
TICK : 227
.
.
.

TICK : 240
Task Out: IDLE
Task IN: A
TICK : 241
TICK : 242
TICK : 243
TICK : 244
Task Delay: A, Task Out: A
Task IN: B
TICK : 245
TICK : 246
Task Delay: B, Task Out: B
Task IN: IDLE
TICK : 247
TICK : 248
TICK : 249
TICK : 250
Task Out: IDLE
Task IN: A
TICK : 251
.
.
.
TICK : 270
Task Delay: A, Task Out: A
Task IN: B
TICK : 271
.
.
.
TICK : 282
Task Out: B
Task IN: IDLE
TICK : 283
.
.
.
TICK : 339
Task Out: B
Task IN: IDLE
TICK : 340
.
.
.
TICK : 350
Task Out: IDLE
Task IN: A
TICK : 351
.
.
.
TICK : 370
Task Delay: A, Task Out: A
Task IN: B
TICK : 371
.
.
.
TICK : 382
Task Delay: B, Task Out: B
Task IN: IDLE
TICK : 383
.
.
.
TICK : 400

.
.
TICK : 300
Task Out: IDLE
Task IN: A
TICK : 301
.
.
.
TICK : 308
Task Out: A
Task IN: B
TICK : 309
.
.
.
TICK : 313
Task Delay: B, Task Out: B
Task IN: IDLE
TICK : 314
.
.
.
TICK : 320
Task Out: IDLE
Task IN: A
TICK : 321
.
.
.
TICK : 332
Task Delay: A, Task Out: A
Task IN: B
TICK : 333
.
```

Figure 4.12: LLREF scheduling for tskA and tskB: Log output

Chapter 5

Conclusion

The main quality of FreeRTOS fixed-priority scheduler is its simplicity. It guarantees a very low overhead and easy system analysis. This work presented two alternative schedulers, that implement dynamic priority scheduler algorithms. EDF scheduler implementation requires an overhead comparable to the original scheduler. Tests shows how the implemented algorithm performs correctly the expected task sequence.
LLREF scheduler implementation requires more complexity: capacity estimation of each task in the system is required along with the task period. The T-L plane management also contributes to increase the scheduler overhead. The test phase validates the schedule correctness.

The two proposed solutions works well according with the given specification. It must be clear, however, that the presented algorithms are intended for academic use only, since a sufficient high level of reliability for commercial use can not be guarantied at this phase of development. For instance, schedulers correctness is tested for low system tick count only, since tick variable overflow is not managed. Future implementations of these algorithms should work on this aspect. Another important aspect to implement in future works could be the sporadic tasks support, since in the proposed algorithms only periodic tasks were considered.

Bibliography

[1] L. B. Das, *Embedded Systems: An Integrated Approach.* 2013.

[2] A. B. Tucker, *Computer Science Handbook, Second Edition.* 2004.

[3] C. L. Liu, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, 1973.

[4] J. Cho, Ravindran, "An optimal real-time scheduling algorithm for multiprocessorsg," *IEEE Computer Society*, 2007.

[5] F. ltd., "Freertos official website." http://www.freertos.org/RTOS.html, Feb. 2016.

[6] D. Knuth, "Knuth: Computers and typesetting."

[7] B. University, "Cheddar tool." http://beru.univ-brest.fr/ singhoff/cheddar/, Feb. 2016.

[8] A. Documentation, "What is semihosting?." http://infocenter.arm.com/help/index.jsp/, Feb. 2016.

[9] P. blog, "Stm32f407 tutorial with coocox." http://patrickleyman.be/blog/stm32f407-tutorial-with-coocox/, Feb. 2016.

[10] C. Mattihalli, "Designing and implementing of earliest deadline first scheduling algorithm on standard linux," *IEEE/ACM International Conference*, 2010.

[11] M. C. Marko Bertogna and G. Lipari, "Improved schedulability analysis of edf on multi-processor platforms," *17th Euromicro Conference on Real-Time Systems*, 2005.