REPORT

Building EDF Scheduler based on FreeRTOS Kernel.

BY: MOHAMMED KHALID

Entrance

FreeRTOS is one of the real time operating systems that uses the fixed priority schedulers to handle tasks executions within a computer machine, that is, every task in the system is pre-assigned a priority value while the system is being designed to represent its weightiness among others tasks.

In the fixed priority topology, it’s up to designer to consider every task’s deadline, and to choose carefully the priorities in order for the system to be schedulable and all tasks be able to catch their deadlines.

Although this might appear to put extra load on a designer’s shoulder, This topology is doing well in many cases where task’s deadline is not the most critical part of the system. In others systems, like safety-critical ones, tasks can be event triggered, and catching their deadline is the most important criteria to achieve. That's why the fixed priority scheduler has some limitations at those kind of systems.

Although, FreeRTOS is offering nice APIs allows a programmer from managing the priorities at runtime and enabling a system to act dynamically, still there is too much headache to relate a task deadline with a dynamic priority.

That's why other typologies in scheduling like EDF, or Earliest Deadline First, is exist. EDF scheduler automatically relates deadlines and prioritization, where at any point of time the task being executed has nearest deadline between all other tasks, and no other task can preempt it unless it has has earliest deadline than it.

Here we present an algorithm to built such scheduler based on the FreeRTOS Kernel implementation and make it compatible with the original fixed priority scheduler to make it easy for switching between them.

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 :-**

1. The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests.

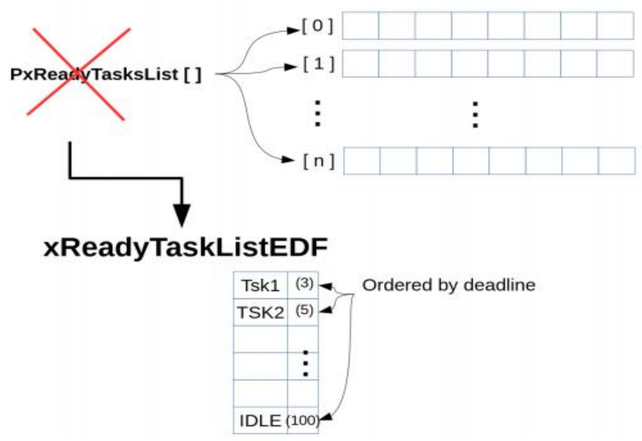
2. Deadlines consist of run-ability constraints only, i.e. each task must be completed before the next requests for it occurs.

3. 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.

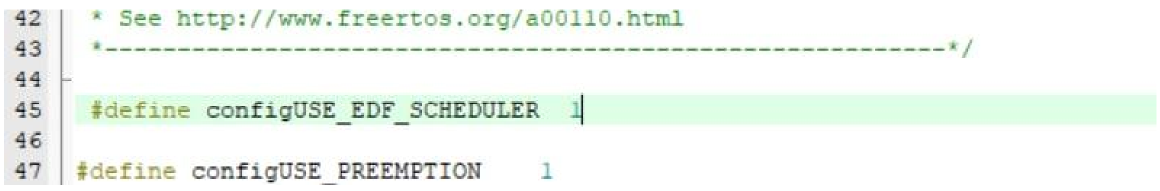
4. Run-time for each task is constant for that task and does not vary with time. Run-time refers to the time which is taken by a processor to execute the task without interruption.

Implementation in FreeRTOS Kernel

As said 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 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 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.

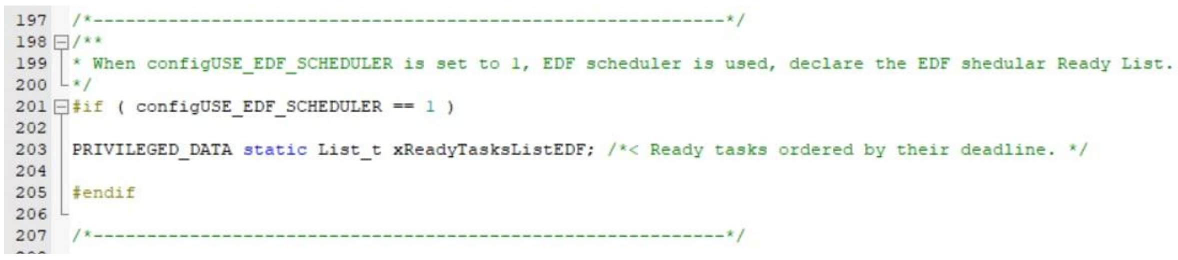


All changes that will be illustrated refer to tasks.c file, since scheduler structures and methods are contained there. According with the FreeRTOS style guideline, a configuration variable ***configUSE_EDF_SCHEDULER***, is added to the ***FreeRTOSConfig.h*** *file*. When ***configUSE_EDF_SCHEDULER***,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



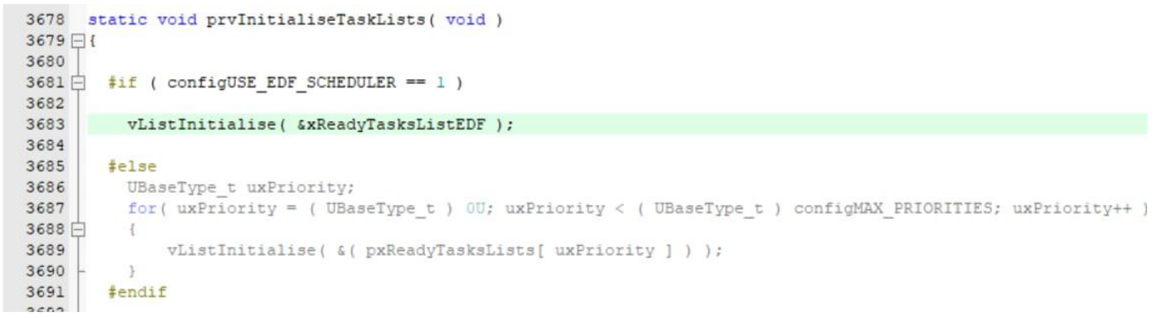
```
 * See http://www.freertos.org/a00110.html
 *----------------------------------------------------------*/

 #define configUSE_EDF_SCHEDULER  1

#define configUSE_PREEMPTION     1
```

Then, the new Ready List is declared: ***xReadyTasksListEDF*** is a simple list structure.



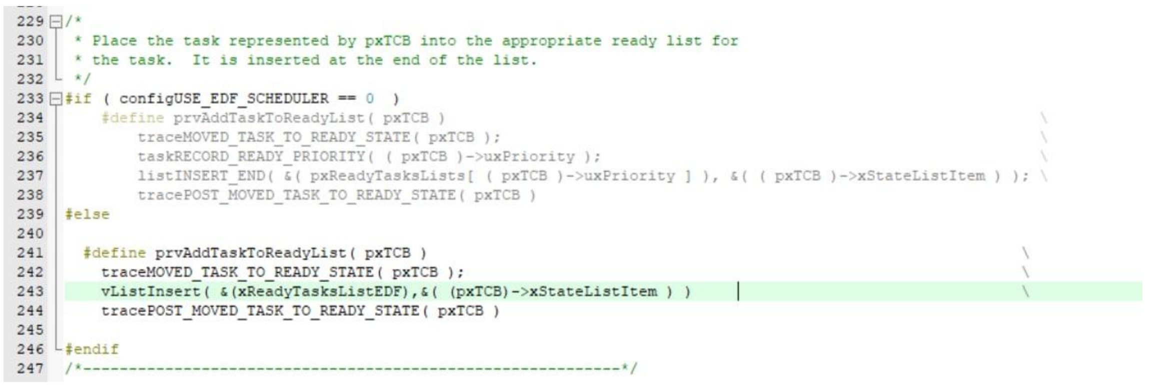
```
/*-----------------------------------------------------------*/
/**
* When configUSE_EDF_SCHEDULER is set to 1, EDF scheduler is used, declare the EDF shedular Ready List.
*/
#if ( configUSE_EDF_SCHEDULER == 1 )

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 )
{

  #if ( configUSE_EDF_SCHEDULER == 1 )

    vListInitialise( &xReadyTasksListEDF );

  #else
    UBaseType_t uxPriority;
    for( uxPriority = ( UBaseType_t ) 0U; uxPriority < ( UBaseType_t ) configMAX_PRIORITIES; uxPriority++ )
    {
        vListInitialise( &( pxReadyTasksLists[ uxPriority ] ) );
    }
  #endif
```

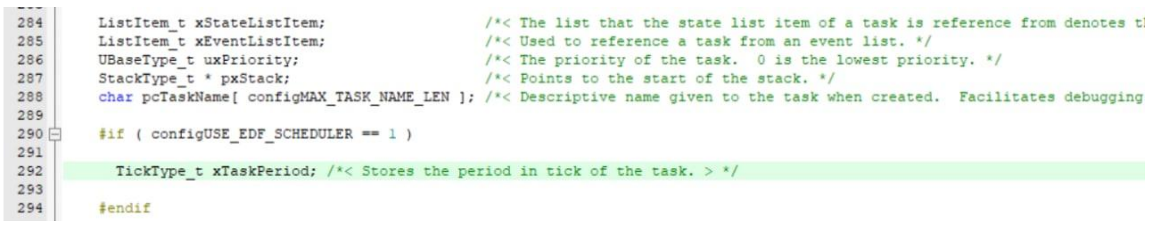
prvAddTaskToReadyList() method that adds a task to the Ready List is then modified as follows:



```
/*
 * 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  )
    #define prvAddTaskToReadyList( pxTCB )                                                    \
        traceMOVED_TASK_TO_READY_STATE( pxTCB );                                              \
        taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority );                                   \
        listINSERT_END( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( ( pxTCB )->xStateListItem ) ); \
        tracePOST_MOVED_TASK_TO_READY_STATE( pxTCB )
#else

  #define prvAddTaskToReadyList( pxTCB )                                                     \
    traceMOVED_TASK_TO_READY_STATE( pxTCB );                                                 \
    vListInsert( &(xReadyTasksListEDF),&( (pxTCB)->xStateListItem ) )                        \
    tracePOST_MOVED_TASK_TO_READY_STATE( pxTCB )

#endif
/*-----------------------------------------------------------*/
```

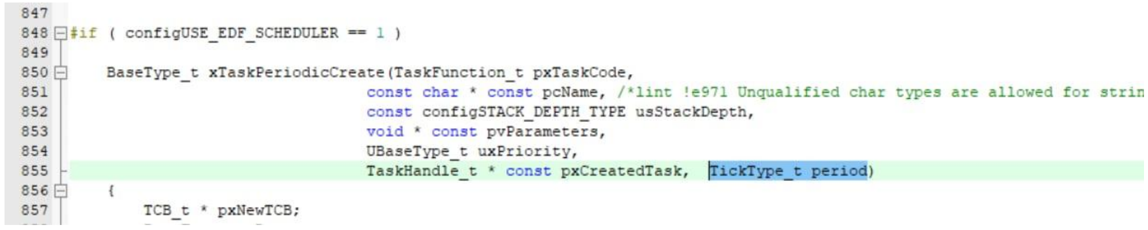
vListInsert() method is called to insert in ***xReadyTasksListEDF*** the task **TCB** pointer. The item will be inserted into the list in a position determined by its item value ***xGenericListItem*** (descending item value order). So it is assumed that ***xStateListItem*** contains the next task deadline.

The second change introduced refers to the task structure. 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**):



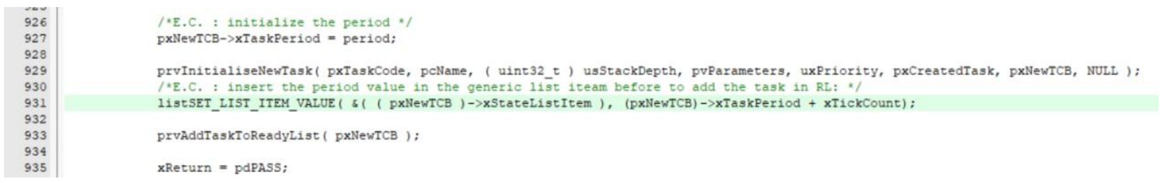
```
    ListItem_t xStateListItem;                  /*< The list that the state list item of a task is reference from denotes t
    ListItem_t xEventListItem;                  /*< Used to reference a task from an event list. */
    UBaseType_t uxPriority;                     /*< The priority of the task.  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

    #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 ***xTaskCreate()***, that receives the task period as additional input parameter and set the ***xTaskPeriod*** variable in the task **TCB** structure.



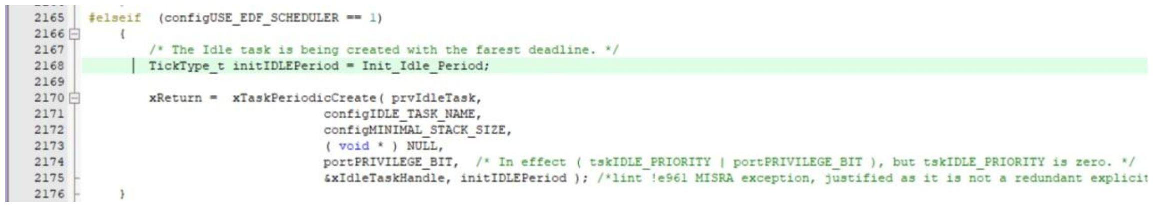
```

#if ( configUSE_EDF_SCHEDULER == 1 )

    BaseType_t xTaskPeriodicCreate(TaskFunction_t pxTaskCode,
                                   const char * const pcName, /*lint !e971 Unqualified char types are allowed for strin
                                   const configSTACK_DEPTH_TYPE usStackDepth,
                                   void * const pvParameters,
                                   UBaseType_t uxPriority,
                                   TaskHandle_t * const pxCreatedTask,  TickType_t period)
    {
        TCB_t * pxNewTCB;
```

Before adding the new task to the Ready List by calling ***prvAddTaskToReadyList()***, the task's ***xStateListItem*** is initialized to the value of the next task deadline.



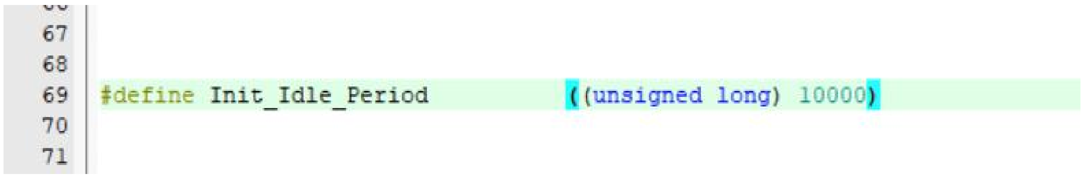
```
        /*E.C. : initialize the period */
        pxNewTCB->xTaskPeriod = period;

        prvInitialiseNewTask( pxTaskCode, pcName, ( uint32_t ) usStackDepth, pvParameters, uxPriority, pxCreatedTask, pxNewTCB, NULL );
        /*E.C. : insert the period value in the generic list iteam before to add the task in RL: */
        listSET_LIST_ITEM_VALUE( &( ( pxNewTCB )->xStateListItem ), (pxNewTCB)->xTaskPeriod + xTickCount);

        prvAddTaskToReadyList( pxNewTCB );

        xReturn = pdPASS;
```

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:



```
#elseif  (configUSE_EDF_SCHEDULER == 1)
    {
        /* The Idle task is being created with the farest deadline. */
      | TickType_t initIDLEPeriod = Init_Idle_Period;

        xReturn =  xTaskPeriodicCreate( prvIdleTask,
                                configIDLE_TASK_NAME,
                                configMINIMAL_STACK_SIZE,
                                ( void * ) NULL,
                                portPRIVILEGE_BIT,  /* In effect ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ), but tskIDLE_PRIORITY is zero. */
                                &xIdleTaskHandle, initIDLEPeriod ); /*lint !e961 MISRA exception, justified as it is not a redundant explicit
    }
```

Where **Init_Idle_Period** is a user defined macro ..

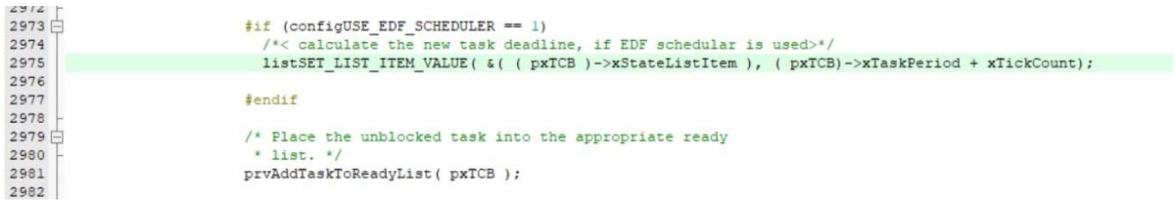


```


#define Init_Idle_Period          ((unsigned long) 10000)


```

The IDLE task is initialized with a period of *initIDLEPeriod* = 10000. We assume that no task can have a period greater than initIDLEP eriod: in this way, when the IDLE task is

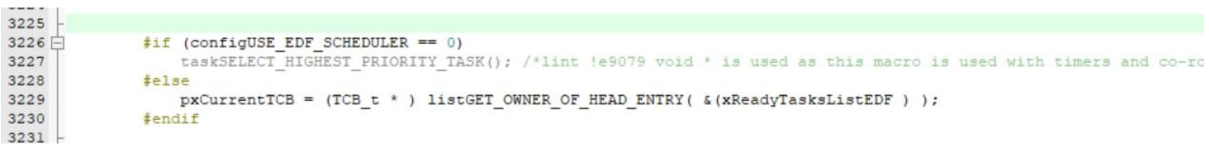
added to the Ready List, it will be at the last position of the list, since its deadline will be greater than any other task ($TASK_{deadline}$ = $Tick_{cur}$ + $TASK_{period}$) with $Tick_{cur}$ = 0 and $IDLE_{period}$ = *initIDLEPeriod* greater than any other task period.

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.



```
                #if (configUSE_EDF_SCHEDULER == 1)
                  /*< calculate the new task deadline, if EDF schedular is used>*/
                  listSET_LIST_ITEM_VALUE( &( ( pxTCB )->xStateListItem ), ( pxTCB)->xTaskPeriod + xTickCount);

                #endif

                /* Place the unblocked task into the appropriate ready
                 * list. */
                prvAddTaskToReadyList( pxTCB );
```

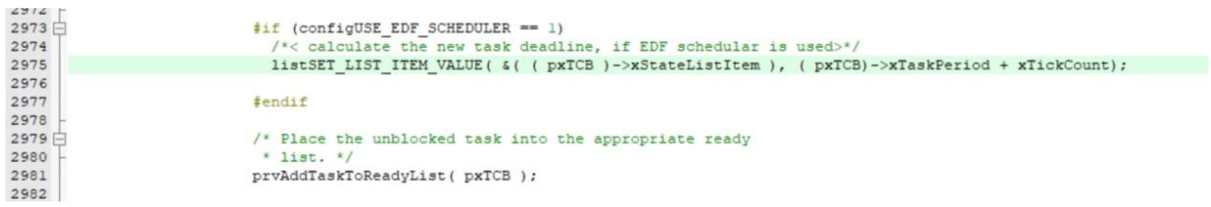
Another change needed involves the switch context mechanism. Every time the running task is suspended, or a suspended task with an higher priority than the running task awakes, a switch context occurs. ***vTaskSwitchContext()*** method is in charge to update the ****pxCurrentTCB*** pointer to the new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



```

        #if (configUSE_EDF_SCHEDULER == 0)
            taskSELECT_HIGHEST_PRIORITY_TASK(); /*lint !e9079 void * is used as this macro is used with timers and co-ro
        #else
            pxCurrentTCB = (TCB_t * ) listGET_OWNER_OF_HEAD_ENTRY( &(xReadyTasksListEDF ) );
        #endif
```

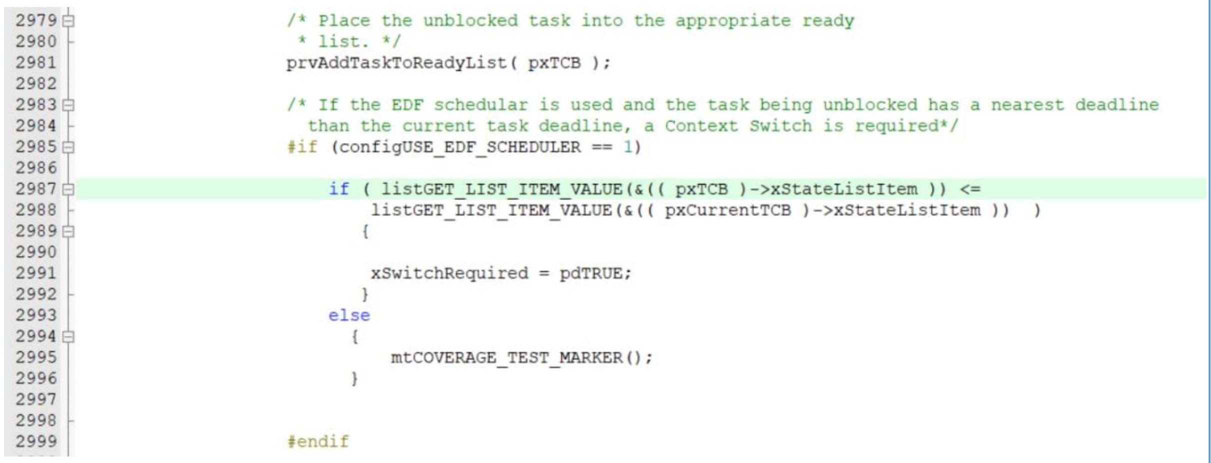
taskSELECT_HIGHEST_PRIORITY_TASK() method is replaced in order to assign to ***pxCurrentTCB*** the task at the first place of the new Ready List.

Last change required to get all the pieces in the new EDF scheduler to work is to update tasks deadlines stored in the ***xStateListItem*** every tick, but actually we Just need to update deadline for those tasks whose deadlines outdated, that is The tasks has recently unblocked from the waiting list, again the new deadline will be $\text{TASK}_{\text{deadline}} = \text{Tick}_{\text{cur}} + \text{TASK}_{\text{period.}}$



```
                #if (configUSE_EDF_SCHEDULER == 1)
                  /*< calculate the new task deadline, if EDF schedular is used>*/
                  listSET_LIST_ITEM_VALUE( &( ( pxTCB )->xStateListItem ), ( pxTCB)->xTaskPeriod + xTickCount);

                #endif

                /* Place the unblocked task into the appropriate ready
                 * list. */
                prvAddTaskToReadyList( pxTCB );
```

Also we have determine if a context switching is required after this tick or not.
If a released task has earliest deadline than the one that being executed then we need a ***Context Switching,*** then we compare both of them in order make the decision.



```
                    /* Place the unblocked task into the appropriate ready
                     * list. */
                    prvAddTaskToReadyList( pxTCB );

                    /* If the EDF schedular is used and the task being unblocked has a nearest deadline
                      than the current task deadline, a Context Switch is required*/
                    #if (configUSE_EDF_SCHEDULER == 1)

                        if ( listGET_LIST_ITEM_VALUE(&(( pxTCB )->xStateListItem )) <=
                            listGET_LIST_ITEM_VALUE(&(( pxCurrentTCB )->xStateListItem ))  )
                           {

                            xSwitchRequired = pdTRUE;
                           }
                        else
                          {
                              mtCOVERAGE_TEST_MARKER();
                          }


                    #endif
```

Testing and Analysis

To test and verify our implementation of the EDF scheduler we aim to create 6 tasks with different deadlines and different execution time.

The six tasks are as follows :-

Task 1: "Button_1_Monitor" , {Periodicity: 50, Deadline: 50}
This task will monitor rising and falling edge on button 1 and send this event to the consumer task.

Task 2: ""Button_2_Monitor"", {Periodicity: 50, Deadline: 50}
This task will monitor rising and falling edge on button 2 and send this event to the consumer task.

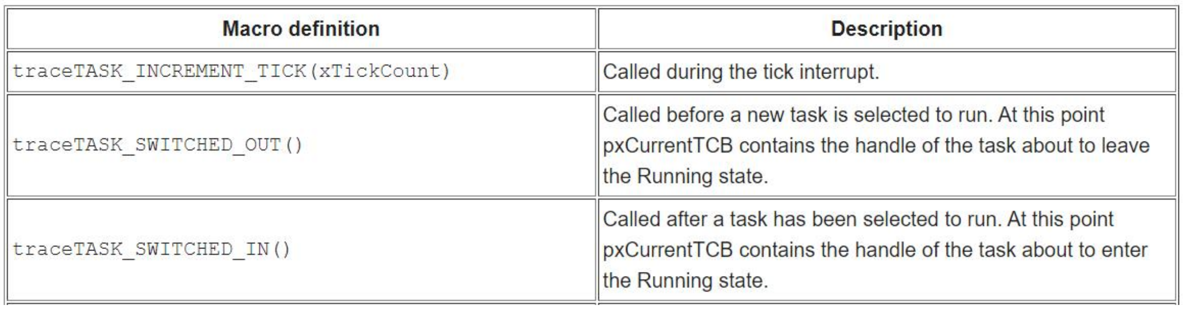
Task 3: ""Periodic_Transmitter"", {Periodicity: 100, Deadline: 100}
This task will send periodic string every 100ms to the consumer task.

Task 4: ""Uart_Receiver"", {Periodicity: 20, Deadline: 20}
This is the consumer task which will write on UART any received string from other tasks.

Task 5: "Load_1_Simulation", {Periodicity: 10, Deadline: 10}, with execution time = 5ms.

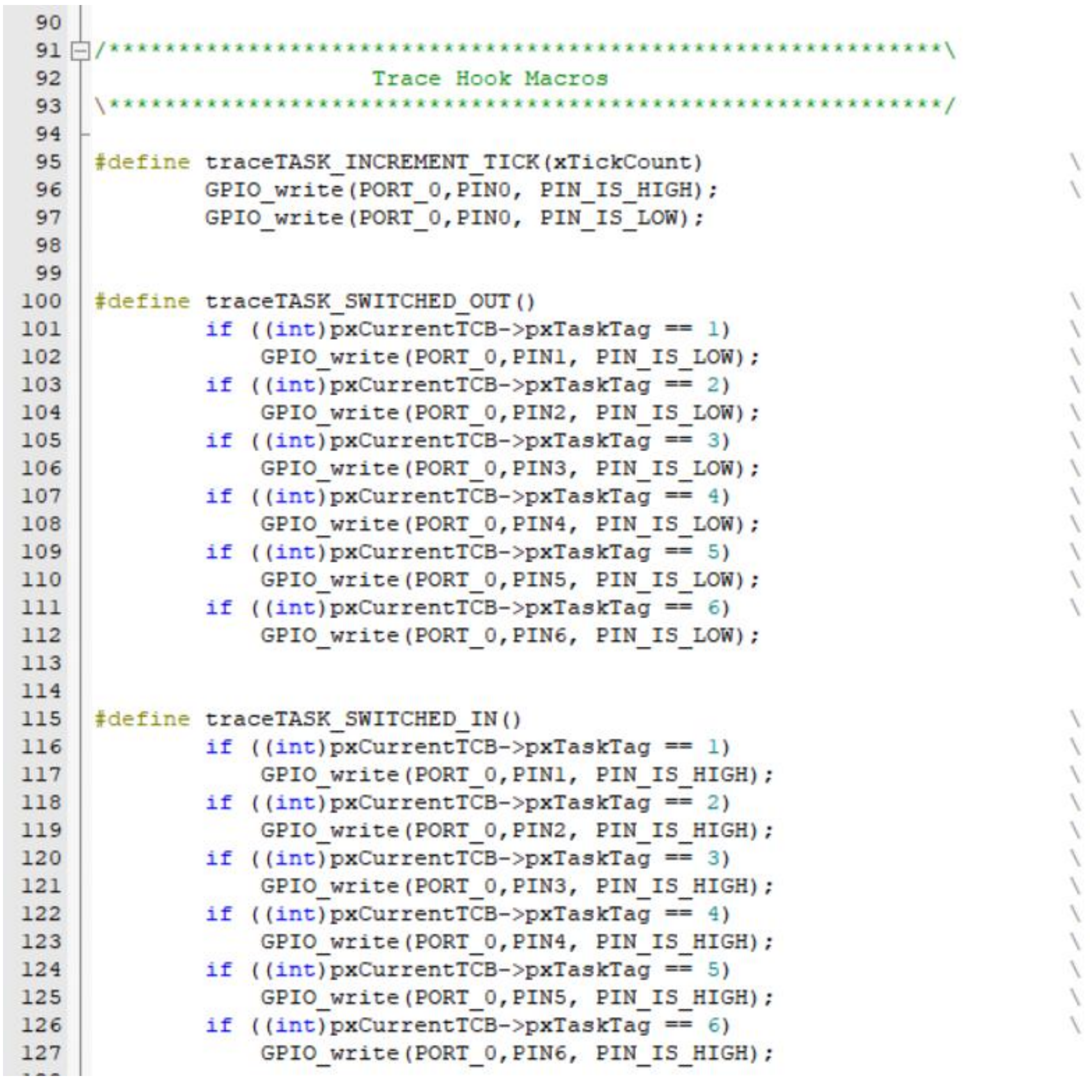
Task 6: "Load_2_Simulation", {Periodicity: 100, Deadline: 100}, with execution time = 12ms.

And to trace the tasks execution and scheduling at runtime we use the Trace Hooks macros offered by FreeRTOS.



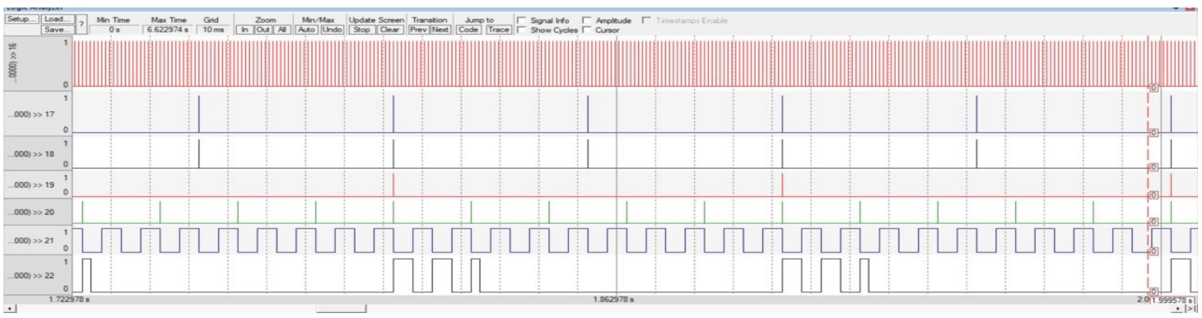
Macro definition	Description
`traceTASK_INCREMENT_TICK(xTickCount)`	Called during the tick interrupt.
`traceTASK_SWITCHED_OUT()`	Called before a new task is selected to run. At this point pxCurrentTCB contains the handle of the task about to leave the Running state.
`traceTASK_SWITCHED_IN()`	Called after a task has been selected to run. At this point pxCurrentTCB contains the handle of the task about to enter the Running state.

Using the GPIOs pin to generate signals in tasks going it and out from execution we can analyse our system at runtime..



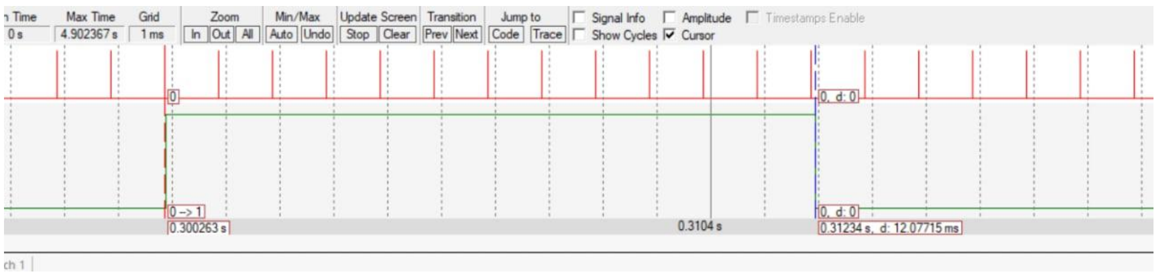
```c

/****************************************************************\
                    Trace Hook Macros
\****************************************************************/

#define traceTASK_INCREMENT_TICK(xTickCount)                          \
        GPIO_write(PORT_0,PIN0, PIN_IS_HIGH);                         \
        GPIO_write(PORT_0,PIN0, PIN_IS_LOW);


#define traceTASK_SWITCHED_OUT()                                      \
        if ((int)pxCurrentTCB->pxTaskTag == 1)                        \
            GPIO_write(PORT_0,PIN1, PIN_IS_LOW);                      \
        if ((int)pxCurrentTCB->pxTaskTag == 2)                        \
            GPIO_write(PORT_0,PIN2, PIN_IS_LOW);                      \
        if ((int)pxCurrentTCB->pxTaskTag == 3)                        \
            GPIO_write(PORT_0,PIN3, PIN_IS_LOW);                      \
        if ((int)pxCurrentTCB->pxTaskTag == 4)                        \
            GPIO_write(PORT_0,PIN4, PIN_IS_LOW);                      \
        if ((int)pxCurrentTCB->pxTaskTag == 5)                        \
            GPIO_write(PORT_0,PIN5, PIN_IS_LOW);                      \
        if ((int)pxCurrentTCB->pxTaskTag == 6)                        \
            GPIO_write(PORT_0,PIN6, PIN_IS_LOW);


#define traceTASK_SWITCHED_IN()                                       \
        if ((int)pxCurrentTCB->pxTaskTag == 1)                        \
            GPIO_write(PORT_0,PIN1, PIN_IS_HIGH);                     \
        if ((int)pxCurrentTCB->pxTaskTag == 2)                        \
            GPIO_write(PORT_0,PIN2, PIN_IS_HIGH);                     \
        if ((int)pxCurrentTCB->pxTaskTag == 3)                        \
            GPIO_write(PORT_0,PIN3, PIN_IS_HIGH);                     \
        if ((int)pxCurrentTCB->pxTaskTag == 4)                        \
            GPIO_write(PORT_0,PIN4, PIN_IS_HIGH);                     \
        if ((int)pxCurrentTCB->pxTaskTag == 5)                        \
            GPIO_write(PORT_0,PIN5, PIN_IS_HIGH);                     \
        if ((int)pxCurrentTCB->pxTaskTag == 6)                        \
            GPIO_write(PORT_0,PIN6, PIN_IS_HIGH);
```

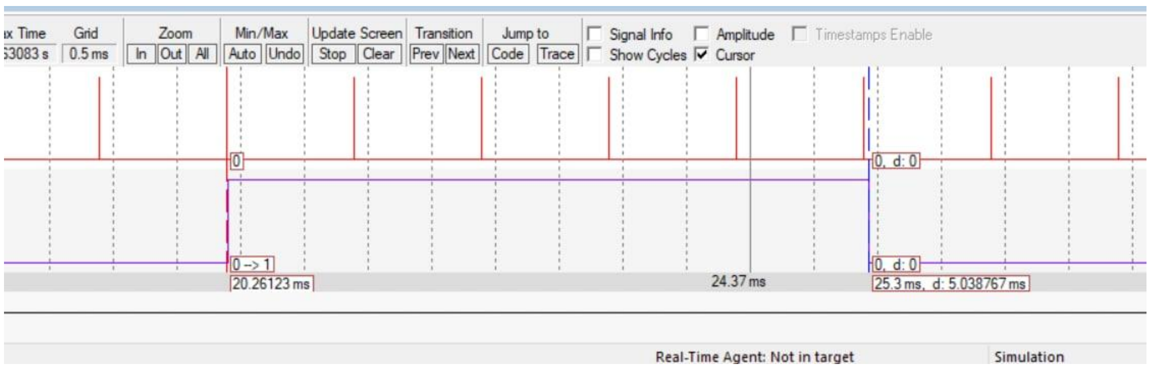
We devote six pins for tracing six task (from PIN1 to PIN6) , and PIN0 for the tick routine.



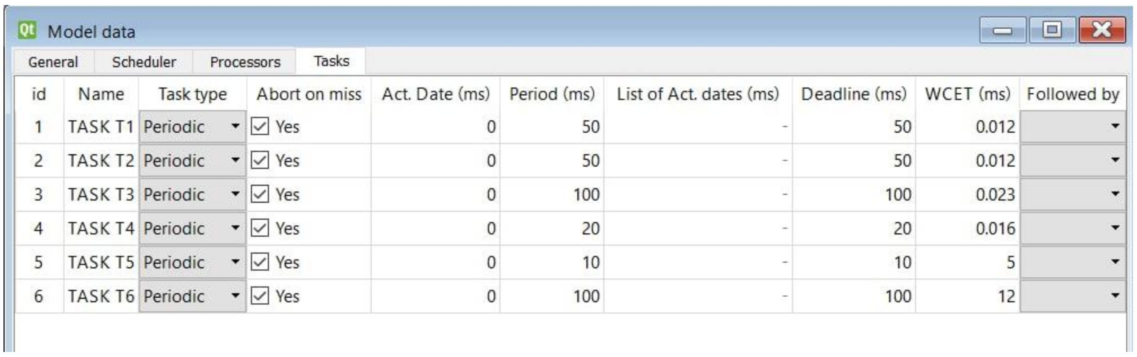
With using of logic analyzer we can see a real time line contain the six tasks and their switching in execution.

Hence, we can also tune our load simulation tasks to achieve the required execution time.
One has 5ms execution time and the other is 12 ms..



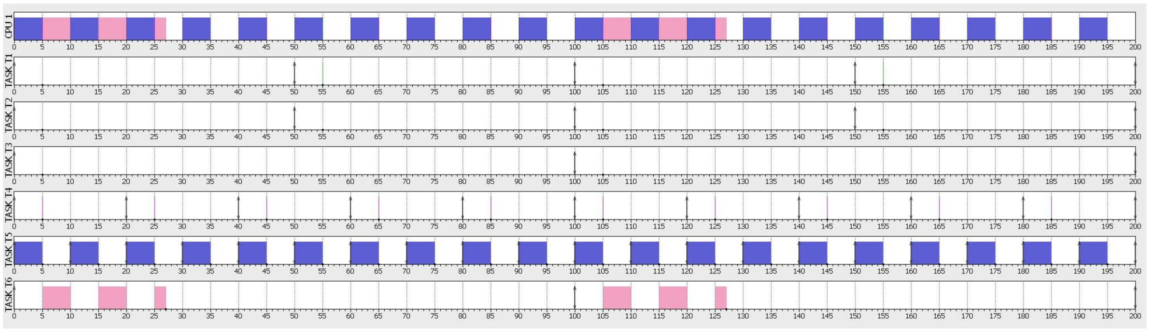


Also to verify our system correctness of scheduling we use the Simso tool to get another accurate Time-line which can be compared with ours.



id	Name	Task type	Abort on miss	Act. Date (ms)	Period (ms)	List of Act. dates (ms)	Deadline (ms)	WCET (ms)	Followed by
1	TASK T1	Periodic	☑ Yes	0	50	-	50	0.012	
2	TASK T2	Periodic	☑ Yes	0	50	-	50	0.012	
3	TASK T3	Periodic	☑ Yes	0	100	-	100	0.023	
4	TASK T4	Periodic	☑ Yes	0	20	-	20	0.016	
5	TASK T5	Periodic	☑ Yes	0	10	-	10	5	
6	TASK T6	Periodic	☑ Yes	0	100	-	100	12	

we calculate every task execution time in our system and configure Simso with these information to get us this illustration graph.

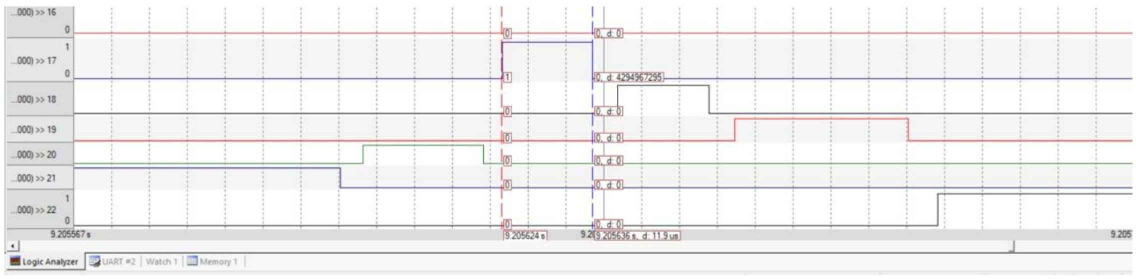


It's obvious that is seems identical to our Time-Line graph, hence our scheduler behaves correctly!

Another important calculation we have to consider is the system **hyperperiod** and **The CPU load**.

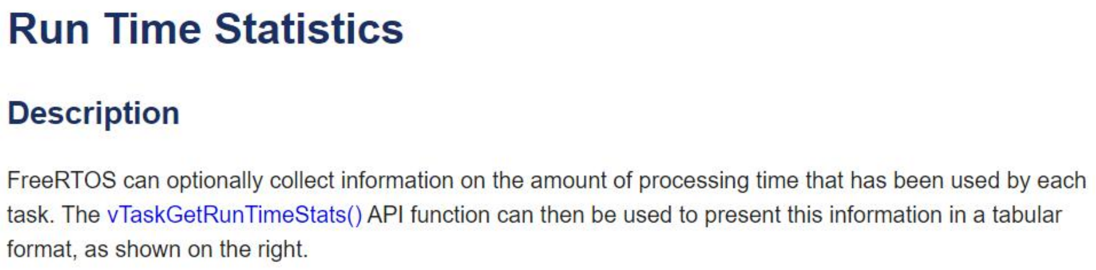
Hyperperiod is the point in where in each the system repeating it self, in other words, all tasks are to be scheduled in the same time. Hyperperiod period is usually The greatest common factor among all the tasks periods.
Here, our GCF is 100ms. And it's clear on the time line that after 100ms and 200ms and off course every 100ms all tasks are came to be scheduled in these moments.

And for the Task.5 to have the least deadline among other 5 tasks it’s scheduled to execute first.



Then to Calculate the CPU usage time we have to utilize a general purpose timer with frequency greater than the Tick Timer, in this purpose we use the internal Timer1 in our MCU.

And to facilitate the gathering of information we use the ready FreeRTOS API Function to do the job.

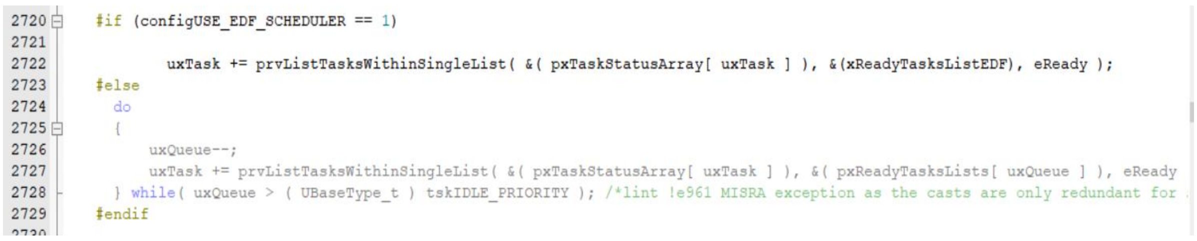


Run Time Statistics

Description

FreeRTOS can optionally collect information on the amount of processing time that has been used by each task. The vTaskGetRunTimeStats() API function can then be used to present this information in a tabular format, as shown on the right.

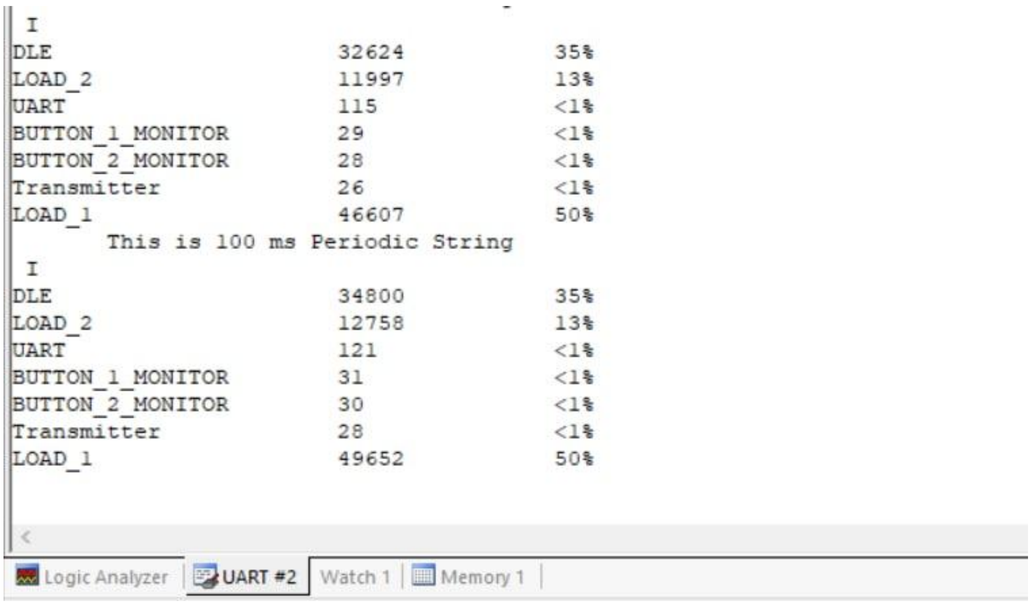
But first we should edit the implementation of the function ***uxTaskGetSystemState()*** Which this API depend on to collect statistics for the system, we edit it to collect the needed information from our EDF Ready List instead of the original Ready List in case of using the EDF scheduler as follows.



```
#if (configUSE_EDF_SCHEDULER == 1)

        uxTask += prvListTasksWithinSingleList( &( pxTaskStatusArray[ uxTask ] ), &(xReadyTasksListEDF), eReady );
#else
  do
  {
      uxQueue--;
      uxTask += prvListTasksWithinSingleList( &( pxTaskStatusArray[ uxTask ] ), &( pxReadyTasksLists[ uxQueue ] ), eReady
  } while( uxQueue > ( UBaseType_t ) tskIDLE_PRIORITY ); /*lint !e961 MISRA exception as the casts are only redundant for
#endif
```

After then we configure our general purpose timer at suitable frequency 10 times the tick frequency and define other needed macros for this API to work.

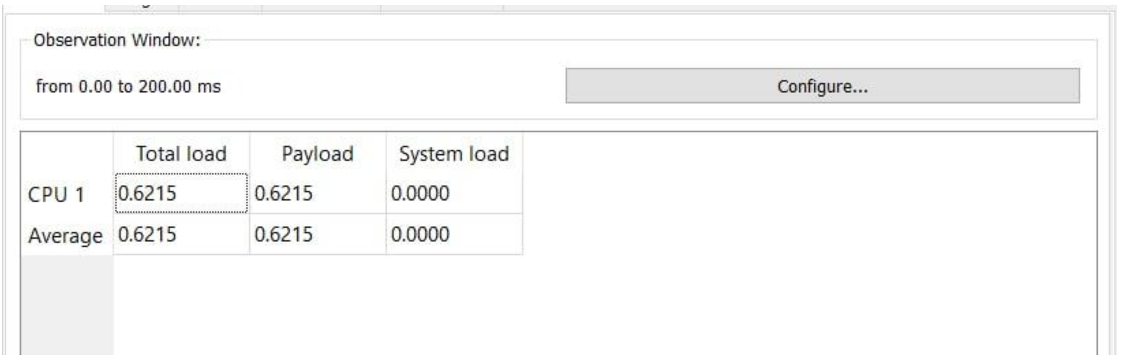
Calling this API function and transmit the formatted ASCII information by UART and displaying them on the terminal we get the follows:



```
 I
DLE                     32624           35%
LOAD_2                  11997           13%
UART                    115             <1%
BUTTON_1_MONITOR        29              <1%
BUTTON_2_MONITOR        28              <1%
Transmitter             26              <1%
LOAD_1                  46607           50%
       This is 100 ms Periodic String
 I
DLE                     34800           35%
LOAD_2                  12758           13%
UART                    121             <1%
BUTTON_1_MONITOR        31              <1%
BUTTON_2_MONITOR        30              <1%
Transmitter             28              <1%
LOAD_1                  49652           50%
```

IDLE task takes around 35% of the CPU time, the time that the CPU if free.
That's mean the CPU load is around 65% from the six tasks.

Verifying our statistics with the Simso tool we get so close observations from ours.



Observation Window:

from 0.00 to 200.00 ms

	Total load	Payload	System load
CPU 1	0.6215	0.6215	0.0000
Average	0.6215	0.6215	0.0000

All these results proves a successful implementation of our EDF scheduler.

Thank you.