technische universität
dortmund

Bachelorthesis

Evaluation of Multi-Mode Tasks under Rate-Monotonic and Earliest-Deadline-First Scheduling in Real-Time-Systems

Vincent Meyers

May 14, 2017

Betreuer:
Professor Dr. Jian-Jia Chen
Dr-Ing. Anas Toma

Fakultät für Informatik
Design Automation For Embedded Systems (Ls12)
Technische Universität Dortmund
http://ls12-www.cs.tu-dortmund.de

Contents

Chapter 1

Introduction and Background

1.1 Introduction

1.1.1 Multi-Mode Tasks in Real-Time Operating Systems

Today auto-mobiles are no longer only mechanical systems but additionally contain embedded electronic and software components. Improvements in functionalities, performance, comfort and safety have been provided significantly by electronics and software technologies [19]. Embedded electronics and networks can be used to control the physical processes in an auto-mobile to create an advanced automotive system [15]. These so called Cyber-Physical Systems(CPS), which are systems designed to have tight coordination between computational and physical resources [15], have advanced in recent years due to the ubiquity of available networks and sensors that can be accessed [11]. In 2005, more than 70 microprocessors embedding up to 500MB were used in automotive systems [19]. The functionality of those embedded systems may range from controlling the wipers and doors to controlling the engine and the fuel flow. In order to guarantee a correct behavior, the automotive embedded system has to react within a precise time constraint on events based on the environment and is therefore called a *real-time system*. The timing correctness is important as delayed reactions can result in faulty behavior in the automotive system and potentially lead to loss of safety for the driver. In order to achieve safety and reliability the hardware and the software of the system have to be considered.

The software of an automotive application may be modeled as a set of independent recurrent tasks with each task generating an infinite number of jobs which are being executed by the system. These tasks are processed by the central processing unit (CPU) of the embedded system. As a processor is a limited resource and can only execute one task at a particular time, the tasks have to share it in some way. Consequently, a set of rules, known as scheduling algorithm, are employed to determine which task is executed by the CPU. Usually a task's job has a fixed worst-case time needed for its execution. Now to control the engine of an auto-mobile, a task may release jobs depending on the engines

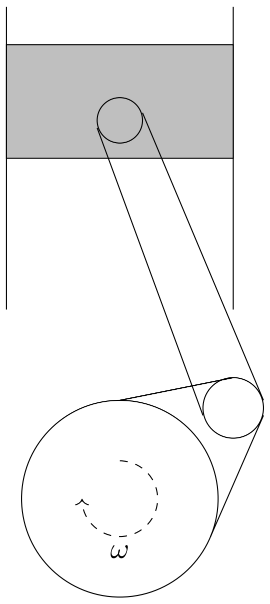


Figure 1.1: Simplified visualization of a crankshaft with rotation speed ω

speed. In order to meet the timing constraint given by the environment and prevent a potential system failure, the job has to react before the next job is generated. Therefore, the task might have to shed some of its functionality to meet its deadline which is its next release time. Classical solutions to this problem feature tasks with different modes of execution where in the case of a mode change all tasks perform a transition to their new parameters [20]. In some cases though, tasks may react on different inputs or differently on the same input and thus have to switch modes independently of another. In our example of the auto-mobile's engine the input for the corresponding tasks is the engine speed and the tasks' functionalities are part of the fuel injection system. Every time the crankshaft, which is visualized in figure 1.1, finishes a rotation the tasks have to execute their respective functions. This happens when the piston reaches its highest position. If the engine speeds up, the tasks eventually need to use another algorithm or function to achieve their goal to avoid deadline misses. Additionally it has been the case that the system is more stable at higher rotation speeds but needs additional functions to be executed at lower speeds to keep the engine stable. Consequently, these functions do not need to be executed at higher speed which can be exploited to reduce execution times [6]. A multi-mode task model suited for this case has been presented in *Rate-Adaptive Tasks: Model, Analysis, and Design Issues* and is referred to as the *Variable Rate Behavior* task model [6]. Huan and Chen provide techniques for analyzing the schedulability of such a model in their paper *Techniques for Schedulability Analysis in Mode Change Systems under Fixed-Priority Scheduling* [11]. Furthermore, they show the advantages of using a fixed-priority scheduling algorithm over a dynamic scheduling algorithm when scheduling multi-mode tasks by a simulation.

1.1.2 Thesis Contribution

The current results regarding the comparison of multi-mode tasks under dynamic and static scheduling were achieved by simulation. Experimental evaluation by implementing the model and using it on real hardware is also important as the results might differ from theory.

The goal of this thesis is to evaluate multi-mode tasks under rate-monotonic(RM) and earliest-deadline-first(EDF) scheduling in a real-time operating system. The main interest here lies in the comparison of the schedulers overheads and the influence that the transitioning between different modes has on specific evaluation metrics like the number of late tasks. Therefore, we implement the multi-mode task model along with the rate monotonic and earliest deadline first scheduling algorithm. The real time system chosen for this task is FreeRTOS. We use the Raspberry Pi, a device with a BCM2835 micro-controller which is used in a lot of embedded systems projects, as the environment for our experiments. In these experiments, we evaluate and compare the presented schedulers using different metrics. Furthermore, we apply two different test procedures, one with randomized values and the other with more realistic values as they can be found in real-world automotive software systems. The results of our evaluation can then be used for further research and analysis.

1.1.3 Thesis Organization

This thesis will start by giving background information about important subjects which we will use in this work. The background is divided into a detailed explanation about real-time tasks and operating systems(RTOS), schedulers and the real-time operating system FreeRTOS. Additionally, particular terms which will be used throughout the thesis are explained. Starting with a presentation of real-time tasks and real-time operating systems gives an insight to the importance of testing the multi-mode task model in practice. We then introduce the rate-monotonic and earliest deadline first scheduling procedures and their characteristics, which will be significant for the implementation. The last section of the background then gives an introduction to the real-time system FreeRTOS and its structure. The reader will then be able to understand the modifications we make to the system which are presented and explained in chapter 3. The chapter also covers the test design and task generation which will provide our test data. The results of the evaluation are then presented and discussed in chapter 4. The final chapter gives a summary of this thesis as well as giving directions for future work and pointing out unresolved problems.

1.2 Background

1.2.1 Real-Time Systems

Real-time systems are computing systems which have precise timing constraints in which they have to react based on events in the environment. Therefore correctness of such a system does not solely depend on the resulting output but also on the time said output gets produced. If the response of the system is too slow, it could render the response useless or even have a dangerous aftermath [5].

The following is a list of examples given by the book *Hard Real-Time Computing System* [5]:

- Chemical and nuclear plant control,
- control of complex production processes,
- railway switching systems,
- automotive applications,
- flight control systems,
- telecommunication systems,
- medical systems,
- military systems,
- space missions.

The term *real-time* does not mean that a system is able to react *very fast* to environmental changes but instead means that its response to external events meets a timing constraint proportional to the characteristics of the physical environment. Also the systems reaction has to happen during the evolution of the external events.

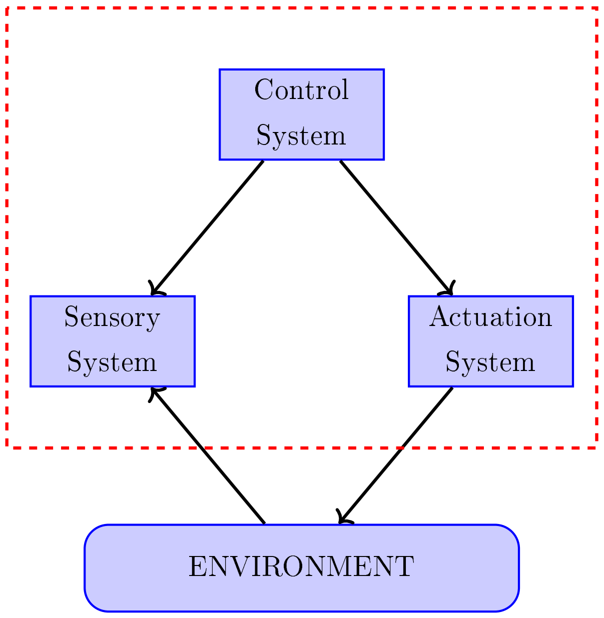


Figure 1.2: Block diagram of a real-time control system

Subsequently the environment is always an essential part of real-time systems [5]. A typical structure of a real-time system for the control of a physical system, e.g. an engine, is shown in figure 1.2.

As mentioned before, being fast is not as important for a real-time system than meeting specific timing constraints. In other words, the system has to be predictable [5].

1.2.2 Real-Time Tasks

A *task* is a set of instructions or computations that are sequentially executed by the central processing unit of a system [5]. In an environment with multiple concurrent tasks, tasks compete with one another for the control of the CPU. Concurrent tasks are tasks that overlap in time [5].

Consequently a task can either be waiting for the processor or executing on the processor. We call waiting tasks *ready*, executing tasks *running* and tasks that could potentially execute *active* [5]. Additionally there might the case of a task being delayed or suspended, in which we will refer to the task as *blocked*.
In cyber-physical systems, like an automotives engine control, tasks might need to meet real-time constraints to prevent system failure or misbehavior [17]. Such a task is called a *real-time task* and the real-time constraint the task has to meet is called a **deadline** [5]. If not meeting that constraint results in a catastrophe, the deadline is called *hard* [17]. In the following we distinguish between the following kinds of tasks [5]:

Hard: The real-time task is *hard* if not meeting the deadline can result in catastrophic consequences

Firm: The real-time task is *firm* if not meeting the deadline renders the produced results useless but does not do any damage to the system

Soft: The real-time task is *soft* if not meeting the deadline reduces the performance but is acceptable for the system

The three categories above are also referred to as the *criticality* of a task [5]. Our example of a task controlling the engine's fuel flow from the introductory chapter 1.1.1 falls under the category of *hard* tasks.

The next part of this section focuses on additional timing constraints which characterize a task and its execution.

1.2.1 Definition. A basic unit of execution or task in execution handled by the operating system is called a **job** J [17].

Jobs have the following timing parameters [5]:

Arrival time: The time at which the job becomes ready for execution is called **arrival time** a_j or **release time** r_j.

WCET: The **Worst-Case Execution Time(WCET)** C_j is the upper bound of the duration of a task execution.

Absolute deadline: The time at which the job should be completed is called **absolute deadline** d_j.

Relative deadline: The time length between the arrival time a_j and the absolute deadline d_j is called **relative deadline** D_j.

Start time: The time at which the job starts its execution is called **start time** s_j.

Finish time: The time at which the job finishes its execution is called **finish time** f_j.

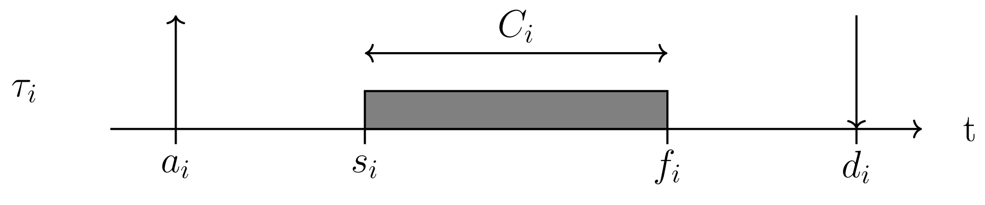


Figure 1.3: Timing parameters of a real-time task

Figure 1.3 shows some typical task parameters explained above. In addition to those parameters we introduce the following characteristics which are not essential for the operating system but for doing measurements in our evaluation [5]:

Response time: The time length the job needs to finish its execution after its arrival, which is $f_j - a_j$, is called **response time** R_j.

Lateness: The **lateness** L_i describes the time between the task's completion and its deadline $f_i - d_i$. If the task exceeds its deadline the lateness is positive.

Tardiness: The time a task stays active after its deadline is called *tardiness* $E_i = max(0, L_i)$

Laxity: The maximum delay acceptable for a task to complete within its deadline is the *laxity* $X_i = d_i - a_i - C_i$.

We have described the parameters for a task which executes a single time and will now introduce tasks which are executing recurrently.

Sporadic and Periodic Tasks

Next to the introduced constraints tasks can be characterized by their regularity of activation. Tasks can be seperated into periodic and aperiodic tasks [17]:

1.2.2 Definition. A task τ_i which requests the processor exactly every p time units is called **periodic task** with **period** T_i and phase ϕ_i which indicates the first release time of the task.
A periodic task τ_i can be denoted as a quadruple(triple) $\tau_i = (\phi_i, C_i, T_i, D_i)$.
If τ_i is omitted, it can be assumed that $\phi_i = 0$.

1.2.3 Definition. Tasks which execute at unpredictable times but have a minimum seperation between the times at which they execute are called **sporadic**. T_i is the minimum seperation time.
A sporadic task τ_i can be denoted as a triple $\tau_i = (C_i, T_i, D_i)$.

Tasks which are neither periodic nor sporadic are called **aperiodic**.
Figure 1.4 shows exemplary sequences of instances for a periodic and an aperiodic task. The phase ϕ_i is zero. Both periodic and aperiodic tasks generate an infinite sequence of identical jobs however the aperiodic jobs to not arrive regularly.

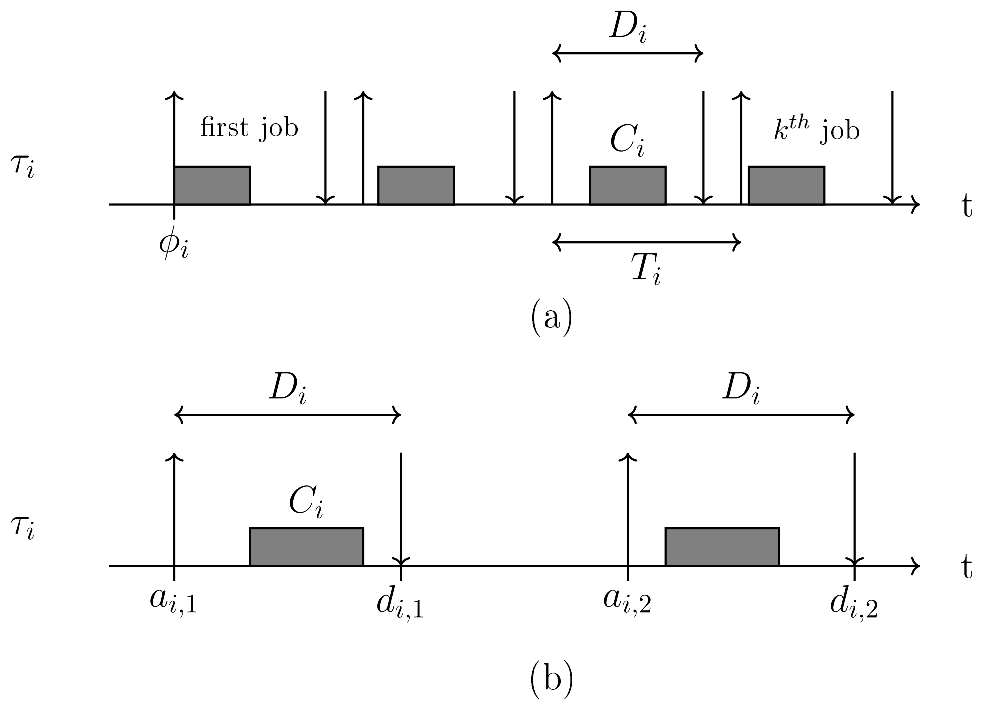


Figure 1.4: Task executions for a periodic (a) and aperiodic task (b)

Taking on the example of engine control in an automotive, such a system may be modelled as multiple independent recurrent tasks instead of implementing all functionality in a single task. We denote a set of tasks as task set Γ. Furthermore a task set with **implicit deadline** is a task set for which $D_i = T_i$ holds for every task τ_i [17]. For clarity, the Jobs of a task τ_i are from now on referred to as $J_{i,1}, J_{i,2}, ...$

The central processing unit of an embedded system is a limited resource which a task may periodically utilize to a certain degree. Using the timing constraints of a task we define the task utilization of a task τ_i and total utilization of a task set Γ as follows [16]:

Task utilization: $U_i := \frac{C_i}{T_i}$

Total utilization: $U(T) := \sum_{\tau_i \in \Gamma} U_i$

1.2.3 Schedulers

In a system where a single processing unit has to handle a set of tasks with overlapping execution times the CPU has to be assigned according to predefined criteria called *scheduling policy* [5]. The set of rules which realizes the scheduling policy is called *scheduling algorithm* and the procedure of actually selecting a task to transition from the *ready* to the *running* state is called *dispatching*. Following a scheduler is the part of the system that chooses which task is going to control the CPU next. The resulting order of tasks is called a schedule [5].

For the process of scheduling each task τ_i of task set Γ is assigned a priority π_i.

1.2.4 Definition. A **schedule** S is a mapping of jobs to the time at which they are granted access to the processor, such that each job is executed until completion.
A schedule can be defined as a function:
$\sigma : \mathbb{R} \to \mathbb{N}$
$\sigma(t) = j$ denotes the job $J_{i,j}$ of task τ_i executed at time t
$\sigma(t) = 0$ denotes that the system is idle at time t
The processor performs a context switch at time t if $\sigma(t)$ changes its value at some time t.

A schedule is *feasible* if all jobs which are scheduled meet their specified constraints [5]. Accordingly, a task set is said to be *schedulable* if an algorithm which produces a feasible schedule for that task set exists [5]. Given a task set Γ with total utilization U we define the *upper bound* $U_{ub}(\Gamma, A)$ of the set under scheduling algorithm A as the maximum value of processor utilization for which Γ set is schedulable [5]. This means that increasing U by increasing the tasks' computation times or decreasing periods results in Γ becoming infeasible. Figure 1.5 shows a schedule with $U_{ub} \approx 0.83$ for two tasks τ_1 and τ_2. τ_2 is the higher priority task in this example. Increasing any execution time will result in an infeasible schedule since the first job of τ_1 will miss its deadline.

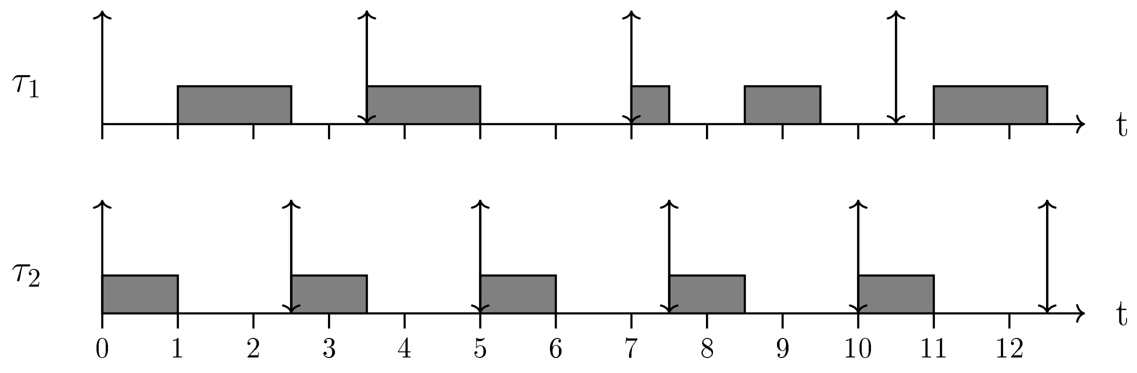


Figure 1.5: A task set with $U_{ub} = \frac{29}{35} \approx 0.83$

Additionally we specify the minimum of all U_{ub} of Γ as the *least upper bound* U_{lub} [5]:

$$U_{lub} = \min_{\Gamma} U_{ub}(\Gamma, A)$$

Consequently every task set with $U < U_{lub}(A)$ is feasible under scheduling algorithm A [5].

If a task is executing while another task with a higher priority becomes ready it might be important to let the higher priority task access the CPU and let the lower priority task finish afterwards. This process is called *preemption*. More reasons for allowing preemption are allowing tasks to perform exception handling in a timely fashion and being able to schedule tasks with a higher processor utilization [5]. Therefore we distinguish scheduling between **preemptive** and **non-preemptive** scheduling as follows [5]:

Non-preemptive: For **non-preemptive** scheduling there can be only one time interval with $\sigma(t) = j$ for every J_j in the corresponding period p_j, where t is covered by the interval.

Preemptive: For **preemptive** scheduling there can be more than one time interval with $\sigma(t) = j$ for every J_j in the corresponding period p_j.

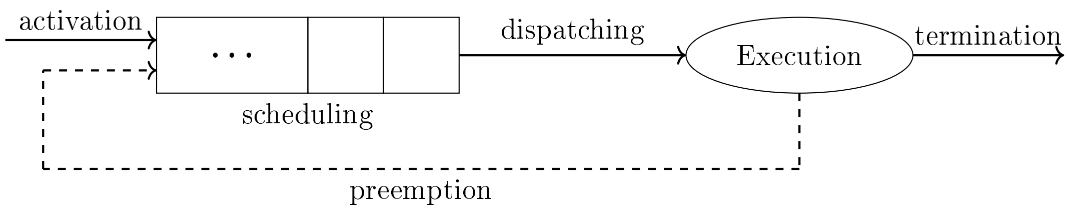


Figure 1.6: A queue of tasks that is being scheduled and dispatched.

When classifying a scheduling algorithm we can do so by considering further characteristics of the algorithm like the parameters its decisions are based on. If the parameters are fixed we call the scheduling *static* and *dynamic* if they can change during the time that our system is active [5]. Additionally the time at which decisions are made is an identifying property. Thus, we distinguish between *off-line* scheduling, which is assigning the task priorities of the entire task set before the tasks are activated, and *online* scheduling, in which the algorithm makes decisions at runtime whenever a task enters the ready state or a task terminates [5]. Lastly scheduling algorithms differ in the resulting schedule that is aimed for and accepted. A *heuristic* algorithm uses a heuristic function in order to find an optimal schedule. Consequently, it is not guaranteed to find one but will always tend towards the optimal schedule. On the contrary, an *optimal* scheduling algorithm minimizes a given cost function, or if not defined, achieves a feasible schedule. If the algorithm always finds an existing feasible schedule it is called *optimal* [5].

An exemplary scheduling algorithm for static scheduling is the Rate-Monotonic Scheduling Algorithm. For dynamic scheduling an exemplary scheduling algorithm is the Earliest-Deadline-First Scheduling Algorithm. These scheduling algorithms are explained in the following sections. When explaining the schedulers we refer to periodic scheduling.

Rate Monotonic Scheduling

One of the scheduling algorithms implemented in this work is the Rate Monotonic (RM) scheduling algorithm. This algorithm assigns priorities based on the periods of tasks. The shorter the period (or the higher the request rate), the higher is the priority assigned to the task. Since periods are fixed parameters which are set before task execution, the rate monotonic scheduling algorithm is classified as a static algorithm [5].

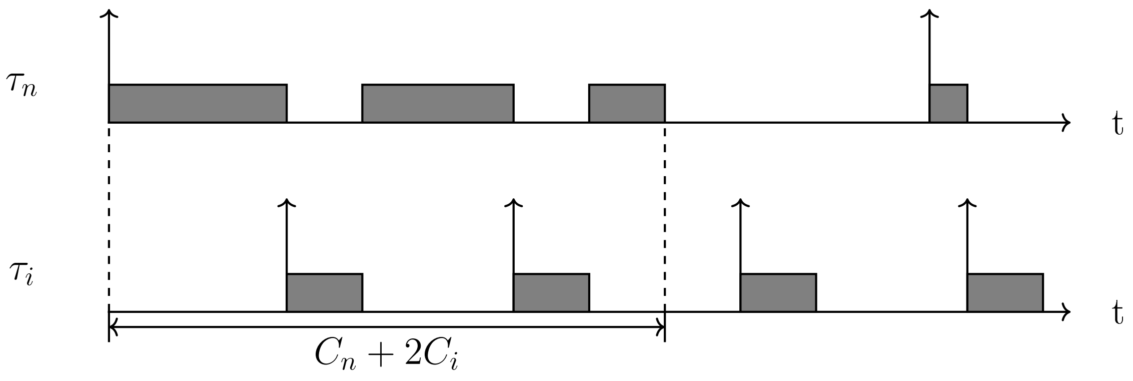


Figure 1.7: A task set scheduled following the RM policy

Figure 1.7 shows a task set scheduled using the rate monotonic algorithm. The response time of τ_n is delayed by the interference of the higher priority task τ_i.

Liu and Layland proved that RM is an optimal fixed-priority algorithm in 1973. In addition they showed that any task that can be scheduled by a static scheduling algorithm can be scheduled by RM [16]. Moreover they derived a *least upper bound* for the processor utilization for n periodic tasks under RM scheduling, which is [16]:

$$U_{lub} = \ln 2 \simeq 0.69$$

Earliest Deadline First Scheduling

The second scheduling algorithm introduced in this section is the Earliest Deadline First (EDF) algorithm, which is a dynamic algorithm that chooses task priorities based on their absolute deadlines. More precisely, the closer a task is to its absolute deadline the higher its priority will be [5]. The algorithm is then dynamic as the absolute deadline of a periodic task is not a static parameter. It is computed as follows:

$$d_{i,j} = \phi_i + (j-1)T_i + D_i$$

The *least upper bound* for EDF is $U_{lub} = 1$ [5]. Specifically, the following theorem holds[16]:

1.2.5 Theorem. *For a given set of n tasks, the EDF scheduling algorithm is feasible if and only if*

$$\sum_{i=1}^{n} = \frac{C_i}{T_i} \leq 1$$

Figure 1.8 shows a schedule produced by RM and EDF on the same task set. It highlights the advantage of EDF being able to schedule task sets with a higher processor utilization.

τ_1 will miss its first deadline's deadline under RM scheduling while the set is schedulable under EDF. However, in praxis using dynamic scheduling like EDF comes at the cost of having to sacrifice computation time on dynamically calculating the schedule each time a task readies. We will show this in the results of our experiments in chapter 4.

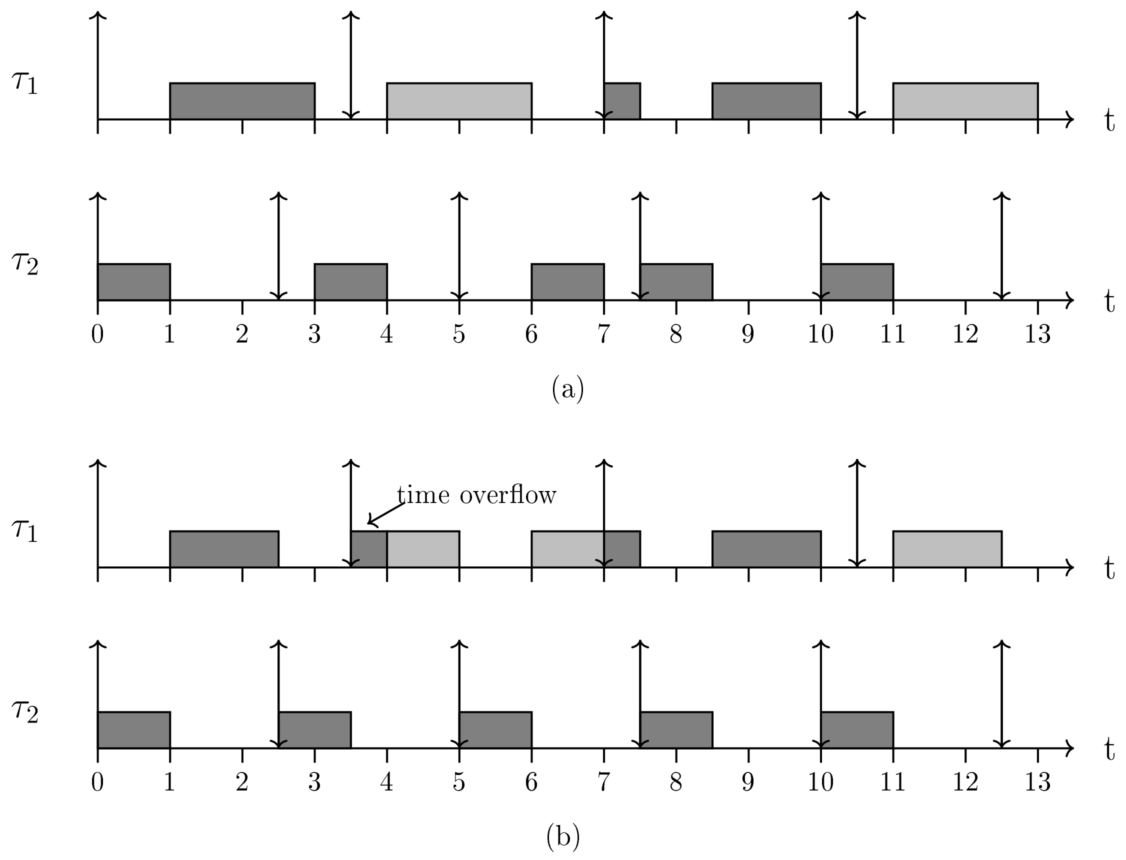


Figure 1.8: A periodic task set scheduled following the EDF (a) and RM (b) policy

Performance metrics

For our evaluation we need specific criteria to measure the performance of the presented schedulers. Therefore, we introduce the following cost functions [5]:
Average response time:

$$t_r = \frac{1}{n}\sum_{i=1}^{n}(f_i - a_i)$$

Total completion time:

$$t_c = \max_i(f_i) - \min_i(a_i)$$

Weightes sum of completion times:

$$t_w = \sum_{i=1}^{n} w_i f_i$$

Maximum lateness:

$$L_{max} = \max_i (f_i - d_i)$$

Maximum number of late tasks:

$$N_{late} = \sum_{i=1}^{n} miss(f_i), \text{ with } miss(f_i) = \begin{cases} 0, & \text{if } f_i \leq d_i \\ 1, & otherwise \end{cases}$$

1.2.4 System Overhead

The information presented in this section is provided by Giorgio C. Buttazzo in his book *Hard Real-Time Computing Systems* [5]. The time it takes for the processor to handle all mechanisms which are not tied to executing jobs is called the overhead of the operating system. Exemplary operations which cause overhead could be context switches, communication with peripheral devices or over channels or interrupt requests.

Switching the context is an important factor in operating system overheads. It is independent from the implemented scheduling algorithm and the application. Another cause for overhead is the system tick interrupt which happens periodically. Let Q be the period of the system tick and σ be the worst-case execution time in which the interrupt timer executes. The resulting utilization U_t can be then computed as:

$$U_t = \frac{\sigma}{Q}$$

1.2.5 FreeRTOS

FreeRTOS is a real-time operating system, short RTOS, an operating system that supports the construction of real-time systems. It is a widely used and relatively small application consisting of up to 6 C files and supports various architectures.

Features of the system which make it a reasonable choice for this work are the preemptive scheduler, which will be explained in section 3 of this chapter, the support for real time tasks and the portable source code structure which allows FreeRTOS to run on a Raspberry Pi B+. Additionally there are no software restrictions on the number of real time tasks or the number of priorities that can be used and the assignment of priorities.

The following sections will provide detailed information about the properties of FreeRTOS that are important for this work. If not stated otherwise, the source for all information in this section is taken from *http://freertos.org* [1].

Structure

In this section the structure of FreeRTOS is explained. The explanation will focus on information which is important for the implementation of multi-mode tasks in FreeRTOS

and the implementation of the schedulers.
Figure 1.9 and 1.10 give an overview of a FreeRTOS projects file structure. In our case the demo application folder also contains the drivers needed to control particular peripherals of our hardware. FreeRTOS can be customised by modifying the configuration file *FreeRTOSConfig.h*, i.e. turning preemption on or off and setting the frequency of the system tick. This file must be present in the pre-processor include path of every application.

```
FreeRTOS
    ¦
    + ---- Demo      Contains the demo application projects
    ¦
    + ---- Source    Contains the real time kernel source code
```

Figure 1.9: The basic folder structure of FreeRTOS

```
FreeRTOS
    ¦
    + -- Source              The core FreeRTOS kernel files
             ¦
             + -- include    The core FreeRTOS kernel header files
             ¦
             + -- Portable   Processor specific code
                      ¦
                      + - Compiler x    Ports supported for compiler x
                      ¦
                      + - Compiler y    Ports supported for compiler y
                      ¦
                      + - MemMang       Sample heap implementations
```

Figure 1.10: The structure of the FreeRTOS source folder including the portable folder

The following are variables and functions naming conventions in FreeRTOS which will be complied in the implementation:
Variables (combinations are posssible):

- c: char
- s: short
- l: long
- x: portBASE_TYPE and any others
- u: unsigned
- p: pointer

Function prefix:

- prv: private function
- returning data type
- v: void

Tasks

This section will give a brief overview about how tasks are handled in FreeRTOS. The properties of tasks, their states and important related functions are explained. Only functions and properties that are significant for this work are covered.

Tasks in FreeRTOS execute within their own context with no dependency on other tasks or the scheduler. Furthermore the scheduler activity is unknown to tasks and therefore they are not responsible for the behavior of processor context switches. This is the sole responsibility of the RTOS scheduler. Upon creation each task is assigned a task control block, short TCB, which contains the stack pointer, two list items and the tasks priority. Tasks can have priorities from 0, lowest, to *configMAX_PRIORITIES - 1*, highest, where configMAX_PRIORITIES is configured in *FreeRTOSConfig.h*.
The creation of a task is possible with a call of the function:

```
xTaskCreate(TaskFunction_t pvTaskCode,
            const char * pcName,
            unsigned short usStackDepth,
            void * pvParameters,
            UBaseType_t uxPriority,
            TaskHandle_t * xTaskHandle)
```

A task can be deleted by calling the function:

```
void vTaskDelete( TaskHandle_t xTask )
```

If *NULL* is passed to *vTaskDelete()* the calling task will delete itself. Memory allocated by the task code needs to be freed manually. Additionally a task can delay itself using either

```
void vTaskDelay( const TickType_t xTicksToDelay )
```

or

```
void vTaskDelayUntil(TickType_t *pxPreviousWakeTime,
                     const TickType_t xTimeIncrement )
```

vTaskDelayUntil() allows a constant execution frequency while *vTaskDelay()* blocks the task for a a given number of ticks relative to the time at which the function is called.

A task in FreeRTOS can be in four different states:

Running: The task is currently executing.

Ready: The task is ready to execute but preempted by a higher priority task. Only tasks in the ready state can be selected to enter the running state.

Blocked: The task is waiting for an event and delays itself. After a timeout the task will be unblocked.

Suspended: The task is blocked but does not unblock after a timeout. Instead the task enters or exits the suspended state only when explicitly commanded to do so.

The possible state transitions are displayed in Figure 1.11. The task's states are realized

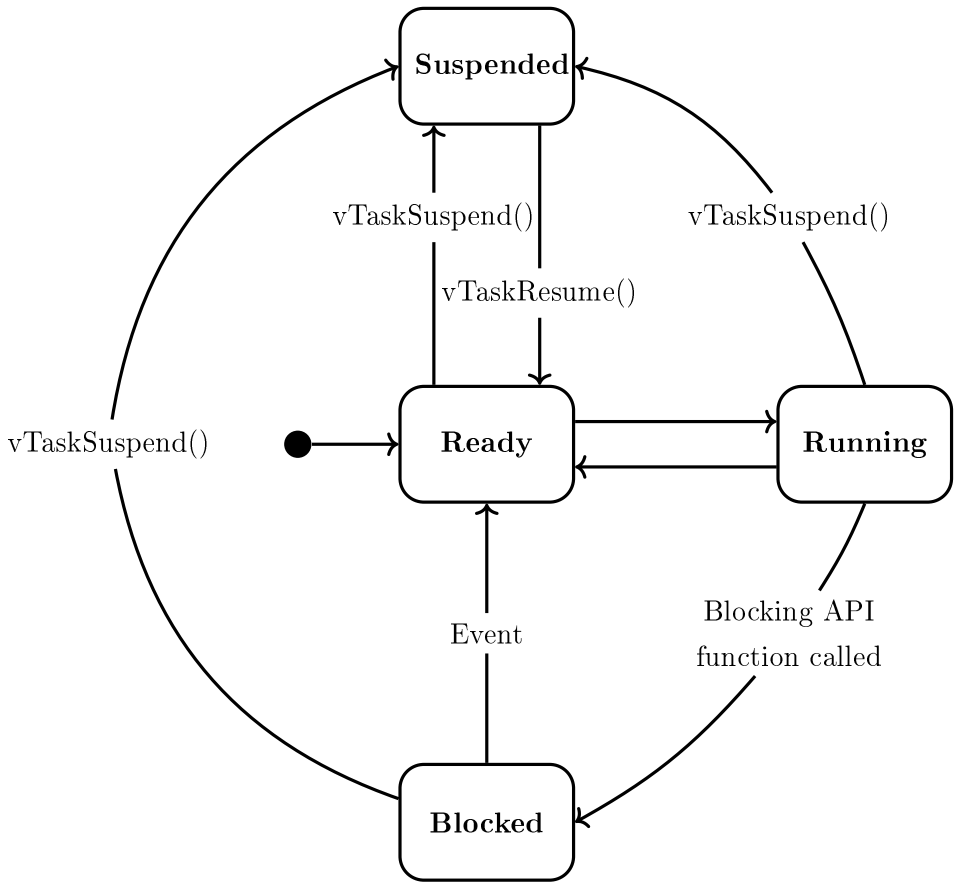


Figure 1.11: Valid state transitions for tasks in FreeRTOS

by doubly linked lists, a list for which each element in the list knows the previous and next element in the list. Whenever we mention that a task is contained or inserted in a list structure we actually mean that the pointer to that task's TCB is being contained or inserted to that list. Tasks which become ready to be executed or the task that is currently running are contained in an array of doubly linked lists *pxReadyTasksLists[]* of size *configMAX_PRIORITIES* according to their priority. Whenever a task is inserted to its ready list its priority is checked against the currently highest priority *uxTopReadyPriority* in order to keep track of the highest priority and speeding up context switches. Suspended tasks are inserted to *xSuspendedTaskList* and delayed(blocked) tasks to *pxDelayedTaskList*. As mentioned before the running task is contained within one of the lists in the array *pxReadyTasksLists[configMAX_PRIORITIES]*. It is also pointed to by the pointer *tskTCB * volatile pxCurrentTCB*. This is how FreeRTOS keeps track of the running task.

Scheduler

The scheduler of FreeRTOS is responsible for deciding which tasks executes at a specific time. It is implemented in the routine which is executed for each system tick interrupt. The implementation might vary with the corresponding port.

The following is an exemplary routine for the tick interrupt as it is implemented in the port for the Raspberry pi:

```
void vTickISR(unsigned int nIRQ, void *pParam )
{
        vTaskIncrementTick();

        #if configUSE_PREEMPTION
                vTaskSwitchContext();
        #endif
}
```

Time is measured in system ticks which can be configured to the desired frequency in *FreeRTOSConfig.h*. The highest value which is possible for the tick ratio depends on the hardware and the port. In *vTaskIncrementTick()* blocked tasks are be unblocked if the required event happened. After that the actual scheduling process takes place in the function *vTaskSwitchContext()*. Therefore the system loops through *pxReadyTasksList* starting from *uxTopReadyPriority* down to the lowest priority and stops when it finds a non-empty list. Consequently the tasks contained in that list have to be the tasks with the highest priority. The value of *uxTopReadyPriority* is updated accordingly and *pxCurrentTCB* is set to the next item in the list. Subsequently if the list *pxReady-TasksLists[configMAX_PRIORITIES]* contains more than one task *pxCurrentTCB* is alternating between those tasks until one or both of them get blocked or suspended. Therefore the processor is shared between tasks of the same priority.

Chapter 2

Multi-Mode Task Model

In this chapter we introduce the task model which is to be implemented. An informal explanation, the motivation for the model and a formal definition is given.

Multi-mode tasks are self-adjusting activities which are independent of other tasks. A multi-mode task can execute in several modes which are specified by different execution times, periods and deadlines. Such a task may change its mode dependent on an external interrupt which may be used to reduce the execution time and lower the total utilization.

2.1 Motivation

In modern automotive systems computers are used to control and improve the performance of various parts of the automotive system. These embedded systems are in continuous interaction with various parts of the auto-mobile, for example the doors, the wipers, the lights and most importantly the engine [19]. As a car is a safety-critical system which needs to ensure functional and timing correctness, the engine control needs to be executing flawlessly. Faulty behaviour can possibly result in a fatality. Now to react accordingly, the embedded systems tasks which are interacting with the engine have the following structure:

```
set timer to interrupt periodically with period T;

at each timer interrupt do:
        receive input over analog to digital conversion;
        use input to compute control output;
        send output over digital to analog conversion;
end
```

An angular task is a task which is is linked to the rotation of specific devices like the crankshaft, gears or wheels. Let τ be a task linked to the automotive's crankshaft and the related engine speed ω. Such a task could be responsible for calculating the time at which the spark signal has to be fired, adjusting the fuel flow as well as minimizing fuel consumption and emissions [11]. Generally such a sporadic task τ is characterized by its fixed worst-case execution time, period/minimum inter-arrival time and relative deadline. Due to its dependency on the source of rotation, here described by the angle of crankshaft θ, speed of the crankshaft ω and acceleration of the crankshaft α , the task's period is inversely proportional to ω. The period can be computed as follows [4]:

$$T_i(\omega) = \frac{\theta_i}{\omega}$$

With increasing rotation speed ω the time available for the task to execute all of its functions might not be long enough and the task will eventually miss its deadline. In a hard real-time system this could potentially lead to catastrophic consequences [5]. Figure 2.1 shows an example for such an occurrence also called task overloading.

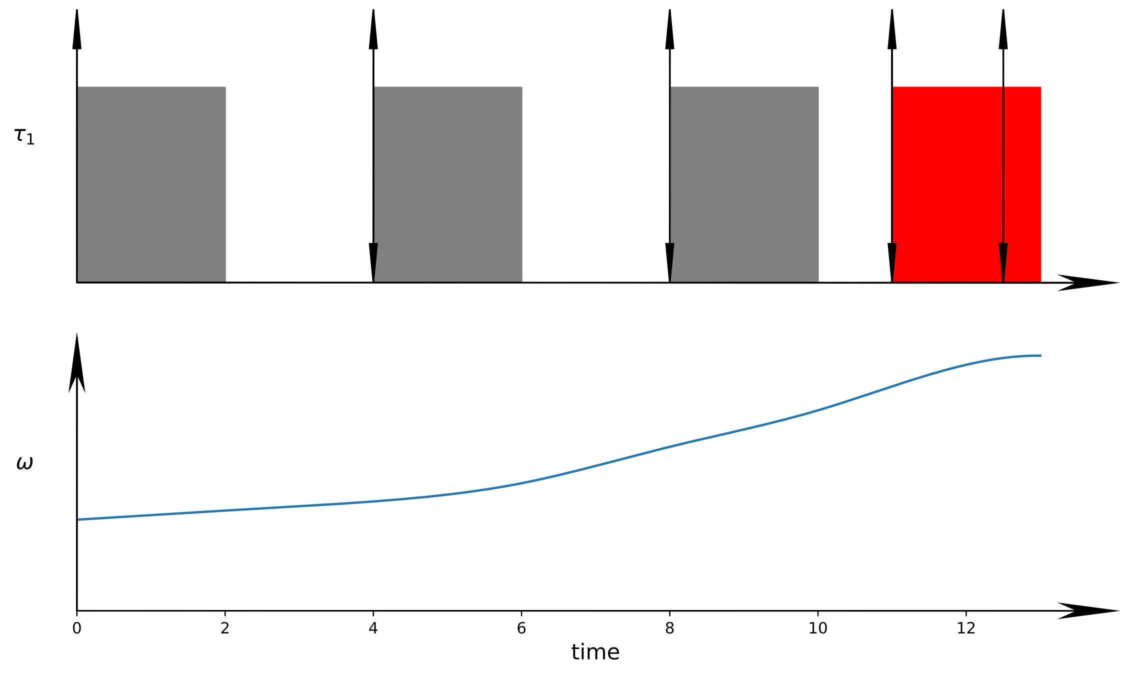


Figure 2.1: The task is overloading during high speed rotations.

To guarantee that deadlines are met in the system described above, the characteristics of the tasks need to be able to adapt to the engine speed. This can be accomplished by shedding functions of the task which are not critical for the correct performance of the automotive, e.g. fuel consumption and emission control. Doing so can reduce the worst-case execution time and the deadline will be met. Consequential a task needs different modes it can run on, depending on the engine speed ω. This leads us to the definition of multi-mode tasks.

2.2 Definition

Multi-mode tasks are denoted by sets of triples:

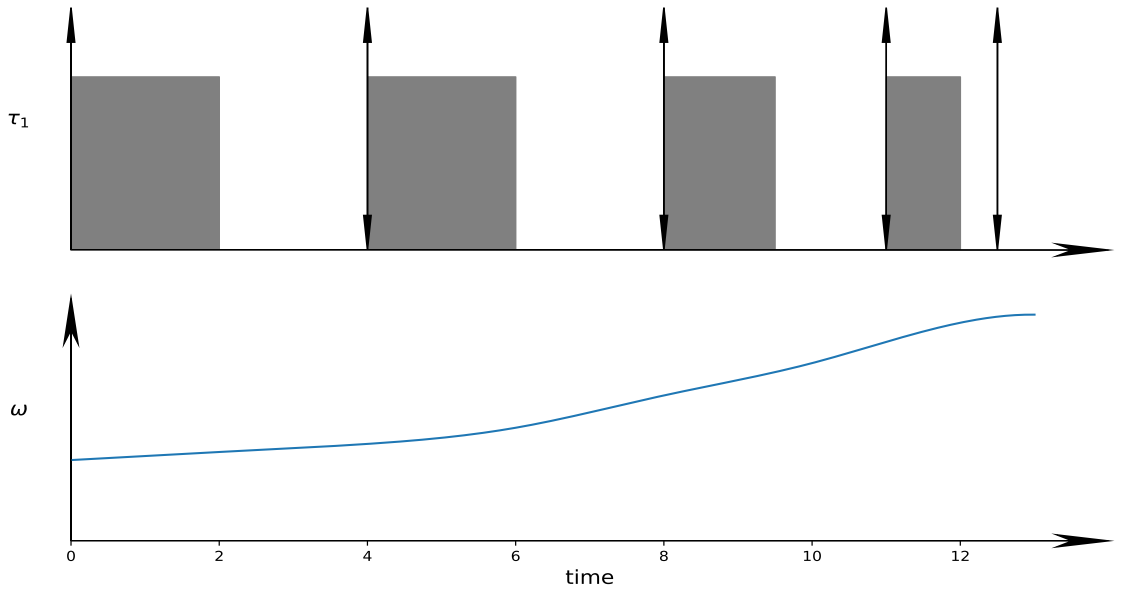
$$\begin{aligned}\tau_i = \{\tau_i^1 &= (C_i^1, T_i^1, D_i^1),\\ \tau_i^2 &= (C_i^2, T_i^2, D_i^2),\\ &\ldots,\\ \tau_i^{M_i} &= (C_i^{M_i}, T_i^{M_i}, D_i^{M_i})\}\end{aligned}$$

M_i is the highest mode of task τ and m is a mode with $1 \le m \le M_i$.
C_i^m is the WCET of task τ_i under mode m.
T_i^m is the minimum inter-arrival time of task τ_i under mode m.
D_i^m is the relative deadline of task τ_i under mode m. It is important to note that if a task τ_i^m is released at time t the next release time of that task will be equal or bigger than $t + T_i^m$ even if a mode change happens while the task is delayed/blocked.

Example. This is an example of a multi-mode task with four types of execution modes dependent on the rotation speed rpm.

rotation (rpm)	functions to be executed
[0,2000]	$f1(); f2(); f3(); f4();$
[2000,4000]	$f1(); f2(); f3();$
[4000,6000]	$f1(); f2();$
[6000,8000]	$f1();$

The chronograph for Figure 2.1 subsequently looks as follows:



Chapter 3

Design and Implementation

This chapter covers the implementation of multi-mode tasks and the rate-monotonic as well as the earliest-deadline-first scheduler in FreeRTOS. We will start with the multi-mode tasks as it is the fundamental system model used by the schedulers.

3.1 Multi-Mode Tasks

In this section we will give a step by step explanation on how the multi-mode task model was implemented in the existing FreeRTOS port.

3.1.1 Real-time constraints

Multi-mode tasks are sporadic tasks. Thus in order to implement the multi-mode task model in FreeRTOS a periodic or sporadic task model is needed. Before any of the systems mechanisms can be exploited for that cause, the task control block structure of FreeRTOS needs to be expanded by typical fields used in a periodic real-time system [5]. In addition to the current fields of the TCB there will be the following added to the end of the TCB:

unsigned int uxPeriod: the period of task τ

unsigned int uxWCET: the worst-case execution time of task τ

unsigned int uxDeadline: the relative deadline of task τ

unsigned int uxPreviousWakeTime: the previous wake time of task τ

The absolute deadline D of task τ can then be computed by

$$D = uxDeadline + uxPreviousWakeTime$$

With these attributes added to tasks they also require initialization upon the tasks creation. This can be achieved by adding them as parameters to the call of *xTaskGenericCreate()* and the corresponding function *xTaskCreate()*. As real-world applications typically consist of hard tasks and non-real-time tasks we want to keep the option of having non-real-time tasks which will be assigned a fixed priority depending on the scheduling algorithm. Therefore we implement an option to create either kind of task by passing different parameters. Which parameters need to be passed is explained at the end of this chapter in section 3.4. For prioritizing hard tasks over non-real-time tasks each scheduler will have its own mechanism. We define the resulting function *xTaskGenericCreate()* as follows:

```
xTaskGenericCreate(pdTASK_CODE pxTaskCode,
                   const signed char * const pcName,
                   unsigned short usStackDepth,
                   void *pvParameters,
                   unsigned portBASE_TYPE uxPriority,
                   xTaskHandle *pxCreatedTask,
                   portSTACK_TYPE *puxStackBuffer,
                   const xMemoryRegion * const xRegions,
                   portTickType period,
                   portTickType wcet,
                   portTickType deadline)
```

The added parameters *period*, *wcet* and *deadline* are unsigned as only values which are greater than 0 are realistic. We also add those parameters to the function *prvInitialiseTCB-Variables()* which is located in task.c. This function is responsible for the initialization of the fields in the TCB. The idle task is initialized with 0 for *wcet*, *period* and *port-MAX_DELAY* for the deadline. Even though these are all the necessary changes for the TCB the task execution is not yet periodic.

In general, tasks in FreeRTOS are not periodic even though the system supports mechanism to implement periodicity. These mechanisms are software timers and the task control function *vTaskDelayUntil()*. Therefore a periodic task system can be realized by using either methods. A software timer can be set up to execute a function at a specific point in the future and allows periodic execution of callback functions which are fired when a timer expires. The period of a timer can be changed natively and a timer may be configured to execute only one time or recurrently. The drawback of timers is that they all share the same TCB of the timer service task, so they also share the same priority and consume additional FreeRTOS heap to store the timer's state. The advantage of software timers however is that they don't add any overhead to the system tick as the timer task does not check expired timers during that time [1].

vTaskDelayUntil() is a task control function located in the file task.c which takes a pointer to a variable of type *TickType_t* and a constant of the same type as parameters. The function is defined as:

```
vTaskDelayUntil(TickType_t *pxPreviousWakeTime,
                const TickType_t xTimeIncrement)
```

A task that calls *vTaskDelayUntil()* will be placed on the sorted blocked list for an *absolute* time [1]. The value which is assigned to the tasks generic list item is *xTimeIncrement*, so tasks get inserted to the list in the correct order. As a result the system only has to check the current tick against the time of the first item in the blocked list and unblock the corresponding task if necessary. Using *vTaskDelayUntil()* will one one hand consume more RAM in comparison to software timers, as for each task a TCB has to be allocated. On the other hand, having a TCB for each periodic activity of our application allows more control, a more structured process when it comes to implementing the modes and the function is not called from the system tick's context either, so there is no overhead added to the tick interrupt at that point. As the system checks for blocked tasks each system tick, time will be consumed for unblocking a task and putting it on the ready queue. In our approach we are using the function *vTaskDelayUntil()* to realize periodicity in FreeRTOS as software timers are not part of the core system and might not be usable in every project. Furthermore using the task control functions of FreeRTOS ensures control of the system and minimizes the changes which need to be made. It is worth mentioning that utilizing the function *vTaskDelay()* is not sufficient for the wanted behaviour as it does not remember the last wake-time and will unblock the task at a *relative* time from the time it is called instead of an *absolute* time in the future. This is shown in the comparison of both functions in figure 3.1. The figure shows that vTaskDelay() is not suited for a periodic task execution. To keep downward compatibility we add a function *vTaskPeriodicDelay()* which has the same functionality as *vTaskDelayUntil()* but also updates the last wake time of the calling task. We will refer to *vTaskPeriodicDelay()* instead of *vTaskDelayUntil()* from now on.

Each task is created with an associated task function which will be carried out each time the task gets executed by the CPU. That task function is passed as the parameter *pdTASK_CODE pxTaskCode*. We can use this knowledge to manipulate the function in a way that will make the related task periodic. To achieve this we will implement a new function

```
void vMakePeriodic(void* pParameters)
```

which is getting passed a struct as a parameter. The reason for creating an additional struct for each task is that the function *pxPortInitialiseStack()* expects parameters of type (void*) for the task function parameters. A different option would have been to adapt

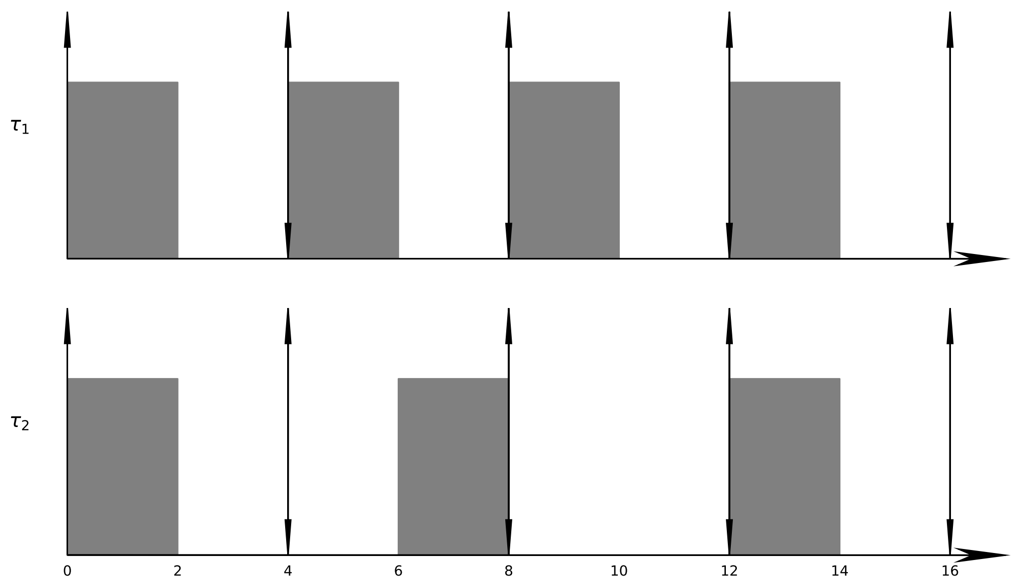


Figure 3.1: A comparison of the functions *vTaskDelay()* (τ_2) and *vTaskDelayUntil()* (τ_2). Both task are using the same worst-case execution time and implicit deadlines.

pxPortInitialiseStack() to our needs but this would increase the work required to make an existing application work in our system. We define the struct *xPeriodicParameters* with the following fields:

pdTASK_ CODE xTaskCode: the task code to execute

*void *xParameters:* the parameters initially passed to *xTaskCode*

The wake time is stored in the variable *xLastWakeTime*. In our approach we are omitting phases so each task starts at tick 0 and the first wake time of the task is set to 0. The system may be expanded to use phases by adding field *uxPhase* to the TCB, initialise it accordingly and set the *xFrequency* in *vMakePeriodic* to said phase. After that a call of vTaskPeriodicDelay(&xLastWakeTime, xFrequency) can be done to actually delay the task for its phase. Going on with the implementation of periodicity the original *xTaskCode* is nested in an endless while-loop which looks like this:

```
while(1)
{
        xFrequency = *(pxCurrentTCB>uxPeriod);
        f(periodicParameters>xParameters);
        vTaskPeriodicDelay( &xLastWakeTime, xFrequency );
}
```

f is pointing to the task function. Considering changes in the period we update *xFrequency* every time the task wakes.

Following these instructions we obtain a periodic system in FreeRTOS which allows scheduling periodic, aperiodic, real-time and non-real-time tasks. The next section will attend to expanding our periodic system to be able use multi-mode tasks.

3.1.2 Modes

To implement different modes of a task we have to turn their characteristics into a readable form for the computer. Multi-mode tasks are, as described in chapter 2, denoted as a set of triplets. The set's length is not variable and therefore a suitable data structure is an array. We will define an array for each real-time constraint of the task. The TCB fields are then changed to:

*portTickType * uxWCETs:* the worst-execution times of the task

*portTickType * uxPeriods:* the periods of the task

*portTickType * uxDeadlines:* the deadlines of the task

Additionally the task has to know the fixed number of modes and for which values a mode change has to take place. Thus we add the following two parameters:

*unsigned int * uxModeBreaks:* denotes the range for each mode

*unsigned int * uxNumOfModes:* the number of modes

uxModeBreaks contains the maximum value for each mode. If compared to an external input it allows to choose the adequate mode for that situation. The first mode at array position 0 ranges from 0 to *uxNumOfModes*[0]. We add the parameters to the corresponding functions as we did for the periodicity of the system and change the existing ones accordingly:

```
xTaskGenericCreate(pdTASK_CODE pxTaskCode,
                   const signed char * const pcName,
                   unsigned short usStackDepth,
                   void *pvParameters,
                   unsigned portBASE_TYPE uxPriority,
                   xTaskHandle *pxCreatedTask,
                   portSTACK_TYPE *puxStackBuffer,
                   const xMemoryRegion * const xRegions,
                   portTickType * periods,
                   portTickType * wcets,
                   portTickType * deadlines,
                   unsigned int * uxModeBreaks,
                   unsigned int * uxNumOfModes)
```

For task creation the application has to specify the arrays and pass them to the function as a pointer. The initialization of the TCB parameters is done in the same manner as explained in the previous section.

Having implemented the model's parameters, mode changes need to be brought into the system now. We declare a function *vUpdateMode()* which chooses the appropriate mode for the given input. This function is not responsible for computing the input, it solely chooses a mode depending on the tasks *uxModeBreaks*. The input value, which is determined by an external interrupt in most cases, is stored in the global variable *volatile unsigned int externalInput*. The variable is declared volatile as its value might change at any time. If an application wants to make use of multi-mode tasks it therefore has to implement some mechanic to change that variables value. Otherwise *externalInput* is always 0 which results in the first mode being chosen. According to our definition of multi-mode tasks in chapter 2, tasks do not change their mode during runtime and a tasks next release time is that of the last modes period even if the mode changes while the task is blocked. Therefore it is sufficient if we update a tasks mode right before its next wake-time. This can be done by calling *vUpdateMode()* at the start of the function prvAddTaskToReadyQueue(). It follows that at the time the task unblocks and gets executed by the CPU it will be in its correct mode for the time that it was released. Subsequently choosing the correct mode adds to the systems overhead .As the time needed for updating the mode is dependent on each individual task it should be measured and taken into consideration when designing an application for this system.

3.2 Rate-Monotonic Scheduler

In this section we will explain how we implemented the rate-monotonic scheduler in our system. We will start by the basic idea followed by further improvements done to enhance the scheduling process.

In order to implement a rate-monotonic scheduling policy as explained in section 1.2.3 we have to assign priorities before starting our scheduler and executing tasks. For that we implement a new function *vAssignPriorities()* which will be called in *vTaskStartScheduler()* right after the creation of the idle task so all tasks to be scheduled are present for the scheduling process. We reserve priority 1 and *pxReadyTasksLists[1]* for the rate-monotonic scheduling algorithm. All real-time tasks are temporarily inserted with priority 1 and thus the list of ready tasks *pxReadyTasksLists[1]* contains all tasks which require scheduling. The non-real-time tasks are excluded from the scheduling process and are assigned a static priority depending on the applications choice. Preferably the application uses priorities which have been created solely for those tasks and which are below the priorities reserved for real-time tasks.
To explain our goals for the rate-monotonic scheduling we first present a simple solution for scheduling multi-mode tasks in a rate-monotonic way. This can be implemented by setting the number of priorities to the highest period which can be assigned in the application. Then each tasks priority can be computed by

$$\pi_i = configMAX\_PRIORITIES - T_i$$

This guarantees that each task is assigned the correct priority, but it might also result in a huge number of unused priorities, e.g. scheduling one task with period 2000 will need 2000 priorities. This will create additional overhead for finding the highest priority task and require more available RAM as a list has to be initialized for each priority. The overhead is a result of FreeRTOS iterating over the list of priorities until it finds the highest priority task. The time needed for this is in the worst-case $O(n)$ where n is the number of priorities. Nonetheless, this approach gives us an impression about our requirements for the rate-monotonic scheduling, which are:

- static priority assignment
- exactly one priority for each period used
- each mode needs to be covered
- no unused priorities exist

Preferably we would also like to do context switches in constant instead of linear time to reduce the system overhead.

Based on these requirements we now begin by designing the function *vAssignPriorities()*. As each mode's period of each task needs to be considered by the scheduling we link them in the following struct:

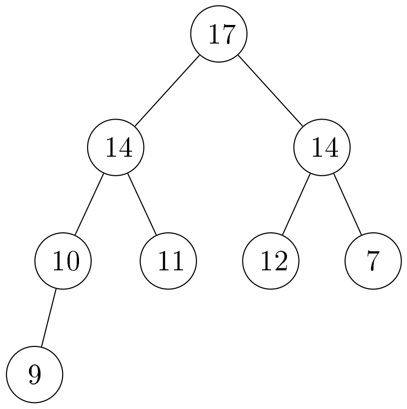
```
struct doublyLinkedListNode {
        unsigned int value;
        void *task;
        int mode;
        volatile struct doublyLinkedListNode *prev;
        volatile struct doublyLinkedListNode *next;
};
```

The structs name gives away that we are going to insert the created structs in a doubly linked list. This is done so that we can iterate over a sorted list and assign the priorities. The struct contains a value field for the modes period, a pointer to the corresponding TCB and a field for the tasks mode. We iterate over our list of tasks stored in *pxReadyTasksLists[1]* and for each mode insert a *doublyLinkedListNode* into our sorted doubly linked list. This leaves us with a list of all modes in a sorted order which we can track back to the corresponding tasks. To be able to assign the priorities statically each task also requires an array of priorities to choose its current priority from. Therefore we add a field *unsigned int *uxPriorities* to the TCB. The values for this array are then set by iterating over the list of modes in a for-loop and for each mode do:

$$TCB \rightarrow uxPriorities[mode] = configMAX\_PRIORITIES - i - 1$$

We subtract an additional 1 from the priority as the highest priority is not to be assigned in FreeRTOS. Finally we have to move each task to the ready-list of its current priority. In order to do so, we remove each list from the list of readied tasks and insert it again. As our priorities are assigned correctly by the procedure explained above, the tasks will get inserted in correct order and dependent on their current task. Thus, using our implementation of *vAssignPriorities()*, we meet our requirements for the rate-monotonic scheduling.

Additionally we want to solve the problem of choosing the highest priority task to execute in constant time and highlight that our improvements greatly reduce the systems overhead. To achieve this we need a data structure in which we can store the priorities of readied tasks. This data structure does not necessarily need to be fully ordered as long as it allows choosing the greatest value stored in it in constant time. Consequently we choose a binary heap, a data structure which can be viewed as a binary tree while it is actually an array object. A binary heap is represented by the attributes $A.length$, the length of the array A, and $A.heap-size$, the number of elements currently in the heap. The heap gets filled like a binary tree except for the lowest level which is filled starting from the left up to a certain point [8]. Figure 3.2 shows how priorities can be contained in a binary heap in descending order. The figure can be read as eight readied tasks with the highest priority task having a priority of 17 and two tasks with priority 14.



Index	1	2	3	4	5	6	7	8
Key	17	14	14	10	11	12	7	9

Figure 3.2: Binary min heap representation with 8 nodes and its actual array

For the scheduling process we are only interested in the time needed for inserting, extracting and finding elements in the heap. According to table 3.1 we can find the maximum priority in constant time $\theta(1)$. In order to make use of the binary heap we need

find-max(A)	extract-max(A)	insert(A, k)
$\Theta(1)$	$O(\log n)$	$O(\log n)$

Table 3.1: Running time of binary heap operations [8]

to implement some additional modifications:

1. Upon readying a task, insert its priority in the heap
2. Assign the highest priority task with $uxTopReadyPriority = heap[0]$ in *vTaskSwitch-Context()*
3. Extract the highest priority from the heap in the call of *vTaskPeriodicDelay()*

The reason for extracting the priority after the task is done executing instead of when it gets selected by the dispatcher is that if the task gets preempted, it cannot be chosen again if its priority has already been extracted. As a task in our model calls *vTaskPeriodicDelay()* after finishing its function, we can extract the priority right before adding the task to the list of delayed tasks in said function. The task cannot get interrupted by a higher priority task while doing so as all other tasks are suspended during the process.

Even though a binary heap allows us to dispatch tasks in constant time, it also generates additional overhead during the tick interrupt, precisely after adding a task to the ready queue. Table 3.1 shows that inserting a priority takes $O(log\ n)$ time. It would be more efficient if a priority could be inserted in constant time as well. Therefore we implement a pairing heap alongside the binary heap.

A pairing heap can be viewed as a heap-ordered tree similar to the binary heap. To make our implementation efficient we will use the child-sibling representation of a tree, also known as the binary tree representation [12]. In this representation we have a half-ordered binary tree, where half-ordered means that the key of any node is at least less than the key of any node in its left subtree. Figure 3.3 shows a visualization of a pairing heap [10].

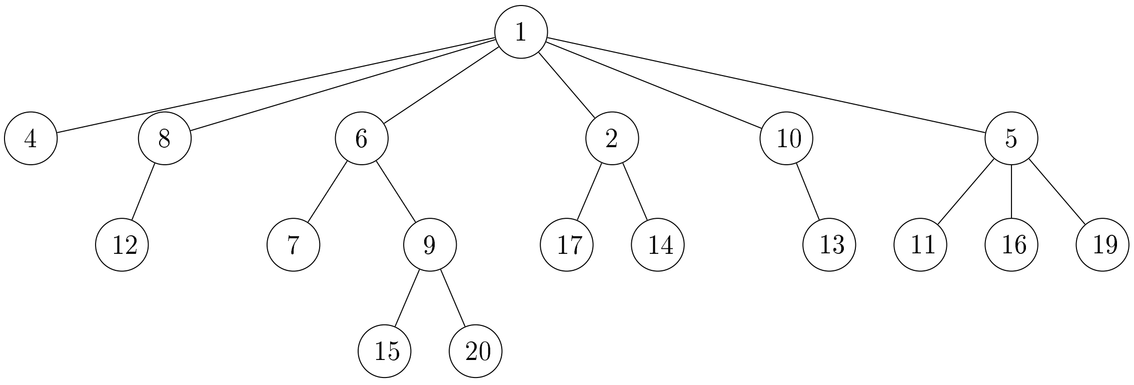


Figure 3.3: A pairing heap tree representation.

We will not go into detail about the specifics of a pairing heap and instead focus on the properties which we can exploit to enhance our RM scheduler. The table 3.2 summarizes the running time of pairing heap's operations.

find-min(A)	extract-min(A)	insert(A, k)	meld(A_1, A_2)
$\Theta(1)$	$O(\log n)^1$	$\Theta(1)$	$\Theta(1)$

^{1} Amortized time.

Table 3.2: Running time of pairing heap operations

Pairing heaps turned out to not be an efficient choice for the system as for each task an additional node structure has to be created and kept at runtime. This required more memory than accessible for a high number of tasks. Therefore we do not include the pairing heap in our evaluation and the final system does not support it. It will still remain in the system as a choice between the standard FreeRTOS dispatcher, binary heap and pairing heap. We also omit the pairing heap from the comparison of generated overhead displayed in figure 3.4 as scheduling 100 or more tasks using pairing heaps was not possible in our system.

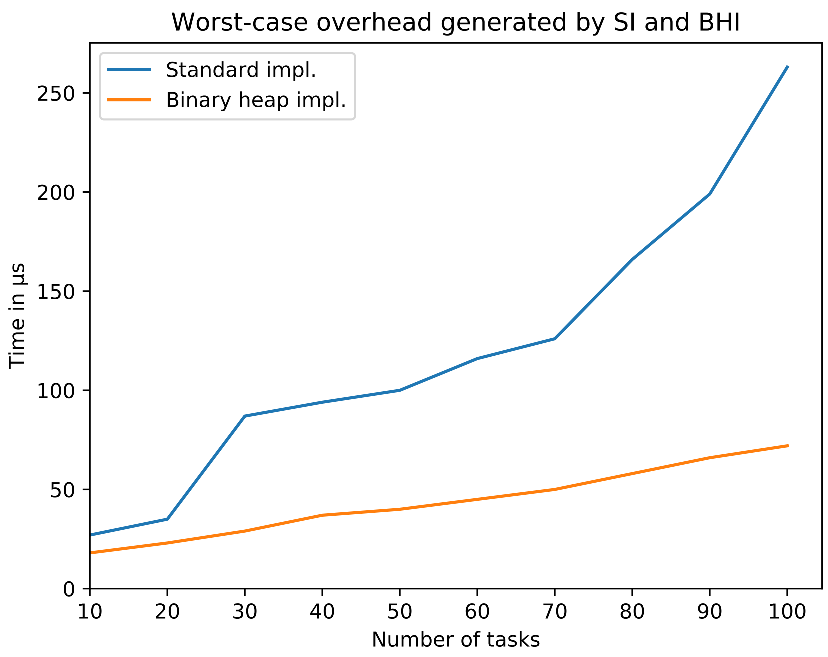


Figure 3.4: A comparison of generated overhead by the standard implementation(SI) and the binary heap implementation(BHI)

3.3 Earliest-Deadline-First Scheduler

This section will cover our design plans and implementation for the Earliest Deadline First scheduler as described in section 1.2.3.

Before beginning with the implementation of EDF we will make changes to our task creation functions just as we did for RM. Real-time tasks will be scheduled after their absolute deadline, therefore we omit the priority(set it to 1) in the call of *xTaskCreate()*. Non-real-time tasks on the other hand have no deadline and need to be executed with a lower priority. Hence, we create the non-real-time tasks with a deadline which is set to the highest value possible and subtract that value with the chosen priority for the task. This way the tasks can still be executed in the order of their priorities but will not be executed by the CPU if a real-time task is ready. Now, with tasks being created with their appropriate parameters we can start implementing the EDF scheduler.

First of all, we have to take a look at the data structure used in FreeRTOS. As described in section 1.2.5 FreeRTOS uses an array of linked lists to store the readied tasks according to their priority. Since arrays have a fixed length in C they are not that suitable for using a dynamic scheduling algorithm as the number of tasks might change during the evolution of the system. Of course arrays could still be used but that would mean losing the advantage of having a variable number of tasks or the array would need to be big enough for any eventual number of tasks. Another option is to reallocate the size of the array but that is not always possible and time consuming. Thus, instead of using an array of lists we reduce the structure to just one doubly linked list which will contain all the readied tasks. Doing that allows us to implement the EDF scheduler while keeping modifications to the system at a minimum. By keeping the amount of modifications low we keep the stability of the system and reduce the potential for faulty behavior. This modification is implemented by removing the array property for every occurrence of *pxReadyTasksLists* and changing the logic of the surrounding code to fit the new structure, i.e. removing for-loops that looped over the list. Any occurrence of *pxReadyTasksLists[configMAX_PRIORITIES]* is changed to *listGET_OWNER_OF_HEAD_ENTRY(&pxReadyTasksLists)* as the head of this list will always be our highest priority task.

Now all tasks get inserted in the same list on creation but they are not ordered by their absolute deadline yet. Lists in FreeRTOS are storing their items in ascending order of their value. We can use this property to employ an EDF scheduling policy. Therefore we add the absolute deadline to the inserted item before the call of *vListInsert* during the execution of *prvAddTaskToReadyQueue()*. This can be done by using the following function from list.c:

```
listSET_LIST_ITEM_VALUE(&( pxTCB->xGenericListItem ),
                            pxTCB->uxPreviousWakeTime +
                            *( pxTCB->uxDeadline + pxTCB->uxMode ) );
```

We then insert the item linked to the task TCB with *vListInsert()* instead of *vListInsertEnd()* as the former iterates through the list and inserts items by ascending order of their values. Additionally we iterate through the list of readied tasks to assign the task priorities in descending order. Potentially, this step could be left out as only the task at the head of *pxReadyTasksLists* will be executed at any time and thus there is no need for a priority. However, some functionalities of the system might need a priority value so we will keep the parameter. The worst-case time our scheduler needs for scheduling one task is $O(n)$.

3.4 Additional Modifications

Shared Processor Behavior

In this section we will perform an additional change to the system in order to improve the overall performance and to make our EDF execute appropriately. In FreeRTOS tasks standardly share the processor if they have the same priority. This behavior is achieved by setting the pointer to the currently executing TCB to the next TCB on the list during each tick interrupt. If the list only contains one item the pointer is set to that item. Two problems arise from this procedure. First, in order to implement our EDF scheduler we reduced the array of ready lists to only one ready list containing all tasks. Therefor, all readied tasks would share the processor without considering their priority. Secondly, the system needs to save the old tasks state and restore the new tasks state each system tick which results in a high overhead for a low interrupt tick frequency. Figure 3.5 shows an exemplary task execution of two tasks with the same priority in FreeRTOS. The figure is a simplification to visualize the cost of context switching and not actual footage of the system. The area between the dashed lines in the figure represent the context switch. This figure exaggerates the time needed for performing a context switch. The actual cost of switching between two tasks is measured approximately $4\mu s$ per context switch.

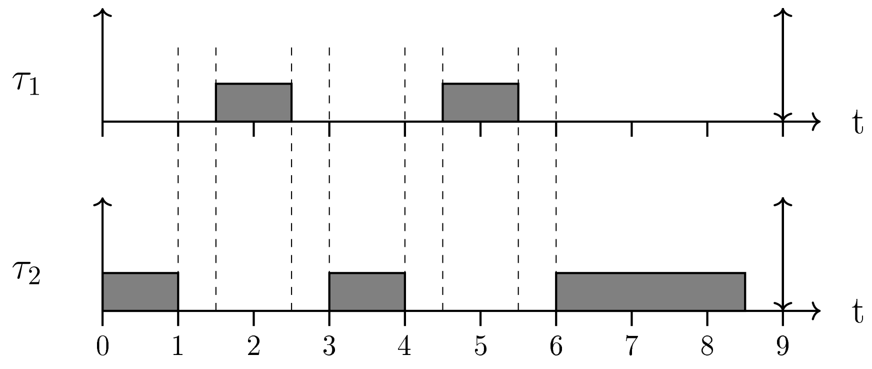


Figure 3.5: Two tasks with the same priority sharing the processor

We solve this problem by setting *xCurrentTCB* to the head of the corresponding ready list instead. Furthermore we only perform a context switch only if the highest priority changed or if the current task moved to the blocked state. Context switches are then only conducted when necessary. Tasks with the same priority are going to run in order of their transition to the ready state.

Configuration and Task Creation

As mentioned before we handle tasks differently depending on the values passed by the call of *xTaskCreate()*:

Real-time-task: uxPriority = 1

Non-real-task: deadline = 0

Periodic: period[0] > 0

Aperiodic period[0] = 0

These values can be mixed to create tasks with the desired functionality. Non-real-time tasks are not scheduled and aperiodic tasks have to take care of recurrent execution themselves. Also aperiodic tasks should delete or suspend themselves if they are not going to be executed again in the evolution of the system.

We add several configurations to the system which will be needed for the evaluation or visualization of the scheduling. The following constants are added to the file *FreeRTOSConfig.h*:

configANALYSE_METRICS: Allows tracking of data for the metrics

configANALYSE_OVERHEAD: Counts the times needed for the tick interrupt

configPLOTTING_MODE: Tracks task parameters at context switches

configTICKS_TO_EVAL: The time in milliseconds for any of above modes to run

configEVAL_THRESHOLD: The time between evaluations, must be big enough for tasks to delete themselves

configUSE_TASK_SETS: 1 if task sets are used, 0 otherwise

configSET_SIZE: The number of task sets that are used

configNUMBER_OF_TASKS: The number of tasks in each set

A python script using for each configuration is provided. The "plotting" mode can be used with the script *plotter.py*. For using the task set loading configuration the application needs to implement the function *createSet(int i)* in *main.h*.

3.5 Test Design and Task Generation

In this section we are presenting our test procedures for the created model and explain how we generated the tasks for the tests. For the evaluation we set *configANALYSE_METRICS* and *configUSE_TASK_SETS* to 1. The corresponding configuration values are set according to the values described for each test design below.

According to the simulation done by Huang and Chen EDF is expected to perform poorly with increasing number of modes and increasing proportion of multi-mode tasks for a set number of modes [11]. In order to check if this behavior is similar for our system model we implement a python script to generate tasks in a similar manner as it is done in their simulation. Furthermore we generate task sets with timing characteristics similar to applications of a real-world automotive software system.

We use the boot-over-serial bootloader *raspbootin* to load the compiled kernel for each evaluation on our hardware to speed up the evaluation process.

Randomized Task Sets

For the the first procedure we begin by generating a set of utilization values for a given total utilization and number of tasks. This can be done by using the UUniFast algorithm [3].

Following, the tasks real-time constraints have to be calculated accordingly. Therefore we choose a similar approach as in *Efficient exact schedulability tests for fixed priority real-time systems* [9] and generate periods in the range of 1-100ms from an exponential distribution. The WCET C_i of each task could then be calculated by $T_i * U_i$ and deadlines are implicit. Now that the tasks are generated a proportion p of those tasks are converted to multi-mode tasks with M modes. Actually the script converts every task into a multi-mode task with non-multi-mode tasks having $M = 1$. The first mode of each task is assigned the aforementioned generated values. If a task is a multi-mode tasks the values for its remaining modes are scaling by the factor 1.5, i.e., $C_i^{m+1} = 1.5 * C_i^m$, $T_i^{m+1} = 1.5 * T_i^m$. For each multi-mode task one of the modes is then chosen to have the highest utilization while the worst-case execution times of the other modes are reduced by multiplying them with random values between 0.75 and 1. The needs to be called over a terminal and requires a set of parameters to be passed in the order described below:

Cardinality: The cardinality of each set

Utilization: The total utilization of each set

Minimum period: The lower bound of task periods

Maximum period: The upper bound of task periods

Number of modes: The number of modes multi-mode tasks have

Proportion: The number(not percentage) of multi-mode tasks

Number of Sets: The number of sets that will be created

An exemplary call would be: python3 taskGen.py 10 60 1 100 5 5 100
Setting the minimum and maximum period to the same value will result in tasks having exactly that value as their period. Using the following parameters we generate and evaluate the system behavior 100 task sets:

Cardinality: 10

Modes: 5, 8, 10

Multi-mode tasks: 50%

Utilization: 10-100% in steps of 10

Each task set is going to run on the system for 10 seconds plus a threshold of 5800 milliseconds for each task to delete itself. The data sent by the hardware is processed by the script *metrics.py* which is making use of a serial connection to first send over the kernel and then receive the data.

Realistic Task Sets

The second set of test data is created by using a script written by Georg von der Brüggen. It allows the creation of sets of tasks which share the characteristics of a automotive software system as presented in *Real World Automotive Benchmarks For Free* [13]. These characteristics cover the distribution of tasks among periods, the typical number of tasks, average execution times of tasks and factors for determining the best- and worst-case execution times. Table 3.3 shows the distribution of tasks among periods [13].

Period	**Share**
1 ms	3 %
2 ms	2 %
5 ms	2 %
10 ms	25 %
20 ms	25 %
50 ms	3 %
100 ms	20 %
200 ms	1 %
1000 ms	4 %
angle-synchronous ms	15 %

Table 3.3: Task distribution among periods

The angle-synchronous tasks which take 15% of all tasks are converted to multi-mode tasks as their worst-case execution time needs to adapt to their reduced period. In our case the maximum engine speed is 6000rpm with 4 available cylinders. For the conversion to multi-mode tasks we will divide the engine speed into 6 intervals and calculate the periods by the upper bound of each mode as displayed in table 3.4.

Mode:	0	1	2	3	4	5
Range:	0-1000	1001-2000	2001-3000	3001-4000	4001-5000	5001-6000
Period:	30 ms	15 ms	10 ms	7.5 ms	6 ms	5 ms

Table 3.4: 6 modes ranging from 0-6000rpm with their periods

The WCET generated by the script was assigned to the lowest mode and the remaining modes had their WCET calculated regarding the first modes utilization, i.e. $C_i = T_i * U_1$, with $U_1 = \frac{C_1}{T_1}$. Therefore all modes have the same utilization with decreasing period and WCET for higher modes.

With the goal of creating a realistic environment for our system we make some changes to our *metrics.py* script. Instead of sending a signal every 5 ms and traversing through the modes we implement a crankshaft simulation. The simulation starts at an angular speed of 1rpm and increases by 1000 rpm over 500 ms while sending a signal to the pi every time the piston reaches is maximum position. This happens every time after one full rotation of the crankshaft. Once the simulated crankshaft gains its highest speed of 6000 rpm it will slow back down to 1 rpm. The acceleration/deceleration is steady during the whole execution, it only changes its sign. The actual angular speed is not transmitted as our application has no use for it. Instead we will use the modes' numbers as breakpoints for mode changes. We generate 100 task sets per utilization 10-100 in steps of 10.

The data sent by the hardware is processed by the script *metrics2.py* which is making use of a serial connection to first send over the kernel and then receive the data. The crankshaft simulation is also implemented in this script.

Hardware and External Interrupt

The hardware used for our evaluation is a Raspberry Pi B+ with the following specifications:

CPU Type/Architecture/Family ARM1176JZF-S/ARMv6(32Bit)/ARM11

CPU Frequency: 700MHz

Cores: 1

RAM: 512MB

Chipset: BCM2835

Figure 3.6 shows the Raspberry Pi B+ which is used in our evaluation.

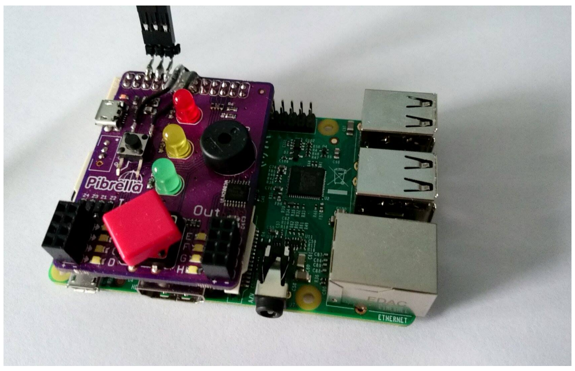


Figure 3.6: The Raspberry Pi B+ with attached Pibrella Board

The port of FreeRTOS to the Raspberry Pi has been released by James Walmsley on GitHub and is a community project. GPIO drivers and a global IRQ handler are part of the project. While implementing the external interrupt we found an error in the IRQ handler which made it impossible to access particular interrupts. To be specific, the upper

10 bits of the of the basic ARM pending interrupt register were not checked in the handler. These bits are used to show that selected interrupts from the GPU are pending. In the original implementation only the lower 7 bits and bit 8 and 9, which signal that interrupts unknown to the CPU register are pending, are processed. The 10 GPU interrupts which could cause the problem are from the GPU side, namely GPU IRQ 7, 9, 10, 18, 19, 53, 54, 55, 56, 57, 62. However, they would not cause bit 8 or 9 of the register to be set, because according to the manual those bits are only for interrupts which are not connected to the basic pending register [2]. We could fix the error by adding the condition:

```
if(ulMaskedStatus & 0xFFC00){
        handleRange(ulMaskedStatus & 0xFFC00 & enabled[2], 64);
}
```

For our evaluation we setup the UART interface of the Raspberry Pi B+ and configure it to cause an interrupt upon receiving a byte. The corresponding interrupt service routine reads the byte and sets the global variable *externalInput* to the number of the mode determined by the script used in the respective evaluation. The function *setupUARTInterrupt()* for setting up the UART interface is implemented in the file uart.c located in the drivers folder. *setupUARTInterrupt()* is then called at the start of our main thread.

Chapter 4

Results and Discussion

In this chapter we submit the results of the evaluation for each test procedure and discuss the results.

4.1 Results

In this section we present and explain the results from each test. We address the results of each test result separately.

Scheduling Overhead

We start by the comparison of each schedulers overhead. Figure 4.1 shows the maximum and median overhead generated by each scheduler. Tasks' modes were not updated for measuring the overhead. It can be seen that RM has an advantage over EDF. This can

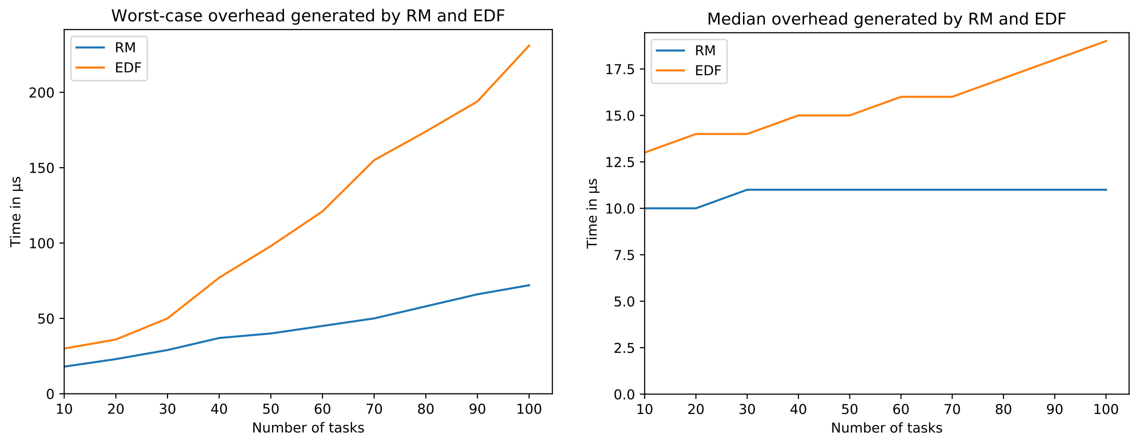


Figure 4.1: A comparison of the maximum and median overhead generated by RM and EDF for increasing number of tasks

be explained by the simplicity of RM's static implementation while the EDF scheduler has to iterate over the whole list of readied tasks each time a task readies. Let n be the cardinality of the task set. In the worst-case of all tasks of the task set becoming ready at the same time, the time needed for scheduling is $O(n^2)$. Precisely, we receive the worst-case scenario when all tasks of the set get inserted with equal or ascending absolute deadline as we then have to iterate over the whole lists for each task. The worst-case time can then be calculated by $\sum_{i=1}^{n} i = \frac{n^2+n}{2}$, thus $O(n^2)$. For our RM scheduler using the binary heap the worst-case time needed is $O(n * log\ n)$ when all tasks arrive at the same time. The right figure also shows that RM performs better when comparing the median values.

First Test Procedure

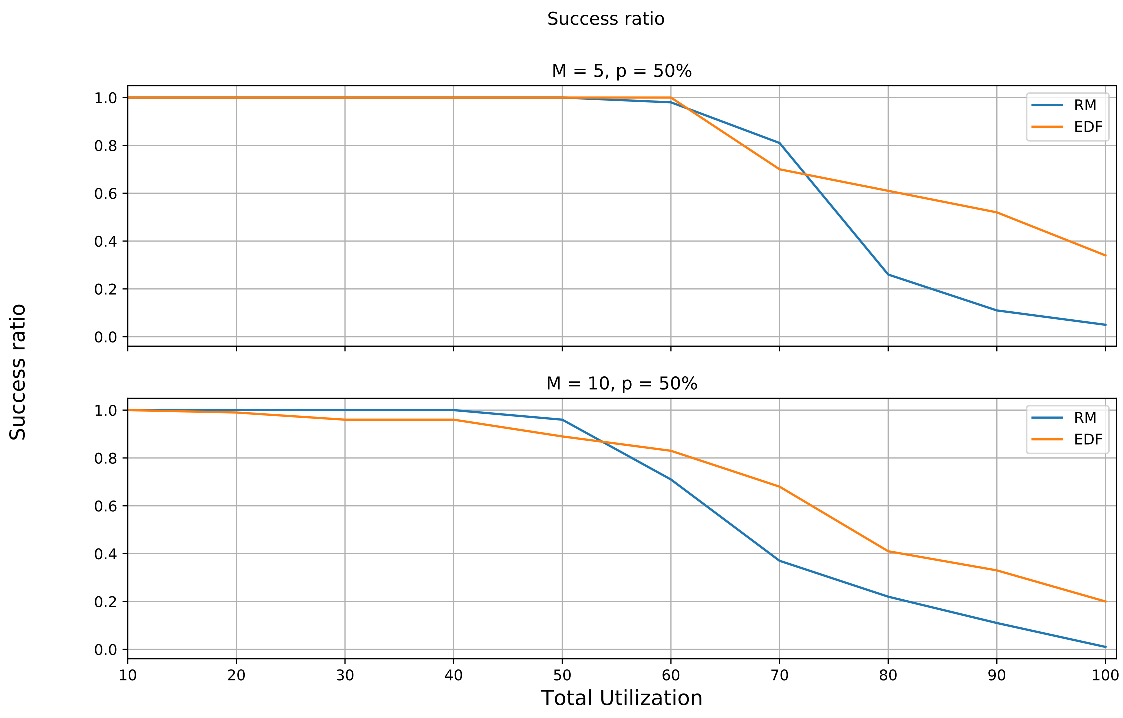


Figure 4.2: The success ratio of all task sets under RM and EDF in %

We begin by comparing the success ratio, which is the number of task sets that are schedulable divided by the number of task sets. Figure 4.2 shows that EDF suffers more from an increasing amount of modes than RM. While RM is able to schedule all of our task sets for up to 40% utilization, EDF can only achieve that for up to 20%. Nevertheless, for EDF more task sets were found schedulable than for EDF for a total utilization of 50% and above.

Continuing with the results it is noticeable that the number of missed tasks only increases slightly compared to EDF when applying the rate-monotonic scheduling algorithm. This can be seen in figure 4.3.

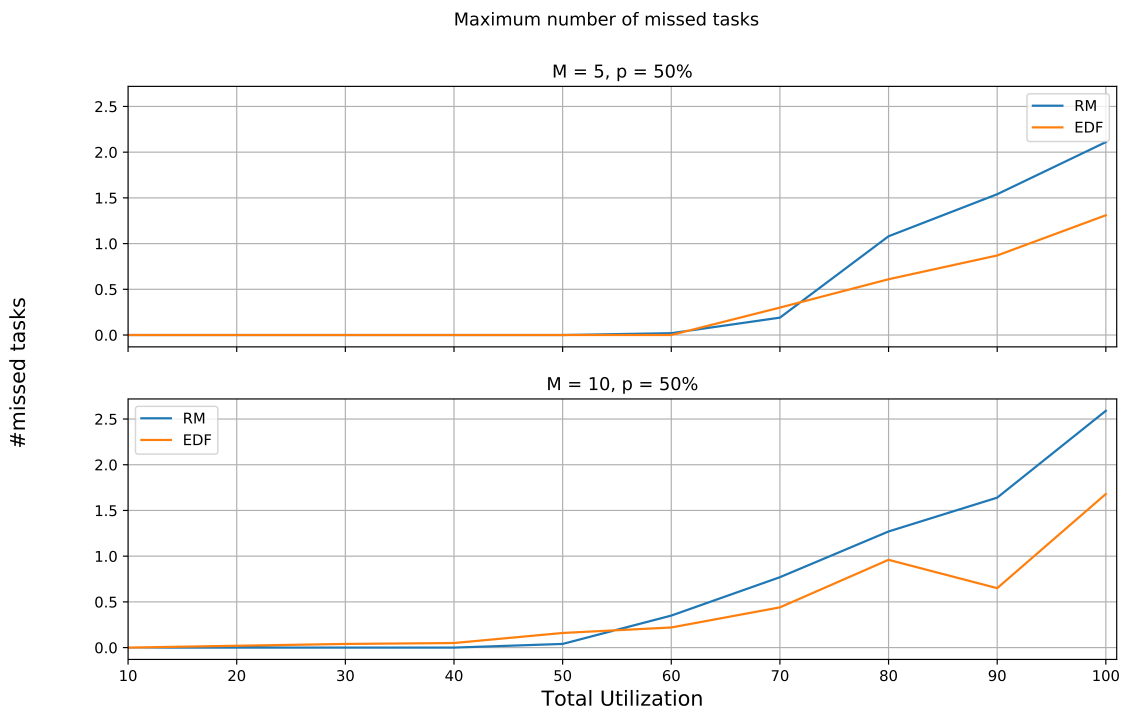


Figure 4.3: The maximum number of missed tasks per task set

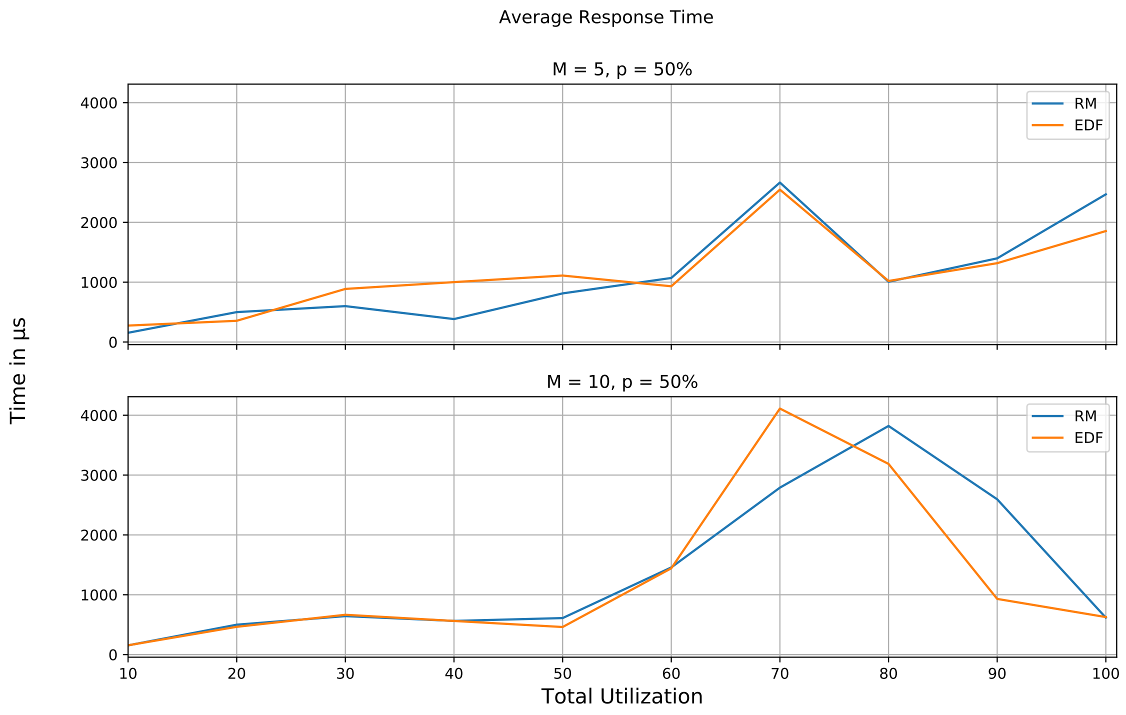


Figure 4.4: The average response time in microseconds with increasing utilization

The most noticeable result in the comparison of the average response time in figure 4.4 is the peak for both schedulers at 70% utilization which shifts to 80% utilization for RM for a number of 10 modes. These peaks can be explained by the way we calculate the average response time as we can only measure it for tasks that are actually executing at some point. So if a task is preempted by higher priority tasks until the end of the evaluation, its response time can not be considered and does not add to the average response time. We conclude that for a total utilization of over 70% for EDF and 70% or 80% for RM(depending on the number of modes) the number of tasks that are delayed until the end of evaluation rises drastically. This results in a lower average response time which only considers the high priority tasks. The design of the task sets for which the task's WCET increases by the factor 1.5^m per mode m is the reason for the higher worst average response time as it can be seen in the plot for M = 10.

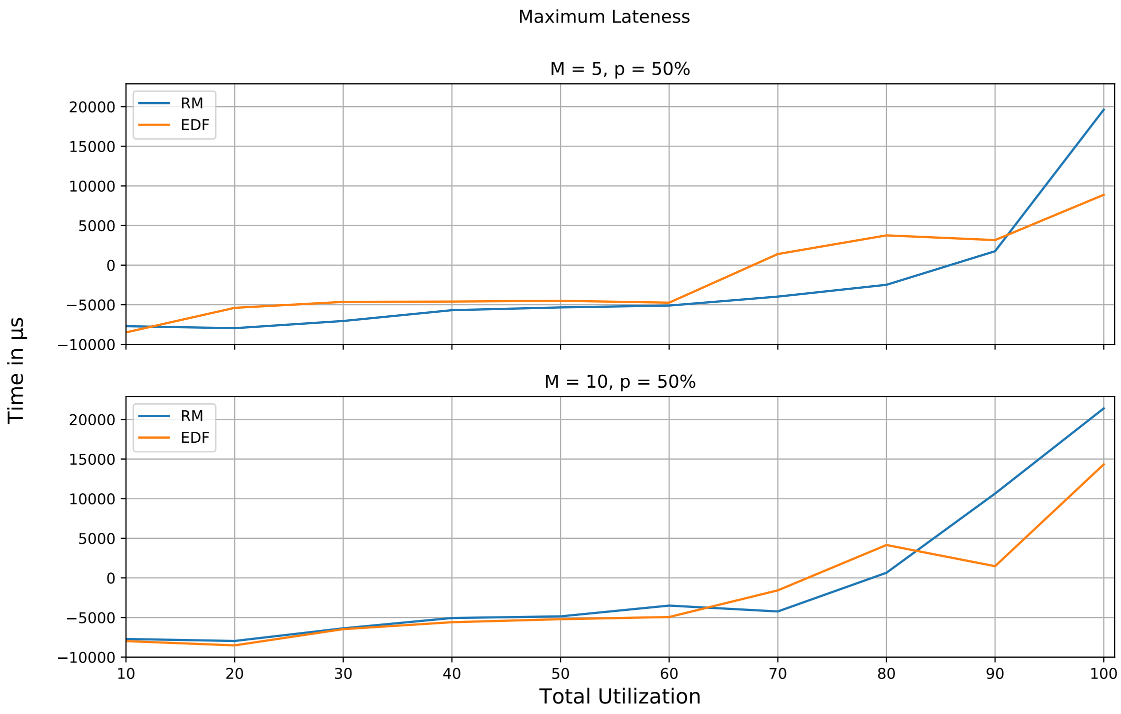


Figure 4.5: The maximum lateness in microseconds with increasing utilization

We could not draw any conclusion about the effect of multi-mode tasks on the maximum lateness from the first test procedure. As shown in figure 4.5 the values only differ minimal which could be caused by the diversity of the task sets used for the different numbers of modes.

Second Test Procedure

In the following we will present the results of the second test procedure.

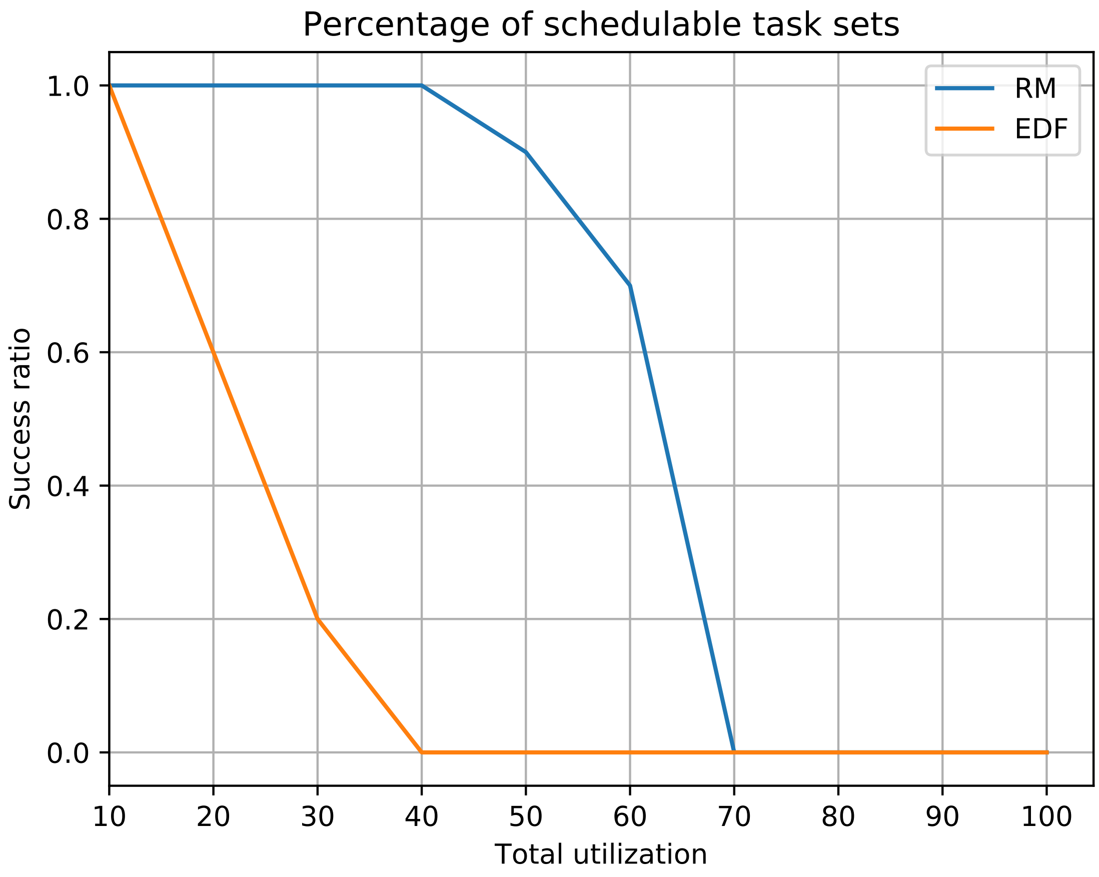


Figure 4.6: The success ratio of all tasks under RM and EDF in %

The earliest-deadline-first scheduler was performing poorly when using realistic task sets. The generated task sets were not found schedulable for a total utilization of over 30% while under RM a schedulability of up to 60% could be achieved. However, neither scheduler could schedule any of the task sets with a total utilization of 70% and above.

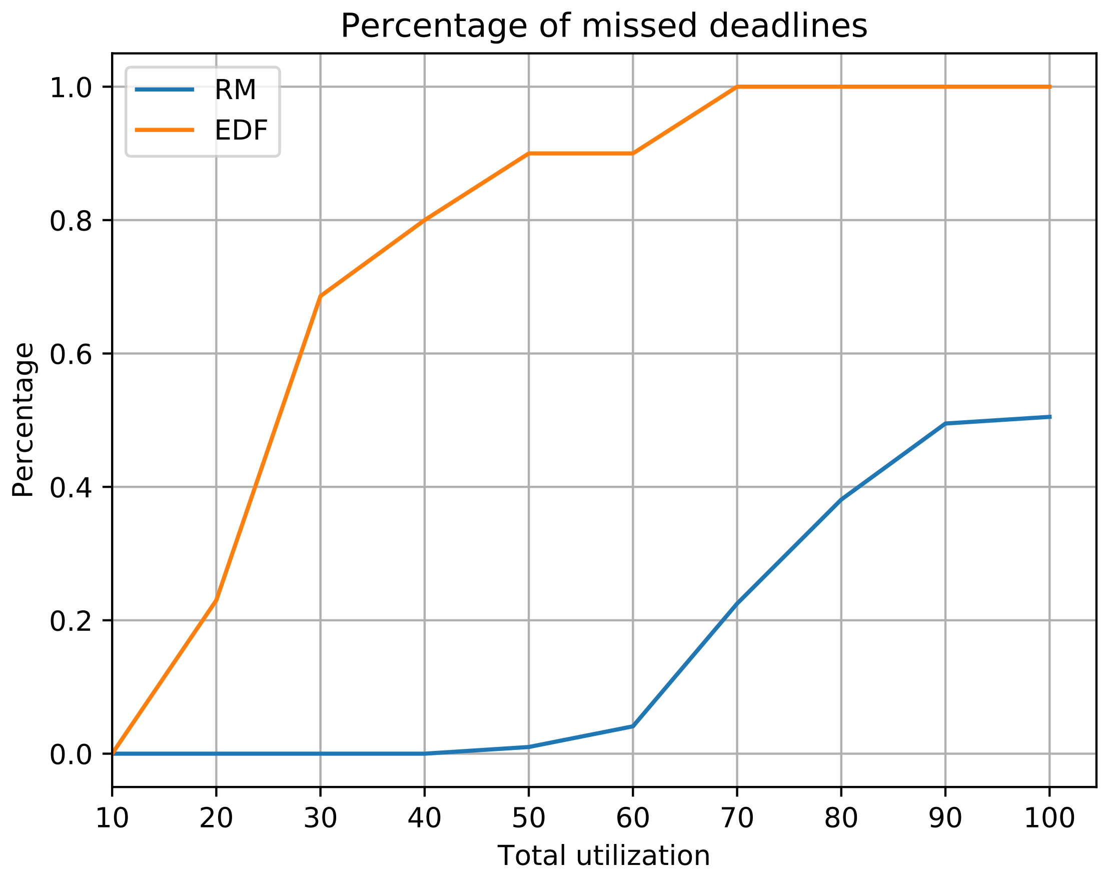


Figure 4.7: The ratio of missed tasks under RM and EDF in %

In addition to the schedulability we compare the percentage of missed tasks for each utilization. We can see that while none of the task sets were schedulable under RM for a total utilization of 70% and above it was still possible to successfully execute around 50% of the tasks. The results shown in figure 4.7 for EDF are much worse here because tasks with small periods which are guaranteed to execute under RM can be preempted by tasks which are close to their deadline. This allows all tasks to eventually finish their execution for the price of missing their deadlines.

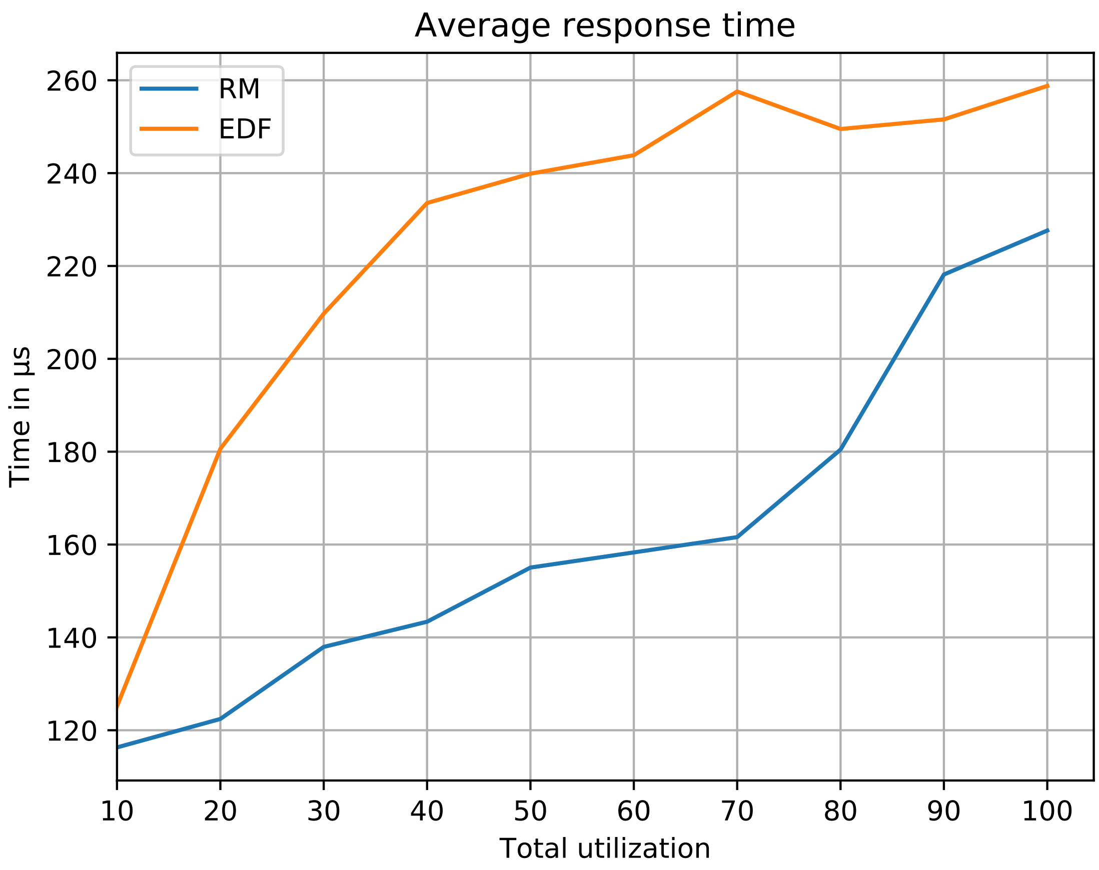


Figure 4.8: The average response time in microseconds with increasing utilization

Figure 4.8 displays the average response time for EDF and RM for the second test procedure. The findings are different from the first test procedure where both schedulers had similar results and a peak at 70%. The average response times for RM and EDF go along with our results for the schedulability and the percentage of missed tasks. For higher utilizations more preemptions occur under both scheduling algorithms, thus increasing the response time of lower priority tasks. While multi-mode tasks with sizable WCET might dominate each other under EDF [11] that is not the case for RM because of the static priority assignment.

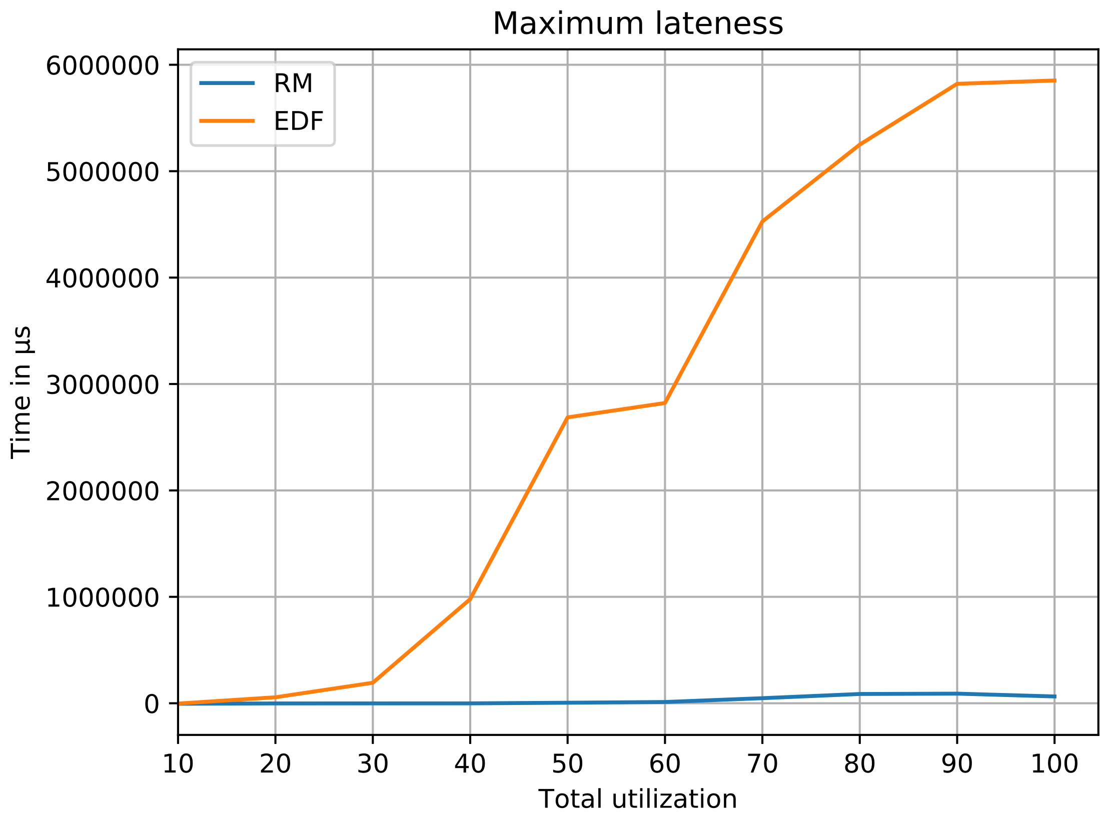


Figure 4.9: The maximum lateness in microseconds with increasing utilization

Finally, we are comparing the maximum lateness produced by each scheduler respectively. The results are shown in figure 4.9. Again, the results from the second test procedure differ from the first test procedure. Under RM the maximum lateness is relatively small even when the task set is not schedulable. Our results for the percentage of missed tasks and the way in which we retrieved the data can be used to explain this. As mentioned before we only retrieved the arrival and finish times of tasks that actually finished at some point during the evaluation. It occurs that under rate-monotonic scheduling, tasks with low priorities might be preempted by higher priority tasks for the whole duration of the evaluation. Therefore, only the maximum lateness of high priority tasks is considered. Under EDF on the other hand, tasks will eventually finish their execution once they get close enough to their deadline and thus contribute higher values to the maximum lateness.

4.2 Discussion

In this section we discuss the results from the two testing procedures which have been conducted.

First of all, it has to be said that the first procedure does not show precise results from which a conclusion can be drawn. For more evident results a higher number of modes and longer evaluation times are needed. This would have exceeded the time limits of this thesis. Nevertheless, we can identify a downward trend in performance for the EDF scheduler with increasing number of modes while RM on the other hand keeps more stable. With the exception of the maximum lateness, EDF has performed slightly worse when setting the number of modes from 5 to 10. Still, overall EDF performed better than RM regarding high values for the total utilization. One reason for this kind of test procedure favoring EDF is that we only generated 10 tasks per set. As shown in our comparison of overhead the difference between the schedulers is only around 3μs for 10 tasks. Therefore, the overhead does not affect the performance as much. Furthermore, the worst-case overhead is not guaranteed to occur as the periods are randomly generated and can be up to 5567ms for 10 modes. Regardless of these results we show that RM is more suitable for scheduling multi-mode tasks in a real environment by analyzing the second test procedure which is more realistic. The task sets used in the second test procedure feature tasks with a period of 1ms which put a lot of pressure on the system considering generated overhead and tasks with periods that overlap often during the evolution of the system. Moreover all non-multi-mode tasks which make up 85% of the system are guaranteed to arrive each 1000ms as that is the least common multiple of the corresponding periods. The task sets generated for the second procedure are therefore determined to create higher overhead which results in a lower amount of schedulable task sets for EDF. As explained in section 4.1 the percentage of missed tasks is higher for EDF than for RM because EDF sacrifices keeping the deadline for letting tasks finish their execution. Considering that, in cases where hard real-time tasks are present, RM will be the preferred choice as missing a deadline can result in catastrophic consequences.

Finally, it has to be mentioned that these results do not consider task overruns which can potentially occur upon a deadline miss as presented in *Overrun Handling for Mixed-Criticality Support in RTEMS* [7].

Chapter 5

Conclusion

5.1 Summary

In this thesis we compared multi-mode tasks under EDF and RM scheduling in a real environment using real hardware. We successfully implemented the task model and each scheduler. We realized a rate-monotonic scheduling algorithm that can schedule multi-mode tasks and leaves no unused priorities in the system. In order to reduce the systems overhead we removed the shared processor behavior that is present in FreeRTOS. Additionally, we reduced the worst-case time for the tick interrupt by implementing a binary heap that stores the priorities for the rate-monotonic scheduling. The earliest-deadline-first scheduling algorithm was realized by inserting task that become ready into an ordered doubly linked list. The tasks get inserted in ascending order by their absolute deadline. An external interrupt triggered through the UART when receiving a byte was also added to the system. To achieve this a missing part in the corresponding driver had to be supplemented. Moreover, we expanded the system by configurations that help with evaluating the system and provide tools to generate two different kinds of task sets. In order to come closer to a real-world situation which benefits from the usage of multi-mode tasks, we implemented a crankshaft simulation. The need for multi-mode tasks when implementing angular tasks with periods tied to the rotation of the crankshaft is an example that was brought up in the introduction and carried through the work.

We found out that EDF allows scheduling of task sets of higher total utilization when tasks in the task set have long periods with mostly non-overlapping arrival times. The reason for this is that the overhead generated by EDF is not affecting the system as much for those kinds of task sets. While we showed that EDF outperforms RM in the first test procedure, it still yielded worse results for each evaluation metric with increasing number of modes. The rate-monotonic scheduler on the other hand performed significantly better than EDF when using realistic task sets which contained a fixed distribution of tasks among periods. Furthermore, the rate-monotonic scheduler proved to perform more stable with increasing numbers of modes in the first test procedure. The results of the first test procedure, even if they are not as precise, match the results from the simulation done by Huan and Chen which showed that the performance of EDF drops when the number of modes increases [11]. Reviewing the results of our evaluation we come to the conclusion that EDF performs poorly in a real environment compared to RM as we showed with the simulation of the crankshaft.

5.2 Future Work

In this work we evaluated multi-mode tasks in a real environment under the rate-monotonic and earliest-deadline-first scheduling algorithms. The modified operating system FreeRTOS has been tested on the Raspberry Pi B+ which was first released 3 years ago. Microprocessor technology though, is developing rapidly [18]. Future analysis could therefore be done using different and more efficient hardware as our designed system can be used on any hardware for which a port to FreeRTOS is provided. For further enhancements the binary heap used in the implementation of the rate-monotonic scheduler can be improved by using a hardware accelerated binary heap as it is proposed in *Hardware-software architecture for priority queue management in real-time and embedded systems* [14]. That approach allows insert operations in time $O(1)$. As mentioned in the discussion of the results in section 4.2, the current system does not consider task overruns. The detection of deadline misses however, is possible by comparing the current system tick against the previous wake time and period of a task. Therefore future research could investigate in the problem of handling task overruns in a multi-mode task model. In addition to that a real application for FreeRTOS which could benefit from using multi-mode tasks could be modified to fit our designed system. The benefits and drawbacks of such modifications could then be evaluated.

List of Figures

Bibliography

[1] *http://www.freertos.org.*

[2] *BCM2835 ARM Peripherals.* Broadcom Europe Ltd. 406 Science Park Milton Road Cambridge CB4 0WW, 2012.

[3] BINI, ENRICO and GIORGIO C BUTTAZZO: *Measuring the performance of schedulability tests.* Real-Time Systems, 30(1):129–154, 2005.

[4] BIONDI, ALESSANDRO and GIORGIO BUTTAZZO: *Real-time analysis of engine control applications with speed estimation.* In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 193–198. IEEE, 2016.

[5] BUTTAZZO, GIORGIO C.: *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 2004.

[6] BUTTAZZO, GIORGIO C, ENRICO BINI and DARREN BUTTLE: *Rate-adaptive tasks: Model, analysis, and design issues.* In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6. IEEE, 2014.

[7] CHEN, KUAN-HSUN, GEORG VON DER BRÜGGEN and JIAN-JIA CHEN: *Overrun Handling for Mixed-Criticality Support in RTEMS.* In *WMC 2016*, 2016.

[8] CORMEN, T. H., C. E. LEISERSON, R. L. RIVEST and C. STEIN: *Introduction to Algorithms.* MIT Press, Cambridge MA, 2009. 3rd edition.

[9] DAVIS, ROBERT I, ATTILA ZABOS and ALAN BURNS: *Efficient exact schedulability tests for fixed priority real-time systems.* IEEE Transactions on Computers, 57(9):1261–1276, 2008.

[10] FREDMAN, M. L., R. SEDGEWICK, D. D. SLEATOR and R. E. TARJAN: *The Pairing Heap: A New Form of Self-Adjusting Heap.* Algorithmica, 1(1-4):111–129, 1986.

[11] HUANG, WEN-HUNG and JIAN-JIA CHEN: *Techniques for Schedulability Analysis in Mode Change Systems under Fixed-Priority Scheduling.* 2015 IEEE 21st Interna-

tional Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 00:176–186, 2015.

[12] Knuth, D. E.: *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Boston, MA, 1997. 3rd edition.

[13] Kramer, Simon, Dirk Ziegenbein and Arne Hamann: *Real World Automotive Benchmarks For Free*.

[14] Kumar, NG Chetan, Sudhanshu Vyas, Ron K Cytron, Christopher D Gill, Joseph Zambreno and Phillip H Jones: *Hardware-software architecture for priority queue management in real-time and embedded systems*. International Journal of Embedded Systems, 6(4):319–334, 2014.

[15] Lee, Edward A: *Cyber physical systems: Design challenges*. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369. IEEE, 2008.

[16] Liu, Chung Laung and James W Layland: *Scheduling algorithms for multiprogramming in a hard-real-time environment*. Journal of the ACM (JACM), 20(1):46–61, 1973.

[17] Marwedel, Peter: *Embedded System Design*. Springer Netherlands, 2010.

[18] Naffziger, Samuel: *Technology impacts from the new wave of architectures for media-rich workloads*. In *VLSI Technology (VLSIT), 2011 Symposium on*, pages 6–10. IEEE, 2011.

[19] Navet, Nicolas and Françoise Simonot-Lion: *Automotive embedded systems handbook*. CRC press, 2008.

[20] Sha, Lui, Ragunathan Rajkumar, John Lehoczky and Krithi Ramamritham: *Mode change protocols for priority-driven preemptive scheduling*. Real-Time Systems, 1(3):243–264, 1989.