SPRINTS

EDF Scheduler Implementation in FreeRTOS

A Real-time Systems Masterclass Graduation Project

Submitted by:

Mahmoud Hamdy

Cairo, 2022

- Contents:

- Introduction:

When dealing with real-time systems, there are many scheduling policies to choose from, depending on the application need, hence we make a proper Real-time operating system (RTOS) choice to realize that need. In this context, we are working with FreeRTOS the entire class, whose scheduling policy is very simple and flexible, it's a fixed priority preemptive round-robin scheduler, with some extended abilities to:

- Enable/Disable preemption.
- Enable/Disable time-sharing (RR policy).
- Change task priority in run-time (using APIs).
- Read task state in run-time (using APIs).

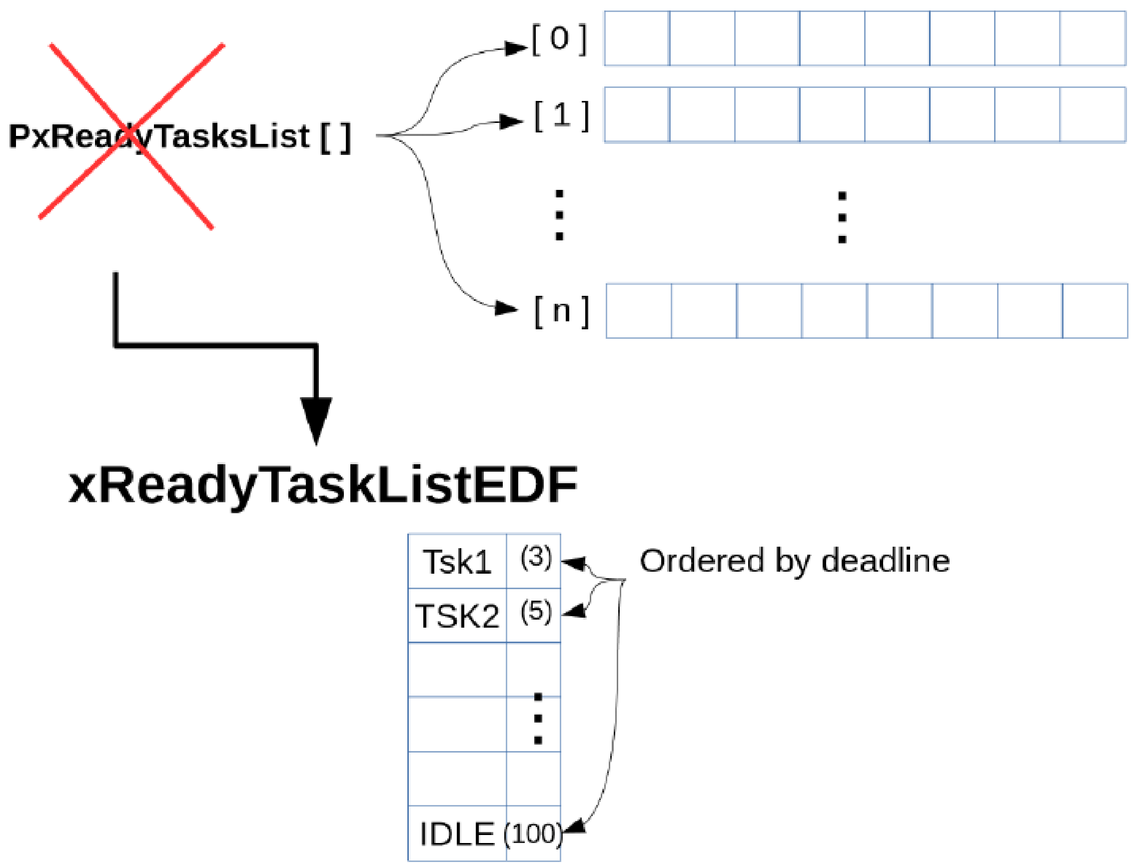
In this project, we try to make good use of the FreeRTOS features and alter its implementation to propose a new type of scheduling policy which is the earliest deadline first (EDF) scheduler, a preemptive scheduler whose task priorities are dynamically assigned - online at every scheduling point - based on deadlines, closer deadline is given higher priority and so on. The new implementation is inspired by a master's degree thesis in computer engineering from University of Padua, more details are mentioned in the "References" section.

I will use this thesis as our theoretical reference for 95% of the implementation and then start introducing new changes one by one (nearly the remaining 5%). Most (if not all) of the changes will be in tasks.c file in the FreeRTOS repository.

We then look at the example systems proposed and perform schedulability testing using three methods: analytically, using simso, and finally on keil simulator with the help of the logic analyzer implemented there. We finally make final comments on the results.

- EDF Implementation (the 95% changes):

First, a major change is proposed which is how we view the ready task queue, we are going to change it from an array of lists - whose indexes correspond to priority levels - to a list whose node positions correspond to the priority of each task with the head of the list having the highest priority (the closest deadline).



According to FreeRTOS coding standard and style guide, macros are prefixed with the file in which they are defined. The pre-fix is lower case. For example, configUSE_PREEMPTION is defined in FreeRTOSConfig.h. We use the same approach to create a macro as our configuration variable to enable/disable EDF scheduler, hence configUSE_EDF_SCHEDULER. When this macro is set to 1, EDF scheduler is used, else the OS uses the original scheduler.

```
#define configUSE_EDF_SCHEDULER 1
```

Then, the next changes are going to happen in tasks.c. The new Ready List is declared: xReadyTasksListEDF is a simple list structure.

```c
// EDF code
#if (configUSE_EDF_SCHEDULER == 1)
#define IDLE_PERIOD (TickType_t)100
PRIVILEGED_DATA static List_t xReadyTasksListEDF;
#endif
```

Then, prvInitialiseTaskLists method, that initializes all the task lists at the creation of the first task, is modified adding the initialization of xReadyTasksListEDF

```c
static void prvInitialiseTaskLists( void )
{
    UBaseType_t uxPriority;

    for( uxPriority = ( UBaseType_t ) 0U; uxPriority < ( UBaseTy
    {
        vListInitialise( &( pxReadyTasksLists[ uxPriority ] ) );
    }

    // EDF code
    #if (configUSE_EDF_SCHEDULER == 1)
    vListInitialise(&xReadyTasksListEDF);
    #endif

    vListInitialise( &xDelayedTaskList1 );
    vListInitialise( &xDelayedTaskList2 );
    vListInitialise( &xPendingReadyList );
```

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

```c
// EDF code: prvAddTaskToReadyList
#if (configUSE_EDF_SCHEDULER == 0)
#define prvAddTaskToReadyList( pxTCB )                                                          \
    traceMOVED_TASK_TO_READY_STATE( pxTCB );                                                    \
    taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority );                                         \
    vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( ( pxTCB )->xStateListItem ) ); \
    tracePOST_MOVED_TASK_TO_READY_STATE( pxTCB )
#else
#define prvAddTaskToReadyList( pxTCB )  /* xStateListItem must contain the deadline value */    \
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 xStateListItem (ascending item value order). So, it is assumed that xStateListItem contains the next task deadline.

Speaking of deadlines, time for a change in task structure, when a task moves to the Ready List, the knowledge of its next deadline is needed to insert it in the correct position. The deadline is calculated as:

Next deadline = current tick + task period

So, every task needs to store its period value. A new variable is added in the tskTaskControlBlock structure (TCB):

```
typedef struct tskTaskControlBlock       /* The old naming convention is
{
    volatile StackType_t * pxTopOfStack; /*< Points to the location of th

    #if ( portUSING_MPU_WRAPPERS == 1 )
        xMPU_SETTINGS xMPUSettings; /*< The MPU settings are defined as p
    #endif

    // EDF code: Period value to help in task deadline calculation
    #if (configUSE_EDF_SCHEDULER == 1)
    TickType_t xTaskPeriod; /*< Stores the period in tick of the task */
    #endif

    ListItem_t xStateListItem;                  /*< The list that the sta
    ListItem_t xEventListItem;                  /*< Used to reference a t
    UBaseType_t uxPriority;                     /*< The priority of the t
    StackType_t * pxStack;                      /*< Points to the start o
    char pcTaskName[ configMAX_TASK_NAME_LEN ]; /*< Descriptive name give
```

Accordingly, a new initialization task method is created. xTaskPeriodicCreate() is a modified version of the standard method xTaskGenericCreate(), that receives the task period as additional input parameter and set the xTaskPeriod variable in the task TCB structure.

```c
// EDF code: xTaskPeriodicCreate()
#if (configUSE_EDF_SCHEDULER == 1)
BaseType_t xTaskPeriodicCreate( TaskFunction_t pxTaskCode,
                                const char * const pcName, /*lint !e971 Unqualified char
                                const configSTACK_DEPTH_TYPE usStackDepth,
                                void * const pvParameters,
                                UBaseType_t uxPriority,
                                TaskHandle_t * const pxCreatedTask, TickType_t period )
{
    TCB_t * pxNewTCB;
    BaseType_t xReturn;
```

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.

```c
        // EDF code:
        pxNewTCB->xTaskPeriod = period; /* Initialize the period */
        /* Insert the period value in the xStateListItem before adding task to RL */
        listSET_LIST_ITEM_VALUE(&(pxNewTCB->xStateListItem), pxNewTCB->xTaskPeriod + xTaskGetTickCount());

        prvAddNewTaskToReadyList( pxNewTCB );
        xReturn = pdPASS;
    }
    else
    {
        xReturn = errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY;
    }

    return xReturn;
}
#endif /* xTaskPeriodicCreate() */
```

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 furthest deadline. vTaskStartScheduler() method initializes the IDLE task and inserts it into the Ready List. The method is modified as follows:

```
    #else /* if ( configSUPPORT_STATIC_ALLOCATION == 1 ) */
        {
            /* The Idle task is being created using dynamically

            // EDF code: Initialize IDLE task

            #if (configUSE_EDF_SCHEDULER == 1)
            xReturn = xTaskPeriodicCreate( prvIdleTask,
                                  configIDLE_TASK_NAME,
                                  configMINIMAL_STACK_SIZE,
                                  ( void * ) NULL,
                                  portPRIVILEGE_BIT,  /* In effect ( tskI
                                  &xIdleTaskHandle, /*lint !e961 MISRA ex
                                  IDLE_PERIOD);
            #else
            xReturn = xTaskCreate( prvIdleTask,
                                        configIDLE_TASK_NAME,
                                        configMINIMAL_STACK_SIZE,
                                        ( void * ) NULL,
                                        portPRIVILEGE_BIT,  /* In ef
                                        &xIdleTaskHandle ); /*lint !
            #endif
        }
```

The IDLE task is initialized with a period of IDLE_PERIOD = 100. We assume that no task can have a period greater than IDLE_PERIOD: 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 (Next deadline = current tick + task period, with current tick = 0 and task period = IDLE_PERIOD = 100, which is 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 to guarantee that IDLE task will remain in the last position of the Ready List. We cover this part in the next section (the 5% changes).

Last change needed involves the switch context mechanism. Every time the running task is suspended, or a suspended task with a 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 )
{
```

```
    /* Select a new task to run using either the generic C or port
     * optimised asm code. */

    // EDF code: Context switching to the earliest deadline task
    #if (configUSE_EDF_SCHEDULER == 0)
    taskSELECT_HIGHEST_PRIORITY_TASK(); /*lint !e9079 void * is used as this ma
    #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 95% of the pieces to get the new EDF scheduler work. In the next section will be looking at the remaining 5% of the changes which are not shown in the thesis but a must to have a fully implemented EDF scheduler.

- EDF Implementation: the 5% changes

First, when we create a new task, its TCB goes through some steps before being added to the Ready List, one of which is checking if the scheduler is not currently running, which often happens at the beginning system initialization. We check if this task has a higher priority (earlier deadline) than the current task and we update the current task based on this result but it has to be a comparison of deadlines not priorities, we add this change in prvAddNewTaskToReadyList():

```
static void prvAddNewTaskToReadyList( TCB_t * pxNewTCB )
{
```

```
            /* If the scheduler is not already running, make this task the
             * current task if it is the highest priority task to be created
             * so far. */
            if( xSchedulerRunning == pdFALSE )
            {
                // EDF code: select current task if scheduler is not running
                #if (configUSE_EDF_SCHEDULER == 0)
                    if( pxCurrentTCB->uxPriority <= pxNewTCB->uxPriority )
                    {
                        pxCurrentTCB = pxNewTCB;
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }
                #else
                    if( pxCurrentTCB->xStateListItem.xItemValue > pxNewTCB->xStateListItem.xItemValue )
                    {
                        pxCurrentTCB = pxNewTCB;
                    }
                    else
                    {
                        mtCOVERAGE_TEST_MARKER();
                    }
                #endif
            }
```

```
}
/*-----------------------------------------------------------*/
```

We must also update the deadline for the idle task each time it executes so that we can always keep it at the lowest priority:

```
static portTASK_FUNCTION( prvIdleTask, pvParameters )
{
    /* Stop warnings. */
    ( void ) pvParameters;

    /** THIS IS THE RTOS IDLE TASK - WHICH IS CREATED AUTOMATICALLY WHEN
     * SCHEDULER IS STARTED. **/

    /* In case a task that has a secure context deletes itself, in which
     * the idle task is responsible for deleting the task's secure contex
     * any. */
    portALLOCATE_SECURE_CONTEXT( configMINIMAL_SECURE_STACK_SIZE );

    for( ; ; )
    {
        // EDF code: Update Idle Task Deadline
        pxCurrentTCB->xStateListItem.xItemValue += (TickType_t)100;

        /* See if any tasks have deleted themselves - if so then the idle
         * is responsible for freeing the deleted task's TCB and stack. *
        prvCheckTasksWaitingTermination();
```

Finally, one of the most crucial changes is when a task unblocks (moves from delayed list to ready list), we must check if there should be a context switching based on our new scheduler. These changes are added to the xTaskIncrementTick() function:

```
BaseType_t xTaskIncrementTick( void )
{
    TCB_t * pxTCB;
    TickType_t xItemValue;
    BaseType_t xSwitchRequired = pdFALSE;
```

We first update the task's new deadline after unblocking and before adding to the ready list:

```c
// EDF code: update task new deadline before adding to ready list
#if (configUSE_EDF_SCHEDULER == 1)
    listSET_LIST_ITEM_VALUE(&(pxTCB->xStateListItem), pxTCB->xTaskPeriod + listGET_LIST_ITEM_VALUE(&(pxTCB->xStateListItem)));
    //pxTCB->xStateListItem.xItemValue = xItemValue + pxTCB->xTaskPeriod;
#endif

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

Then we check (based on how our new Ready List looks like after adding the new task) if there should be a context switching by comparing with the current task's deadline (determining the new currentTCB based on currentTCB's deadline):

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

/* A task being unblocked cannot cause an immediate
 * context switch if preemption is turned off. */
#if ( configUSE_PREEMPTION == 1 )
    {
        /* Preemption is on, but a context switch should
         * only be performed if the unblocked task has a
         * priority that is equal to or higher than the
         * currently executing task. */

```

```c
        // EDF code: Preemtion decision based on deadline not priority
        #if (configUSE_EDF_SCHEDULER == 1)
            if( listGET_LIST_ITEM_VALUE(&(pxTCB->xStateListItem)) <= listGET_LIST_ITEM_VALUE(&(pxCurrentTCB->xStateListItem)) )
            {
                xSwitchRequired = pdTRUE;
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        #else
            if( pxTCB->uxPriority >= pxCurrentTCB->uxPriority )
            {
                xSwitchRequired = pdTRUE;
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        #endif
    }
#endif /* configUSE_PREEMPTION */
```

It’s also worth mentioning that a small change is required in list.c:

```c
void vListInsert( List_t * const pxList,
                  ListItem_t * const pxNewListItem )
{
```

```c
// EDF code: Fine adjustments to the EDF preemption according to the thesis requirements
#if(configUSE_EDF_SCHEDULER == 1)
for( pxIterator = ( ListItem_t * ) &( pxList->xListEnd ); pxIterator->pxNext->xItemValue < xValueOfInsertion; pxIterator = pxIterator->pxNext )
{
    /* There is nothing to do here, just iterating to the wanted
     * insertion position. */
}
#else
for( pxIterator = ( ListItem_t * ) &( pxList->xListEnd ); pxIterator->pxNext->xItemValue <= xValueOfInsertion; pxIterator = pxIterator->pxNext )
{
    /* There is nothing to do here, just iterating to the wanted
     * insertion position. */
}
#endif
```

- Our real-time example systems

System 1:

	T	C
Task A	5	2
Task B	8	2

System 2:

	T	C
Task A	5	3
Task B	8	3

Tick time for both systems = H.C.F(5, 8) = 1 ms

Hyper-period for both systems = L.C.M(5, 8) = 5 x 8 = 40 ms

- Schedulability testing analytically:

From thesis:

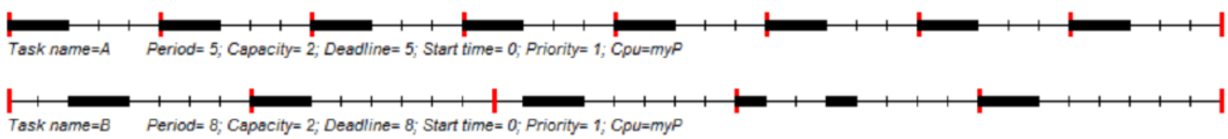
Theorem 3.1 *A task set of periodic tasks is schedulable by EDF if and only if:*

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

System 1:

U = C1/T1 + C2/T2 = 2/5 + 2/8 = 16/40 + 10/40 = 26/40 = 65% = 0.65 < 1

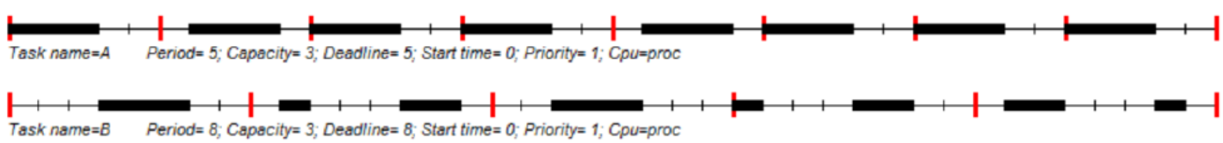
Hence, system is healthy and schedulable by EDF according to the theorem.



System 2:

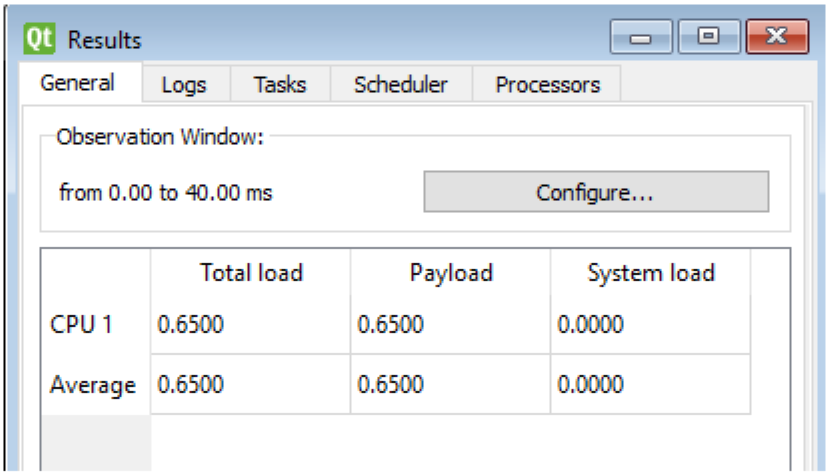
U = C1/T1 + C2/T2 = 3/5 + 3/8 = 24/40 + 15/40 = 39/40 = 97.5% = 0.975 < 1

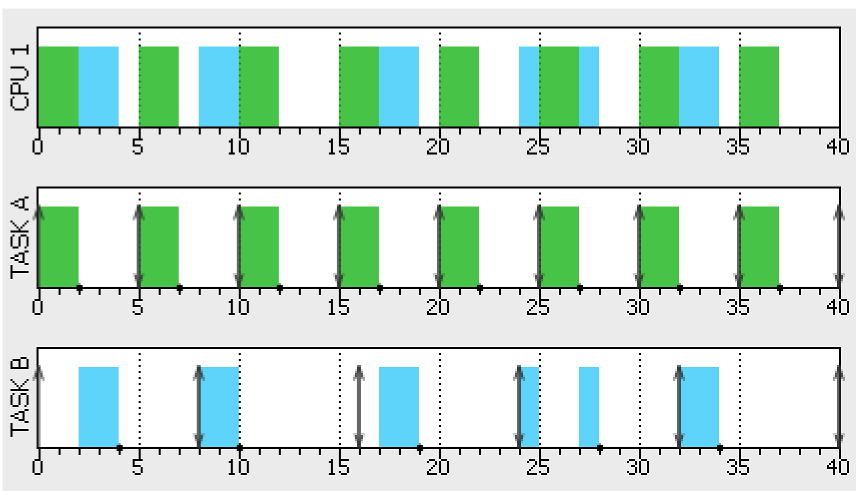
Hence, system is healthy and schedulable by EDF according to the theorem.



- Schedulability testing using simso:

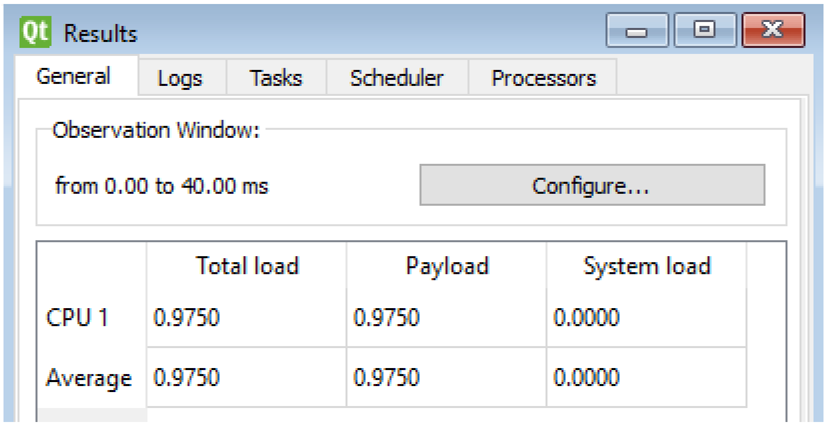
System 1:

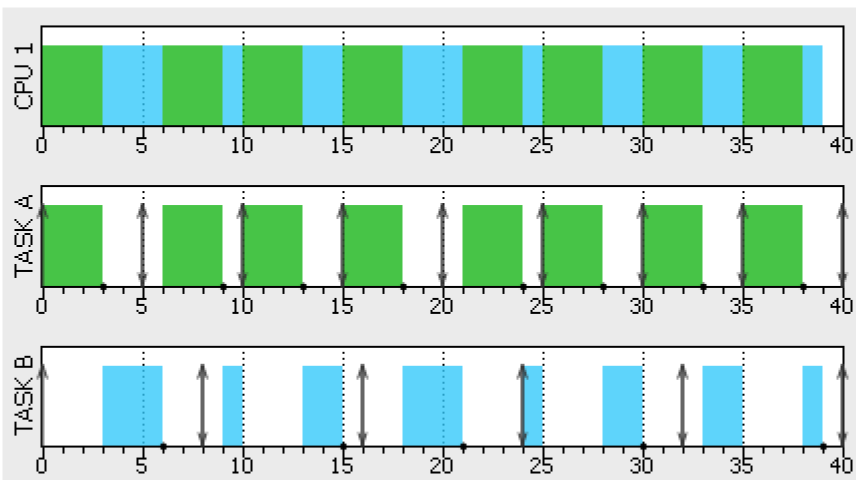




Results on simso match the analytical results.

System 2:





Results on simso (using EDF2) match the analytic results.

- Schedulability testing using keil:

In this simulation we use GPIO pins with our system hooks using the following configuration:

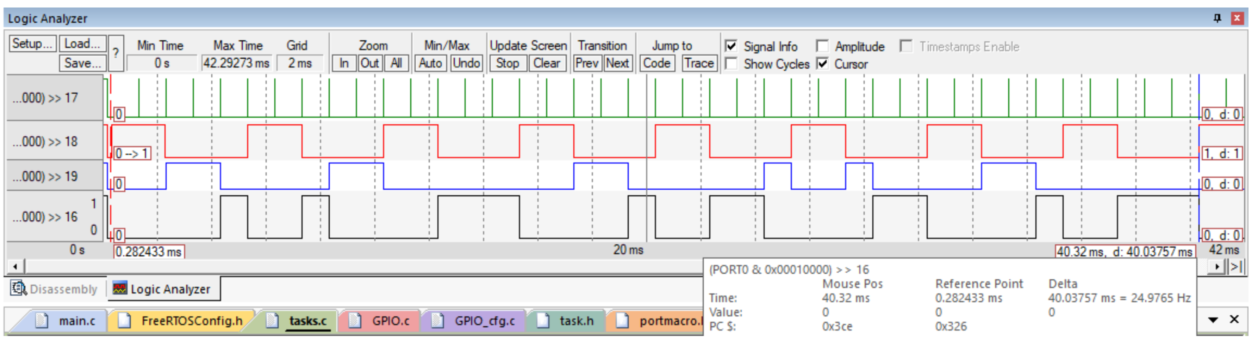
Port 0 pin 17 (PIN1) → Refers to the system tick

Port 0 pin 18 (PIN2) → Refers to Task A

Port 0 pin 19 (PIN3) → Refers to Task B

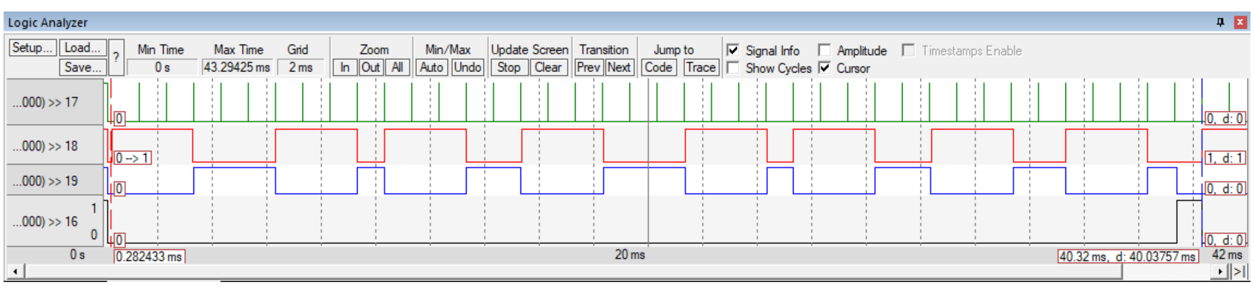
Port 0 pin 16 (PIN0) → Refers to the IDLE Task

System 1:



Results on keil match simso and the analytical.

System 2:



Results on keil match simso and the analytical.

- Conclusion:

According to the results obtained, we can safely say that system 1 and system 2 are both healthy (with one more healthy than the other) and schedulable in a dynamic scheduling policy namely the EDF scheduler and this was proved analytically using the theorem of EDF scheduling, using a simulator like Simso, and finally using real-time code on keil environment with a powerful software simulator and logic analyzer.

- References:

Carraro, Enrico (2016), Implementation and Test of EDF and LLREF Schedulers in FreeRTOS, Padua @ thesis, accessed 20 July 2021, http://tesi.cab.unipd.it/51896/1/Implementation_and_Test_of_EDF_and_LLREF_Scgheduler_in_FreeRTOS.pdf

FreeRTOS, accessed 1 January 2022, https://www.freertos.org/FreeRTOS-Coding-Standard-and-Style-Guide.html