Final Report
Earliest Deadline First Scheduler on FreeRTOS

Shambhavi Kuthe
Virginia Tech

Rohit Mehta
Virginia Tech

1 Problem Statement and Motivation

This report explores the Earliest Deadline First Scheduling policy on FreeRTOS - an open source Real time operating system for various micro-controllers. Real-time systems have hard timing deadlines that need to be catered to in order to avoid catastrophe in Hard Real-time systems and system failures in soft real-time systems. Earliest Deadline First(EDF) is a scheduling policy that takes the deadlines of the tasks into consideration and schedules the tasks such that tasks having the earliest absolute deadline are scheduled first. This policy therefore ensures that least amount of deadline misses occur by providing predictable behaviour of the system. FreeRTOS is a popular real-time operating systems providing multiple scheduling policies to manage the scheduling of tasks. Our aim is to implement the EDF scheduling policy which will cater to the applications that ensures the earliest deadline task is always scheduled first.

2 Scheduling Policies

2.1 Fixed Priority Scheduling Policies

Fixed Priority Scheduling is a scheduling policy used in real-time systems where the priorities are fixed beforehand. This policy ensures that at any given time, the processor executes the highest priority task among all the tasks that are in the ready state before running. In fixed priority scheduling, each task is assigned a priority based on some differentiating parameter related to the task. For e.g. Period, Deadline etc. Fixed Priority Scheduling being is an easily implementable and efficient scheduling policy that provides a predictable behaviour of the system. However, one of the major drawbacks of the setting fixed priorities is that it does not guarantee that deadlines of tasks will be met. Moreover, if the priorities are not assigned correctly, it may also lead to priority inversion and other related issues which might lead to an unpredictable behaviour of the system.

Two of the Fixed Priority Scheduling policies considered in our evaluation are:

2.1.1 Rate Monotonic Scheduling

In Rate Monotonic Scheduling, tasks are managed on the basis of their periodicity i.e. priorities are assigned to tasks by taking the periods of each tasks into consideration. Rate Monotonic priority assignment is based on the principle that the tasks with shorter periods are assigned higher priority. This ensures that the tasks with shorter periods are executed more frequently. In this case, the deadlines are met if the the task's execution time is shorter that the task period. This policy ensures that deadlines of the tasks are met with minimum overhead. This policy also requires that at design time, the periods of the tasks are known and remain fixed throughout the execution which may limit its applicability in some real-time systems.

2.1.2 Deadline Monotonic Scheduling

In Deadline Monotonic Scheduling, tasks are managed on the basis of their relative deadlines i.e. priorities are assigned to tasks by taking the relative deadlines of each tasks into consideration. Deadline Monotonic priority assignment is based on the principle that the tasks with shorter relative deadlines are assigned higher priority. This ensures that the tasks with shorter relative deadlines are executed more frequently. This policy is better than Rate Monotonic scheduling policy in terms of efficiency and scheduling performance. This policy also requires that at design time, the relative deadlines of the tasks are known and remain fixed throughout the execution which may limit its applicability in some real-time systems. Additionally, Deadline Monotonic Scheduling Policy may suffer from priority inversion or blocking thus affecting the system performance.

2.2 Dynamic Priority Scheduling Policies

Dynamic Priority Scheduling is a scheduling policy used in real-time systems to manage tasks based on their execution characteristics and the system's current state. [2] The priority assignment is done dynamically based on the go as tasks are released. This scheduling policy allows the system to adapt to changes in task execution times and system load, improving system performance and meeting the task's deadlines. Dynamic Priority Scheduling is an efficient scheduling policy than fixed-priority scheduling policies in the sense that it provides better scheduling performance. However, this policy requires more system overhead than fixed-priority scheduling policies, as the priority levels of tasks need to be updated frequently based on the system's state. Additionally, it also requires accurate estimation of parameters that are being considered for selecting the dynamic priority assignment, which can be challenging in some real-time systems. Nevertheless, Dynamic Priority Scheduling can be useful in real-time systems with unpredictable execution times or systems that require better performance than fixed-priority scheduling policies.

The dynamic priority scheduling discussed, implemented and evaluated in this report is the Earliest Deadline First scheduling algorithm.

2.2.1 Earliest Deadline First

Earliest Deadline First (EDF) is a scheduling algorithm used in real-time operating systems to manage tasks based on their deadlines. In EDF, each task is assigned a deadline, and the task with the earliest deadline is scheduled first. EDF ensures that jobs with the earliest deadlines are completed first, increasing the likelihood that the tasks' deadlines will be met. In terms of minimizing the amount of missed deadlines, EDF is the best scheduling algorithm, but it necessitates precise calculation of job execution times and deadlines. EDF is commonly used in dynamic priority scheduling policies, where priorities are assigned based on the remaining time until a task's deadline. EDF is efficient and provides predictable behavior, making it a popular scheduling algorithm for real-time systems. [1]

3 Implementation

3.1 Extended TCB

To implement the scheduling policies, we use an extended version of the TCB known as SchedTCB which includes additional information to manage various tasks using our scheduling policy implementation. pvPortMalloc() is used to assign memory to the SchedTCB struct. SchedTCB consists of the following members:

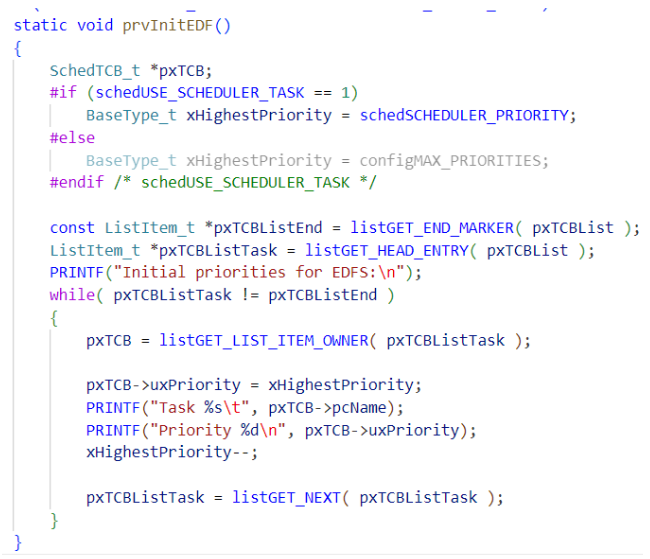
- pvTaskCode : Function pointer to the code that will be run periodically.
- pcName : Name of the task.
- uxStackDepth : Stack size of the task.
- pvParameters : Parameters to the task function.
- uxPriority : Priority of the task.
- pxTaskHandle : Task handle for the task.
- xReleaseTime : Release time of the task.
- xRelativeDeadline : Relative deadline of the task.
- xAbsoluteDeadline : Absolute deadline of the task.
- xPeriod : Task period.
- xLastWakeTime : Last time stamp when the task was running.
- xMaxExecTime : Worst-case execution time of the task.
- xExecTime : Current execution time of the task.
- xWorkIsDone: pdFALSE if the job is not finished, pdTRUE if the job is finished.
- xTCBListTask : Tasks in the List
- xExecutedOnce : pdTRUE if the task has executed once.
- xAbsoluteUnblockTime : The task will be unblocked at this time if it is blocked by the scheduler task.
- xSuspended : pdTRUE if the task is suspended.
- xMaxExecTimeExceeded : pdTRUE when execTime exceeds maxExecTime.

3.2 Algorithm

For the implementation of the EDF scheduler, we have decided to go for a linked list approach. The linked list data structure is used to store the arriving tasks sorted by the value of their deadlines. The list is kept in sorted order, with the assignment having the earliest deadline at the top. The linked list-based method makes it efficient to add or remove jobs from the list, which is essential in a real-time operating system. When a task is created, it is added to the list based on its deadline. The scheduler selects the task at the head of the list to be executed, as it has the earliest deadline. If a task's deadline changes while it is waiting to be executed, the task is removed from the list and reinserted into the list based on its new deadline. The linked list-based approach allows the scheduler to maintain a list of tasks with their deadlines, ensuring that the tasks are executed in the order that minimizes the number of missed deadlines.

Thread local storage(TLS) allows the application writer to store values inside a task's control block, making the value specific to the task itself, and allowing each task to have its own unique value. Thread local storage is most often used to store values that a single threaded program would otherwise store in a global variable. In our implementation, we are using the TLS to store the SchedTCB task corresponding to its handle. The thread local storage is used to extract the SchedTCB task when the corresponding task handle is known. Whenever a task is created or recreated, the SchedTCB task is stored for its corresponding task handle and the storage is freed using vPortFree() when a task is removed from the list.

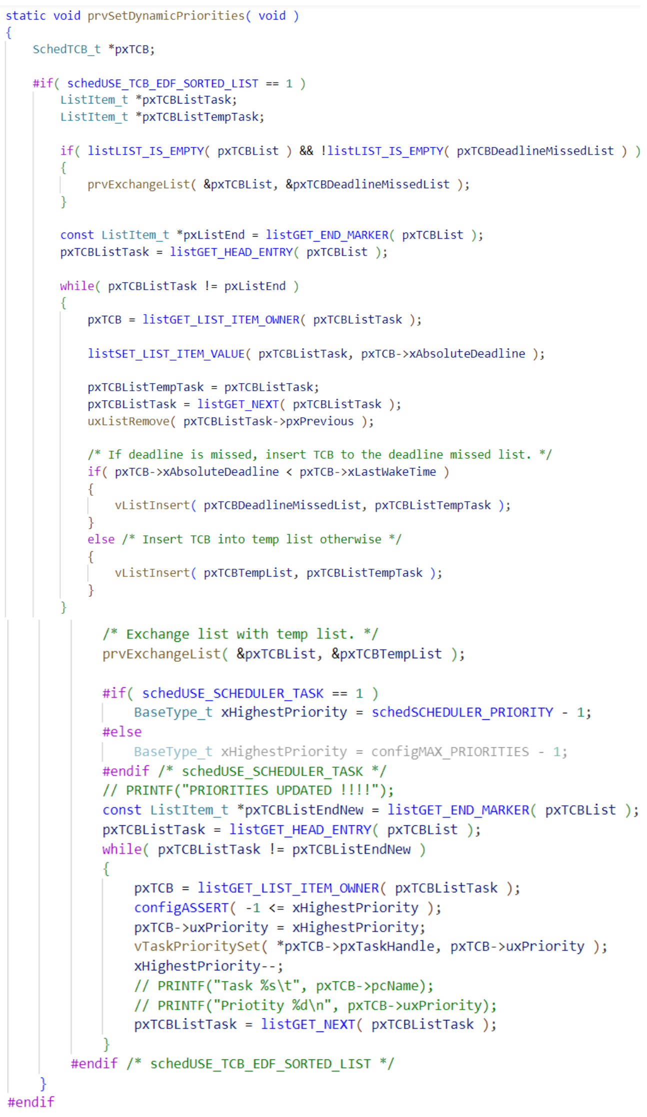
- **prvinitEDF() :** This function is used to set the initial priorities of the tasks. This is done by traversing through list of tasks and assigning the highest priority to the first task in the sorted list and then reducing the priority for the next tasks. *listGET_END_MARKER()* is the macro to get the last default task in the list which marks the end of the list. *listGET_HEAD_ENTRY()* macro gets the pointer to the first SchedTCB task in the list. *listGET_LIST_ITEM_OWNER()* is used to get the owner of the SchedTCB task in the list. *listGET_NEXT()* macro is used to get the next task in the list. The priority of the SchedTCB task is denoted by uxPriority which is assigned the xHighestPriority before decrementing it.



```
static void prvInitEDF()
{
    SchedTCB_t *pxTCB;
    #if (schedUSE_SCHEDULER_TASK == 1)
        BaseType_t xHighestPriority = schedSCHEDULER_PRIORITY;
    #else
        BaseType_t xHighestPriority = configMAX_PRIORITIES;
    #endif /* schedUSE_SCHEDULER_TASK */

    const ListItem_t *pxTCBListEnd = listGET_END_MARKER( pxTCBList );
    ListItem_t *pxTCBListTask = listGET_HEAD_ENTRY( pxTCBList );
    PRINTF("Initial priorities for EDFS:\n");
    while( pxTCBListTask != pxTCBListEnd )
    {
        pxTCB = listGET_LIST_ITEM_OWNER( pxTCBListTask );

        pxTCB->uxPriority = xHighestPriority;
        PRINTF("Task %s\t", pxTCB->pcName);
        PRINTF("Priority %d\n", pxTCB->uxPriority);
        xHighestPriority--;

        pxTCBListTask = listGET_NEXT( pxTCBListTask );
    }
}
```

Figure 1: prvinitEDF() function

- **prvSetDynamicPriorities() :** This function is the main function that is used to set the dynamic priorities of the tasks based on their absolute deadlines. *listLIST_IS_EMPTY()* macro checks if the said list is empty and is used when the *pxTCBDeadlineMissedList* is full and *pxTCBList* is empty. In this case, the two lists are exchanged using the *prvExchangeList()* function which swaps the contents of the two lists given in its parameters. The list is then traversed and for every task in the list, the item value of the task is set to the absolute deadline of that task. The FreeRTOS linked list is sorted according to this item value parameter of the list. Thus, after setting the item value according to the absolute deadline, that task is removed from the list using *uxListRemove()* function and then inserted into a temporary list: *pxTCBListTempTask*. After setting the item values of each task in the list, the *pxTCBListTempTask* and *pxTCBList* are exchanged such that *pxTCBList* is re-sorted according to the updated deadlines and additional tasks. After re-sorting, the *uxpriority* of all the tasks are again set such that the first task in the sorted list is given the highest priority.



```
static void prvSetDynamicPriorities( void )
{
    SchedTCB_t *pxTCB;

    #if( schedUSE_TCB_EDF_SORTED_LIST == 1 )
        ListItem_t *pxTCBListTask;
        ListItem_t *pxTCBListTempTask;

        if( listLIST_IS_EMPTY( pxTCBList ) && !listLIST_IS_EMPTY( pxTCBDeadlineMissedList ) )
        {
            prvExchangeList( &pxTCBList, &pxTCBDeadlineMissedList );
        }

        const ListItem_t *pxListEnd = listGET_END_MARKER( pxTCBList );
        pxTCBListTask = listGET_HEAD_ENTRY( pxTCBList );

        while( pxTCBListTask != pxListEnd )
        {
            pxTCB = listGET_LIST_ITEM_OWNER( pxTCBListTask );

            listSET_LIST_ITEM_VALUE( pxTCBListTask, pxTCB->xAbsoluteDeadline );

            pxTCBListTempTask = pxTCBListTask;
            pxTCBListTask = listGET_NEXT( pxTCBListTask );
            uxListRemove( pxTCBListTask->pxPrevious );

            /* If deadline is missed, insert TCB to the deadline missed list. */
            if( pxTCB->xAbsoluteDeadline < pxTCB->xLastWakeTime )
            {
                vListInsert( pxTCBDeadlineMissedList, pxTCBListTempTask );
            }
            else /* Insert TCB into temp list otherwise */
            {
                vListInsert( pxTCBTempList, pxTCBListTempTask );
            }
        }
            /* Exchange list with temp list. */
            prvExchangeList( &pxTCBList, &pxTCBTempList );

            #if( schedUSE_SCHEDULER_TASK == 1 )
                BaseType_t xHighestPriority = schedSCHEDULER_PRIORITY - 1;
            #else
                BaseType_t xHighestPriority = configMAX_PRIORITIES - 1;
            #endif /* schedUSE_SCHEDULER_TASK */
            // PRINTF("PRIORITIES UPDATED !!!!");
            const ListItem_t *pxTCBListEndNew = listGET_END_MARKER( pxTCBList );
            pxTCBListTask = listGET_HEAD_ENTRY( pxTCBList );
            while( pxTCBListTask != pxTCBListEndNew )
            {
                pxTCB = listGET_LIST_ITEM_OWNER( pxTCBListTask );
                configASSERT( -1 <= xHighestPriority );
                pxTCB->uxPriority = xHighestPriority;
                vTaskPrioritySet( *pxTCB->pxTaskHandle, pxTCB->uxPriority );
                xHighestPriority--;
                // PRINTF("Task %s\t", pxTCB->pcName);
                // PRINTF("Priotity %d\n", pxTCB->uxPriority);
                pxTCBListTask = listGET_NEXT( pxTCBListTask );
            }
        #endif /* schedUSE_TCB_EDF_SORTED_LIST */
    }
#endif
```

Figure 2: prvSetDynamicPriorities() function

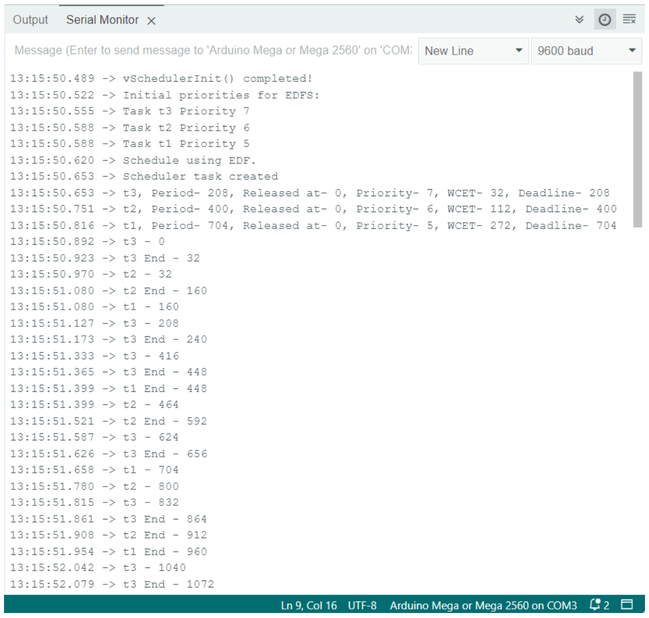
4 Output

The implementation has been demonstrated on the following task set :

Tasks	Release Time	WCET	Period	Deadline
T1	0	280	700	700
T2	0	120	400	400
T3	0	40	200	200

Figure 3: Task Set 1

The task set is implemented in the Arduino IDE and tested on ATMega2560 board. The screenshot shows the running of the tasks scheduled according to our implementation of the EDF algorithm and has been verified manually.


Output
Serial Monitor
Message (Enter to send message to 'Arduino Mega or Mega 2560' on 'COM:
New Line
9600 baud
Ln 9, Col 16
UTF-8
Arduino Mega or Mega 2560 on COM3
2

```
13:15:50.489 -> vSchedulerInit() completed!
13:15:50.522 -> Initial priorities for EDFS:
13:15:50.555 -> Task t3 Priority 7
13:15:50.588 -> Task t2 Priority 6
13:15:50.588 -> Task t1 Priority 5
13:15:50.620 -> Schedule using EDF.
13:15:50.653 -> Scheduler task created
13:15:50.653 -> t3, Period- 208, Released at- 0, Priority- 7, WCET- 32, Deadline- 208
13:15:50.751 -> t2, Period- 400, Released at- 0, Priority- 6, WCET- 112, Deadline- 400
13:15:50.816 -> t1, Period- 704, Released at- 0, Priority- 5, WCET- 272, Deadline- 704
13:15:50.892 -> t3 - 0
13:15:50.923 -> t3 End - 32
13:15:50.970 -> t2 - 32
13:15:51.080 -> t2 End - 160
13:15:51.080 -> t1 - 160
13:15:51.127 -> t3 - 208
13:15:51.173 -> t3 End - 240
13:15:51.333 -> t3 - 416
13:15:51.365 -> t3 End - 448
13:15:51.399 -> t1 End - 448
13:15:51.399 -> t2 - 464
13:15:51.521 -> t2 End - 592
13:15:51.587 -> t3 - 624
13:15:51.626 -> t3 End - 656
13:15:51.658 -> t1 - 704
13:15:51.780 -> t2 - 800
13:15:51.815 -> t3 - 832
13:15:51.861 -> t3 End - 864
13:15:51.908 -> t2 End - 912
13:15:51.954 -> t1 End - 960
13:15:52.042 -> t3 - 1040
13:15:52.079 -> t3 End - 1072
```

Figure 4: EDF practical output for Task set 1

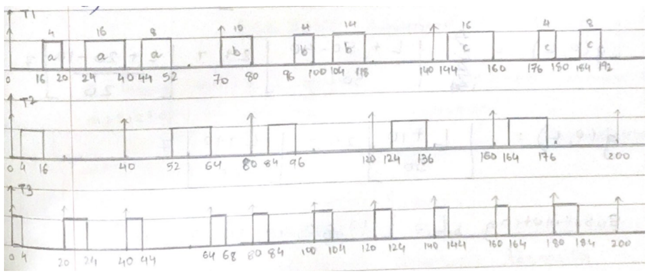


Figure 5: EDF theoretical output for Task set 1

The output shows the task name followed by the time in milliseconds at which it is released. The task name followed by "end" denotes the completion of the task followed by the time in milliseconds at which it ends. According to the theoretical calculations, it can be seen in the output that the tasks are scheduled in the order such that t3 is executed completely, followed by t2 and then followed by t1. Also, t1 gets preempted by t3 as its absolute deadline is earlier than t1. It is also observed that none of the tasks should miss their deadlines and should not exceed their worst case execution times. As the theoretical calculations match the practical output, it can be verified that our implementation of the EDF algorithm is working correctly.

5 Evaluation

Our implementation of the EDF dynamic priority scheduling algorithm is evaluated against the Rate-monotonic and Deadline-monotonic fixed priority scheduling algorithms. To demonstrate the difference between the EDF, RM and DM scheduling policies, we use the following dataset.

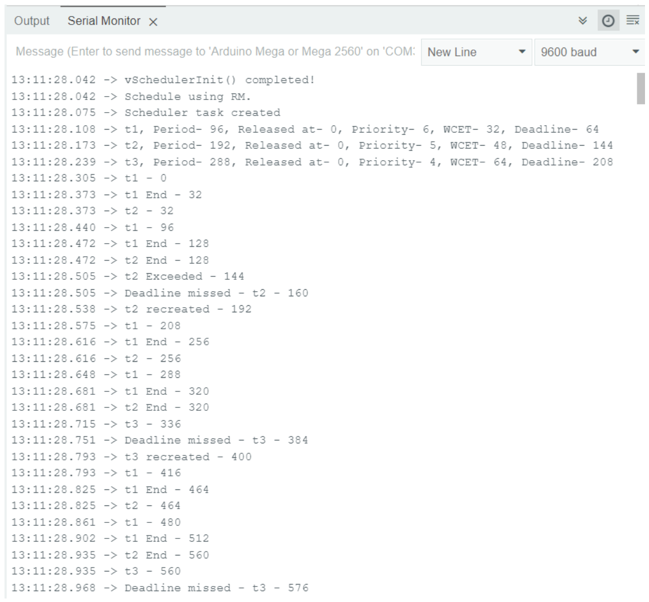
Tasks	Release Time	WCET	Period	Deadline
T1	0	40	100	80
T2	0	60	200	160
T3	0	80	300	220

Figure 6: Task set 2

The Rate-monotonic and Deadline-monotonic scheduling algorithms are implemented in the same file and can be selected from the scheduler.h file by setting the appropriate policy in schedSCHEDULING_POLICY. The output of each of the policies is shown below.

5.1 Rate-monotonic Scheduling

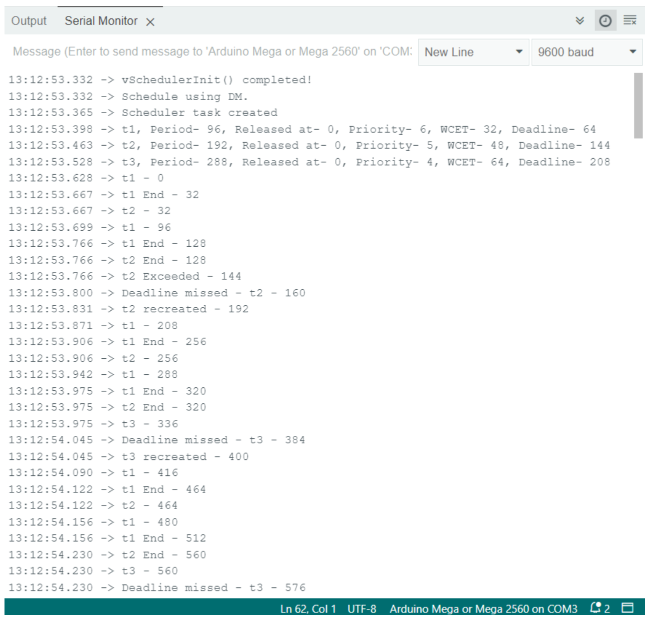
It can be seen in the output of RM that for Task set 2, tasks t2 and t3 miss their deadlines more frequently than EDF. This is because tasks with shorter periods are scheduled more frequently i.e. t1 is scheduled frequently making t2 and t3 with larger periods to miss their deadlines eventually. Thus, for RM scheduling, it can be inferred that the deadlines can only be met if the system load is within a specific limit.


Output
Serial Monitor
Message (Enter to send message to 'Arduino Mega or Mega 2560' on 'COM:
New Line
9600 baud

```
13:11:28.042 -> vSchedulerInit() completed!
13:11:28.042 -> Schedule using RM.
13:11:28.075 -> Scheduler task created
13:11:28.108 -> t1, Period- 96, Released at- 0, Priority- 6, WCET- 32, Deadline- 64
13:11:28.173 -> t2, Period- 192, Released at- 0, Priority- 5, WCET- 48, Deadline- 144
13:11:28.239 -> t3, Period- 288, Released at- 0, Priority- 4, WCET- 64, Deadline- 208
13:11:28.305 -> t1 - 0
13:11:28.373 -> t1 End - 32
13:11:28.373 -> t2 - 32
13:11:28.440 -> t1 - 96
13:11:28.472 -> t1 End - 128
13:11:28.472 -> t2 End - 128
13:11:28.505 -> t2 Exceeded - 144
13:11:28.505 -> Deadline missed - t2 - 160
13:11:28.538 -> t2 recreated - 192
13:11:28.575 -> t1 - 208
13:11:28.616 -> t1 End - 256
13:11:28.616 -> t2 - 256
13:11:28.648 -> t1 - 288
13:11:28.681 -> t1 End - 320
13:11:28.681 -> t2 End - 320
13:11:28.715 -> t3 - 336
13:11:28.751 -> Deadline missed - t3 - 384
13:11:28.793 -> t3 recreated - 400
13:11:28.793 -> t1 - 416
13:11:28.825 -> t1 End - 464
13:11:28.825 -> t2 - 464
13:11:28.861 -> t1 - 480
13:11:28.902 -> t1 End - 512
13:11:28.935 -> t2 End - 560
13:11:28.935 -> t3 - 560
13:11:28.968 -> Deadline missed - t3 - 576
```

Figure 7: RM output for Task set 2

5.2 Deadline-monotonic Scheduling


Output
Serial Monitor
Message (Enter to send message to 'Arduino Mega or Mega 2560' on 'COM:
New Line
9600 baud

```
13:12:53.332 -> vSchedulerInit() completed!
13:12:53.332 -> Schedule using DM.
13:12:53.365 -> Scheduler task created
13:12:53.398 -> t1, Period- 96, Released at- 0, Priority- 6, WCET- 32, Deadline- 64
13:12:53.463 -> t2, Period- 192, Released at- 0, Priority- 5, WCET- 48, Deadline- 144
13:12:53.528 -> t3, Period- 288, Released at- 0, Priority- 4, WCET- 64, Deadline- 208
13:12:53.628 -> t1 - 0
13:12:53.667 -> t1 End - 32
13:12:53.667 -> t2 - 32
13:12:53.699 -> t1 - 96
13:12:53.766 -> t1 End - 128
13:12:53.766 -> t2 End - 128
13:12:53.766 -> t2 Exceeded - 144
13:12:53.800 -> Deadline missed - t2 - 160
13:12:53.831 -> t2 recreated - 192
13:12:53.871 -> t1 - 208
13:12:53.906 -> t1 End - 256
13:12:53.906 -> t2 - 256
13:12:53.942 -> t1 - 288
13:12:53.975 -> t1 End - 320
13:12:53.975 -> t2 End - 320
13:12:53.975 -> t3 - 336
13:12:54.045 -> Deadline missed - t3 - 384
13:12:54.045 -> t3 recreated - 400
13:12:54.090 -> t1 - 416
13:12:54.122 -> t1 End - 464
13:12:54.122 -> t2 - 464
13:12:54.156 -> t1 - 480
13:12:54.156 -> t1 End - 512
13:12:54.230 -> t2 End - 560
13:12:54.230 -> t3 - 560
13:12:54.230 -> Deadline missed - t3 - 576
```

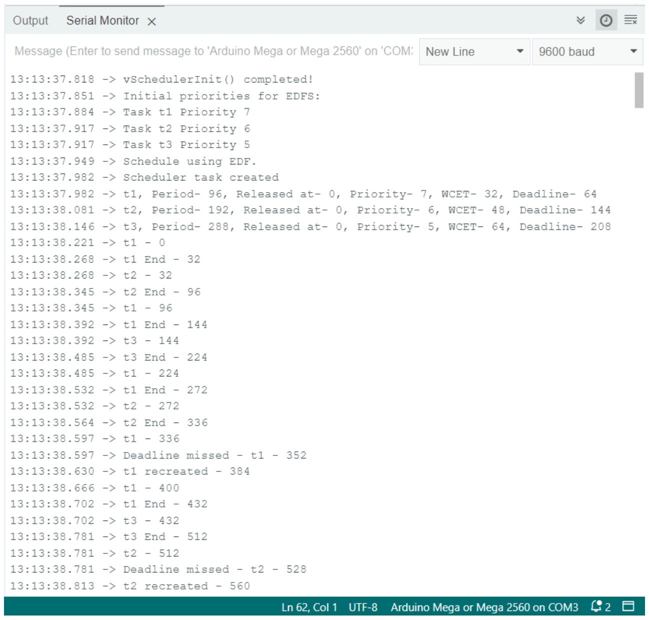
Ln 62, Col 1   UTF-8   Arduino Mega or Mega 2560 on COM3   2

Figure 8: DM output for Task set 2

It can be seen in the output of DM that for Task set 2, tasks t2 and t3 miss their deadlines more frequently than EDF. This is because tasks with shorter deadlines are scheduled more frequently i.e. t1 is scheduled frequently making t2 and t3 with farther deadlines to miss their deadlines eventually. Thus, it can be inferred that DM scheduling guarantees that tasks with shorter deadlines will meet their hard deadlines but the other lower priority tasks may starve if the high priority task is perpetually released.

5.3 Earliest deadline first Scheduling

It can be seen in the output of EDF that for Task set 2, tasks t2 and t3 miss their deadlines less frequently as compared to RM and DM. Even though t1 is missing its deadline, it can be seen that in a definite time span, the number of deadline misses occurring during EDF scheduling is less than RM and DM scheduling in the same time span as indicated by the time stamps on the Serial monitor of Arduino IDE. Thus, it can be inferred that EDF enhances the system performance by minimizing the number of deadline misses at the cost of requiring accurate estimation of absolute deadlines and frequent calculation of the priorities during the execution. This also proves that dynamic priority scheduling policies are better suited for some task sets than fixed priority scheduling policies.


Output
Serial Monitor
Message (Enter to send message to 'Arduino Mega or Mega 2560' on 'COM:
New Line
9600 baud

```
13:13:37.818 -> vSchedulerInit() completed!
13:13:37.851 -> Initial priorities for EDFS:
13:13:37.884 -> Task t1 Priority 7
13:13:37.917 -> Task t2 Priority 6
13:13:37.917 -> Task t3 Priority 5
13:13:37.949 -> Schedule using EDF.
13:13:37.982 -> Scheduler task created
13:13:37.982 -> t1, Period- 96, Released at- 0, Priority- 7, WCET- 32, Deadline- 64
13:13:38.081 -> t2, Period- 192, Released at- 0, Priority- 6, WCET- 48, Deadline- 144
13:13:38.146 -> t3, Period- 288, Released at- 0, Priority- 5, WCET- 64, Deadline- 208
13:13:38.221 -> t1 - 0
13:13:38.268 -> t1 End - 32
13:13:38.268 -> t2 - 32
13:13:38.345 -> t2 End - 96
13:13:38.345 -> t1 - 96
13:13:38.392 -> t1 End - 144
13:13:38.392 -> t3 - 144
13:13:38.485 -> t3 End - 224
13:13:38.485 -> t1 - 224
13:13:38.532 -> t1 End - 272
13:13:38.532 -> t2 - 272
13:13:38.564 -> t2 End - 336
13:13:38.597 -> t1 - 336
13:13:38.597 -> Deadline missed - t1 - 352
13:13:38.630 -> t1 recreated - 384
13:13:38.666 -> t1 - 400
13:13:38.702 -> t1 End - 432
13:13:38.702 -> t3 - 432
13:13:38.781 -> t3 End - 512
13:13:38.781 -> t2 - 512
13:13:38.781 -> Deadline missed - t2 - 528
13:13:38.813 -> t2 recreated - 560
```

Ln 62, Col 1   UTF-8   Arduino Mega or Mega 2560 on COM3   2

Figure 9: EDF output for Task set 2

6 Future Work

There is a lot of scope for future work in the current implementation of our Earliest Deadline First Scheduling algorithm. Some of the work would include the consideration of Aperiodic and Sporadic jobs and their scheduling using Rate-monotonic, Deadline-monotonic and Earliest Deadline First Scheduling policies.

7 Conclusion

We successfully implemented the Earliest Deadline First scheduling policy on the ATMega2560 development board, tested it for multiple task sets given through the Arduino IDE and verified the output with the theoretical calculations on the Serial monitor of Arduino IDE. We also evaluated the EDF scheduling policy against the RM and DM scheduling policies and identified the difference between fixed and dynamic priority scheduling agorithms.

8 Project Deliverables

1. Project Report
2. README.md
3. scheduler.cpp
4. scheduler.h
5. Arduino_FreeRTOS.h
6. FreeRTOSConfig.h
7. project3.ino

References

[1] https://en.wikipedia.org/wiki/earliest_deadline_first_scheduling.

[2] Giorgio C. Buttazzo. Hard real time computing systems - predictable scheduling algorithms and applications, 1997.