A dissertation submitted to the **University of Greenwich**
in partial fulfilment of the requirements for the Degree of

Bachelor of Science honours

in

Computer Science

Final Year Project: Implementation of EDF in FreeRTOS for Dynamic Task Scheduling

Name: Prayush Raj Udas

Student ID: 001049882

School of Computing and Mathematical Sciences
Faculty of Liberal Arts and Sciences

University of Greenwich

Supervisor: Dr Atsushi Suzuki

Submission date: April 2022

Word count: 10000

Final Year Project: Implementation of EDF in FreeRTOS for Dynamic Task Scheduling

Prayush Raj Udas

School of Computing and Mathematical Sciences
Faculty of Liberal Arts and Sciences

University of Greenwich

30 Park Row, Greenwich, United Kingdom

ABSTRACT. Rise of embedded systems has technologically evolved the way of life, making complex application possible. With these complexity in applications, there is a need for an operating system that achieves scheduling of tasks efficiently improving the provided quality of service. FreeRTOS is a lightweight real time operating system that handles scheduling via a static Rate Monotonic scheduling. After analysing the scheduler in depth, an alternative dynamic Earliest Deadline First (EDF) scheduling algorithm is proposed and implemented for FreeRTOS. Detailed implementation and procedure on EDF for FreeRTOS is presented with correctness verification test conducted.

Keywords: Real Time Operating Systems, Real time, Scheduling, FreeRTOS, EDF, RMS

Preface

Many topics and subjects are exposed as a BSc computer science student. But nothing stood out to me as Operating Systems and low level programming. Since my second year on my degree, learning about operating systems evoked a passion in me to understand the fundamental software that rules technology. The more I learned, the more I got curious, and this curiosity made me choose it as my dissertation in the field of Earliest Deadline First and Real Time Operating Systems. The research was difficult, but conducting extensive investigation has allowed me to grow and answer questions that puzzled me.

This dissertation conducts research on the scheduling algorithm that is crucial to every operating system. It more specifically focuses on dynamic EDF scheduling algorithm in Real time operating systems. These systems are all around us working indefinitely, used in everyday household products to nuclear power plants and UAV by the military. Detailed analysis of EDF will be presented to show the benefits and advantages it gains over the Rate Monotonic Scheduling(RMS) which is the most widely used and default scheduling algorithm in Real Time operating System, FreeRTOS. To test and evaluate the scheduler, it will be implemented on a STM32F413H-DISCO Evaluation board with a ARM Cortex M4 uniprocessor where it will be tasked to handle various periodic tasks with varying deadlines.

Acknowledgements

Throughout the writing of this dissertation I have received a great deal of support and assistance. Firstly I wish to show my appreciation towards my supervisor, Dr A Suzuki, whose guidance and help was crucial in appropriately organising and formulating my dissertation. I would like to acknowledge my peers for their support and company. We have been on the same journey and the time with you have entertained as well as excelled me towards completing this dissertation. In addition, I would like to thank my parents for their constant and continuous motivation. You were always there ready to uplift me, push me and counsel me throughout this whole process.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

In recent days, we have been directly or indirectly experiencing, utilising, applying many electronic devices to simplify and enhance the quality of our life-style. Most of these electronic devices are computers, tablets, mobile phones, IoT devices, sensors etc. In many cases, the control and communication of such devices are supporting very important and mission critical tasks. These complex workaround are realised by hardware, software and communication infrastructures. In recent years, real-time operating systems (RTOS) are being deployed in embedded systems that requires deadline sensitive task completion feature.

This project examines general real-time operating systems, their functionalities, problems faced, proposed and future work needed to be done to improve and enhance the performance and accuracy of the system.

1.2 Background

The application of Internet of Things (IoT) and embedded systems technologies by using wired/wireless interacting devices in our daily life activities have risen significantly (Musaddiq et al. 2018). These low end devices perform as a miniature form of computer that utilise varieties of CPUs, power and memory models based on specific requirement. As a result, traditional general purpose operating system (GPOS) such as Linux are too large to port and manage on these low end devices. Thus, an Operating System (OS) with lighter footprint and real-time functionality is needed for accurate and efficient use of resources that also offers greater control and operational customisation

to applications.

1.3 Operating Systems

A computer system comprises of hardware, software, data and people. Taking advantage of the power of hardware, Operating System (OS) is used. The operating system is a program that acts as an interface between the computer user and computer hardware, and controls the execution of programs. It also manages all of the software and hardware on the computer system and performs basic tasks such as file, memory, process management, handling peripheral devices etc.

Mostly general purpose OS such as MS Windows, MacOs, Linux etc are being used on desktop computers, tablets etc. There are mobile operating systems such as Android, iOS, Windows mobile etc which are being used in mobile devices.

In addition, there are specialised operating systems used in specialised/dedicated computer systems, IoT devices, embedded devices etc. These embedded operating systems even have to perform well on many constraints such as low power, low end processor, low memory, weather condition, mission critical systems. Real Time Operating system (RTOS) is a special implementation of embedded operating system where the task execution deadline is the most important factor.

1.4 Real time Systems

Real Time Operating System (RTOS) manages time constrained tasks associated with deadlines while maintaining small footprint and portability. They ensure real time application's tasks are executed within the required time constraints, contrasting to the general purpose operating system where task are prioritised to execute in terms of balanced system performance and uncertain of timed executions.

Wang (2017) categorises real time systems mainly into Hard and Soft real- time systems. Systems where an execution of a real time task is deadline critical and if missed would cause whole system failure are called Hard Real Time Systems. On the other hand systems which executed most but miss some executions before quoted deadline and resulting in no system failure but rather just degraded performance are called Soft Real Time Systems. Example of hard real time tasks maybe in avionics where measurements are absolutely necessary to perform adjustments in split seconds

Many RTOSes (commercial, proprietary and open source versions) have been developed and are available for use based on the devices, requirements and conditions. FreeRTOS (*FreeRTOS* 2022) is an open source real-time OS widely used in micro controllers (MCU) and small micro processors and is available under MIT license as well as commercial license. It consists of a small micro

kernel architecture providing very limited functionality like threading, mutexes, semaphores, and software timers (Hahm et al. 2016)

1.5 Problems

In RTOSes, the time sensitive task execution is the most important factor and has to meet all the deadlines without fail to avoid system failure. The task execution is managed by the task scheduling algorithm employed by the operating system.

The main purpose of scheduling algorithm is to address maximum CPU utilisation, fair allocation of CPU, maximum throughput, minimum response time. To meet the requirement of hard and soft real-time systems, the tasks scheduling algorithm is of paramount importance for any RTOSes.

In FreeRTOS, a static scheduler is used where tasks are given a fixed priority. This may result on missing the most important task deadline. This problem should be addressed by utilising alternative scheduling algorithm

1.6 Aims of the project

This project aims to evaluate the task scheduling algorithms in FreeRTOS and implement an alternative scheduling algorithm based on Earliest Deadline First (EDF). The EDF scheduling algorithm is proposed to support dynamic task scheduling that will address maximum CPU utilisation, fair allocation of CPU, maximum throughput and reduced latency.

For the proposed algorithm, an implementation and integration description supported by relevant tests on FreeRTOS are presented.

1.7 Rational for the selection of the project topic

Being a student of BSc computer science, I have got the opportunity to learn and understand major areas in computer systems such as operating systems, database, computer networks, programming languages, software engineering etc. Embedded systems and real time operating systems are very specific area in computer systems and brings various further opportunities because of wide use of IoT, embedded devices and mobile devices. This appears to me to be a positive scope in the business and employment.

Moreover, developing a project on software algorithm to be implemented in an embedded systems involves most of the major areas of my skill so far gained. I find this research project as a great

opportunity to apply my prior theoretical understanding of various subjects in real world systems to make myself more confident. Developing such a system would fully substantiate my work and fully demonstrate the skills I have gained earlier. In addition, my immense interest in the low level IT systems integration, communication and deployment to work in this project will be of great experience for me.

1.8 Project Environment

In order to achieve the aim of the project, we will be utilising the following environment/settings.

Development/Working PC Host:

iMac(retina 5k, 27 inch) with Quad core 3.4GHz Intel Core i5 and MacOS STM32CubeIDE (GCC, ST-Link, ARM Cortex M4 firmware) tool-chains

Real Time Operating System:

FreeRTOS LTS 202012.04

Target Board :

STM32F413H-DISCO Evaluation board
ARM Cortex M4 (STM32F413ZHT6 MCU)
1.5 Mbytes of flash memory
320 Kbytes of SRAM
240 x 240 pixel LCD
Integrated WiFi module
USB OTG FS

1.9 Contributions

The project aim has been achieved by implementing the EDF scheduling algorithm. The FreeRTOS port has been performed by enabling/disabling the default scheduling algorithm during the compilation phase on the FreeRTOS customisation configuration file (FreeRTOSConfig.h).

The STM32CubeIDE along with required packages and firmwares enabled to upload the compiled HEX file in to the MCU flash memory. Multiple tasks with various priorities using the FreeRTOS's API call were handled during the testing and results were observed as expected.

1.10 Summary

Real time operating systems play vital role in automating, interconnecting, communicating and reacting based on environmental parameters such as sensor fed events or natural events. Simplicity, manageability, security and efficiency, small footprint are some of the qualities preferred in RTOS. The main quality of FreeRTOS fixed priority (static) scheduler is its simplicity that adequately addresses the periodic tasks.

To addresses the problems risen by static scheduling in FreeRTOS, a compile time plug-play implementation of EDF scheduling algorithm has been worked out. The tests comprehends how the implemented dynamic scheduling algorithm (EDF) conforms to the expectation.

In spite of the periodic tasks handling currently achieved, sporadic tasks handling needs to be addressed in future work.

Chapter 2

Literature Review

2.1 Overview

In this chapter, literature review of embedded systems, application areas, specific requirement for using embedded systems and potential opportunity of such systems in our day to day life. Moreover, the importance of real time operating systems and the constraints that need to be addressed using RTOS is discussed. Also, the how task management is handled by FreeRTOS in real application by calling the FreeRTOS API function is reviewed.

2.2 Embedded Systems

Xiao (2018) describes an embedded system as a small-sale computer system, that is part of a machine or a larger electrical/mechanical system and is often designed to perform certain dedicated tasks and often a real-time system. Embedded systems also often need to be small in size, low in cost, and have low power consumption. Also, Siewert & Pratt (2015) describes that to successfully implement real-time services in a system providing embedded functions, resource analysis must be completed to ensure that these services are not only are functionally correct but also produce output on time and with high reliability and availability.

Siewert & Pratt (2015) emphasises that real-time embedded systems must provide deterministic behaviour and often have more rigorous time and safety-critical system requirements compared to general purpose desktop computing system.

Xiao (2018) also highlights the importance of embedded systems as they are getting increasingly used in many daily appliances such as digital watches, cameras, microwave ovens, washing ma-

chine, TVs and cars.

2.3 Real Time Operating Systems

Real Time Operating System (RTOS) manages time constrained tasks associated with deadlines while maintaining small footprint and portability. They ensure real time application's tasks are executed within the required time constraints, contrasting to the general purpose operating system where task are prioritised to execute in terms of balanced system performance and uncertain of timed executions.

Wang (2017) categorises real time systems mainly into Hard and Soft real- time systems. Systems where an execution of a real time task is deadline critical and if missed would cause whole system failure are called Hard Real Time Systems. On the other hand systems which executed most but miss some executions before quoted deadline and resulting in no system failure but rather just degraded performance are called Soft Real Time Systems. Example of hard real time tasks maybe in avionics where measurements are absolutely necessary to perform adjustments in split seconds. Examples of soft real time task may include a keyboard key press being missed and not registered in the screen. The RTOS must address the above described cases of real-time systems.

Many RTOSes (commercial, proprietary and open source versions) have been developed and are available for use based on the devices, requirements and conditions. FreeRTOS (*FreeRTOS* 2022) is an open source real-time OS widely used in micro controllers (MCU) and small micro processors and is available under MIT license as well as commercial license. It consists of a small micro kernel architecture providing very limited functionality like threading, mutexes, semaphores, and software timers. (Hahm et al. 2016)

A typical embedded system solves a complex problem by decomposing it into a number of smaller, simpler pieces called tasks that work together in an organised way and such a system is called multitasking system. Several important aspects of a multitasking design include exchanging/sharing data between tasks, synchronising tasks, scheduling tasks execution and sharing resources among the tasks. The piece of software that provides the required coordination is called an operating system. When the control must ensure that task execution satisfies a set of specified time constraints, the operating system is called a real-time operating system (RTOS) (Peckol 2019).

Carraro (2016) explains that a real-time application is normally composed of multiple tasks with different levels of criticality and formally defines real-time system with example as :

Considering a system consisting of a set of tasks, $T = \tau_1, \tau_2, ..., \tau_n$, where the finishing time of each task $\tau_i T$ is F_i. The system is said to be real-time if there exists at least one task $\tau_i T$, which falls into one of the following categories

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

Buttazzo (2011) illustrates that a typical timing constraint on a task is a deadline, which represents the time before which a process should complete its execution without causing any damage to the system. Moreover, Buttazzo (2011) categorises the real time tasks depending on the consequences of a missed deadline as :

Hard : A real-time task is said to be hard if missing its deadline may cause catastrophic consequences on the system under control.

Firm: A real-time task is said to be firm if missing its deadline does not cause any damage to the system, but the output has no value.

Soft: A real-time task is said to be soft if missing its deadline has still some utility for the system, although causing a performance degradation.

2.4 FreeRTOS

FreeRTOS is a real-time operating system kernel for embedded devices. FreeRTOS has been ported to various microcontroller platforms and provides methods for multiple threads or tasks, mutexes, semaphores, message queues and software timers. FreeRTOS has a minimal ROM, RAM and processing overhead and is available under MIT as well as commercial licenses.

FreeRTOS (*FreeRTOS* 2022) offers the features like the real-time scheduling functionality, inter-process communication, timing and synchronisation primitives.

FreeRTOS is static in nature prioritising tasks compared to its duration of execution (lower the execution duration means higher the priority)

FreeRTOS is a real-time kernel (or real-time scheduler) on top of which embedded applications can be built to meet their hard real-time requirements. It allows applications to be organised as a collection of independent threads of execution. On a processor that has only one core, only a single thread can be executing at any one time. The kernel decides which thread should be executing by examining the priority assigned to each thread by the application designer. (Barry 2016)

2.5 Task implementation in FreeRTOS

Tasks are implemented as C functions which must return void and take void parameter(Barry 2016). The prototype is :

```
void vATaskFunction( void *pvParameters );
```

A single task function definition can be used to create any number of tasks - each created task being a separate execution instance, with it own stack and its own copy of any stack variables defined within the task itself.

2.5.1 Task Structure

Carraro (2016) illustrates FreeRTOS task as a small program with a priority assigned and each task executes within its own context. At each instant (software timer tick), the RTOS selects the task that will be executed based on the priority. Technically, FreeRTOS maintains the task in a data structure called Task Control Block (TCB) as below:

```
typedef struct tskTaskControlBlock
{
    volatile StackType_t *pxTopOfStack;
    Listitem_t xGenericListItem;
    ListItem_t xEventListItem;
    UBaseType_t uxPriority;
    StackType_t *pxStack;
    char pcTaskName[configMAX_TASK_NAME_LEN];
    StackType_t *pxEndOfStack;
    UBaseType_t uxBasePriority;
} tskTCB;
```

The TCB task structure above is designed to hold general information about the task such as current, top and end stack pointers, overflow checking, task priority.

2.5.2 Task States

Barry (2016) outlines that a task in FreeRTOS can be in any of the four states of Running, Ready, Blocked and Suspended as shown in the Figure 2.1.

(Enrico Carraro, 2016) exlains the task states as below:

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

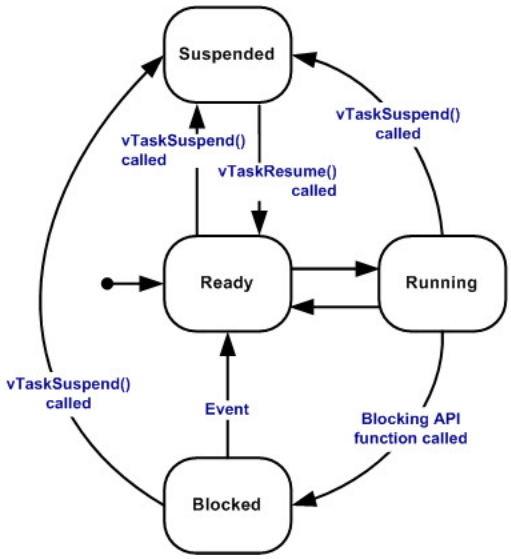

Figure 2.1: States of a task in FreeRTOS (Barry 2016)

2.5.3 Task Priorities

Barry (2016) shows that each task is assigned a priority from 0 to (`configMAX_PRIORITIES - 1`), where `configMAX_PRIORITIES` is defined within FreeRTOSConfig.h. Low priority numbers denote low priority tasks. The idle task has priority zero (`tskIDLE_PRIORITY`).

The FreeRTOS scheduler ensures that tasks in the Ready or Running states are always given priority over jobs with lower priorities that are also in the Ready state. To put it another way, the work that is placed in the Running state is always the most important task that can run.

2.5.4 Task management API

2.5.4.1 Creating a new task

Ibrahim (2020) illustrates to create an instance of a new task as :

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

Ibrahim (2020) describes the task creation process as : A task can be created before or after the scheduler is started. When a task is created, the required RAM is automatically allocated from the FreeRTOS heap. A new created task is initially in the READY state, but possibly moves to the RUNNING state if there are no other higher priority tasks that are able to run, or if the currently running task has the same priority as this task and the running task gives up the CPU.

2.5.4.2 Task delay

Ibrahim (2020) illustrates to delay a task for a specified time by blocking the task temporarily:

```
   vTaskDelay(TickType_t xTicksToDelay);
```

This function blocks the calling task for a fixed number of tick interrupts.

2.6 EDF in RTOS

Over the years since RTOS was first introduced in 1990s, various algorithms and methodologies have been proposed and published to improve the predictability of real time systems. All these algorithms and methodologies are derived adhering to set of defined concepts. First and important concept is defined as process for also known as tasks. A process is a computation that is executed by the CPU in a sequential fashion. Process and tasks are synonyms in our context of Real time operating systems as almost all real time process are not complex or usually only composed of one task. Scheduling algorithm assign tasks to be executed on the CPU using scheduling policies.

Scheduling policy assign tasks to CPU according to predefined criterion, Scheduling algorithm determines the order at any given time in which the tasks are executed.

Many different methods and implementation has been propose to port an EDF scheduling algorithm on RTOS. Páez et al. (2015) presents his EDF in user space for mixed critical systems containing two sub sets of tasks, critical and periodic tasks, and no real time tasks. Páez et al. (2015) states implementation of such a scheduling algorithm like EDF requires extensive knowledge of the kernel and the RTOS structures. Furthermore, they questions the loss of safety certifications, robustness and reliability when modifying the kernel to adjust the new scheduling scheme. They also along with proving safety and reliability, user can have choices and quickly modify needed changes to suit their need of scheduling. Similarly to Páez et al. (2015), Kase et al. (2016) also expresses the need to implement scheduling algorithm at user spaces to allow user to have options. They propose a user level scheduling library "ESFree" primarily aiming for user flexibility, giving user the option to explore and use advance theory with low development time, closing the gap between taught theory and practicality. In their scheduling library, they introduced two different implementation of RMS and EDFs, the difference being the latter implementation being optimised using trace macro features in FreeRTOS to get around constant context switching. Although user level provides for quick and easy development, the drawbacks gained from purely using various API calls communicate with kernel to perform context switching and queues management lead to large overhead which would take effect in the scheduling. This is observed in the evaluation conducted by Kase et al. (2016) on their "efficient EDF" from their scheduling library. The results shows context switching occurrence was was two or three times higher than compared to default FreeRtos and RMS schedulers. It is expected for EDF to have higher preemption than RMS but when implemented on kernel level, the rate of preemption is significantly less than two times unlike user space (Buttazzo 2005). Thus, implementation of such scheduling algorithm is not ideal in user space for performance.

Contrasting to Páez et al. (2015), Kase et al. (2016) others have considered implementing their scheduling algorithms on kernel level. Belagali et al. (2016) performs his dynamic priority scheduling analysis on popular ARM Cortex M4 implementing on FreeRTOS kernel similar to our implementation environment. ARM Cortex M4 is a uniprocessor system with low processing power and thus benefits from optimum schedulers. As stated earlier EDF is an optimum dynamic scheduler with schedule-ability criterion $U_i < 1$ but so is Least Slack Time (LST) scheduler (Leung 2004). They primary integrate LST as the scheduler but performs tests by assigning practical real time tasks against EDF and static scheduler. Results shows LST algorithms successfully scheduler all three job instances of two tasks provided, EDF scheduled 3 out of six total jobs and static only 2 jobs of the first tasks. The test conducted, however, was very weak considering only two tasks with 3 jobs with moderate utilisation per task. On the larger amount of task and jobs, EDF is expected perform better with lower overhead than LST as LST schedulers are known for greater preemp-

tions than EDF (Pelleh 2006). Conversely to Belagali et al. (2016), EDF on FreeRTOS have also been implemented on higher performing processors like Toma et al. (2018). They provide empirical evaluation for the schedulability of the multi-mode real-time tasks under fixed and dynamic priority scheduling algorithm implemented on FreeRTOS ported on Raspberry pi B+. Raspberry pi B+ features a Quad core CPU with speed of 700 MHz and RAM of 514MB. This is comparatively three times greater CPU speed then ARM Cortex M4 used by Belagali et al. (2016) and us. Higher processing power systems come at a cost of higher energy demand and in embedded systems energy is a resource that is very scares. Thus, although tasks can be scheduled and computed at faster rate, higher energy requirement is not a good trade off for systems that are meant to run indefinitely and autonomously. Though with higher CPU, task schedulability of EDF with synthetic tasks were observed to have 100% success rate compared to 50% of RM by Belagali et al. (2016), when schedulers were tasked to schedule realist multi-mode tasks, EDF showed to preform worse than RM by a significant margin. EDF was able to schedules all tasks with utilisation of only 10% but failed to schedule any tasks with utilisation higher than 50%. However RM scheduled all tasks with utilisation up to 40% and schedule fewer tasks up till 60%. Toma et al. suggests the reasoning for this unexpected results were due to the higher overhead of EDF and the distribution of tasks among the periods in the data set. Short period of real task set couples with workload of computing scheduling decisions deemed the jobs unscheduled. Even with EDF optimal schedulability, the overhead cost of EDF could not compete with RM. Therefore, there is a need to optimise EDF to make it comparative to RM and fully utilising its greater schedulability.

EDF and LLREF schedulers were implemented and verified by Carraro (2016). LLREF refers to Largest Local Remaining First Execution time (LLREF) scheduler, originally developed for multiprocessor systems. Although LLREF is intended for multiprocessor systems, it can also be used for uniprocessor by simply changing its parameters. Its implementation with the kernel however is not as straight forward as changing few parameters. Carraro implements on FreeRTOS to replace its default static priority scheduler, upgrading it's scheduling ability. The FreeRTOS is ported to ST STM32F429Discovery Board which uses ARM Cortex M4 CPU with 180MHz CPU speed for testing and verification of scheduling order. Scheduling capabilities of implemented EDF and LLREF were not testing or evaluated but rather the order of the scheduled tasks was verified. No synthetic tasks like the ones used in Kase et al. (2016), Páez et al. (2015), Toma et al. (2018) were generated but tasks were created manually to verify the tasks were correctly scheduled in the correct order using the "cheddar tool".

Very few have proposed ways to optimise EDF scheduling algorithm. It is usually due to the efficiency reasons, RM scheduling is preferred over EDF in the industry. Pathan (2016) suggested method to compose the ready queue based on linked list rather than priority queue normally used. Performing this data structure transformation allowed Pathan to achieve insertion and deletion from the ready queue at constant time. They suggested using a bit map with array of linked list for each

to efficiently traverse through this structure to get to Task Control Block (TCB) on the queue. Scheduling algorithms constantly remap the ready queue according to its policies to correctly scheduling the next tasks, especially with EDF as it is expected to have higher preemptive rate than its static counter parts Buttazzo (2005). Oliveira & Lima (2020) built upon Pathan's theory and implemented this Earliest Deadline First-Linked List (EDF-L) scheme into FreeRTOS as an alternative efficient scheduler to the native FreeRTOS. When the EDF-L was evaluated against Earliest Deadline First-MinHeap (EDF-H), EDF-L consistently observed lower overhead than EDF-H and concurred theoretical results gained by Pathan (2016).Results showed RMS outperformed EDF-H in scheduling more than 50 task set with 85% utilisation. They suggest the Min heap structure as the ready queue was a bottle neck for EDF even if theoretically it should have been able to schedule better than RMS. On the other hand EDF-L had on problems scheduling all task sets. However, this improved performance was only observed in all test conducted with huge number of tasks.Rather than Large set of tasks, smaller set of tasks with higher CPU utilisation per task set is more commonly observed in embedded systems utilising lightweight kernel like FreeRTOS.

2.7 Summary

The technological advancement in the portable and embedded devices brought the demand of having a stable and efficient real time operating system that could be portable across many hardware, having low cost and efficient. FreeRTOS is one of the most widely used RTOS in embedded devices. Moreover, in this chapter, the operation and implementation details of task in FreeRTOS has been reviewed. Various proposed and implemented EDF in RTOS were discussed. We have also seen that the task in FreeRTOS is implemented as C function and each task is managed in different context. Task Context and switching with respect to FreeRTOS will be discussed in next chapters.

Chapter 3

Analysis

3.1 Problem Setting

The main purpose of scheduling algorithm is to optimise maximum CPU utilisation, fair allocation of CPU, maximum throughput, minimum response time. As we have already discussed that the problem is caused by the missed task deadlines that result in catastrophic failures of the system. It has also been identified that such missed deadlines are because of the fixed task scheduling algorithm implemented in FreeRTOS.

In FreeRTOS, a static scheduler is used where tasks are given a fixed priority. This may result on missing the most important task deadline. This problem should be addressed by utilising alternative scheduling algorithm

Ibrahim (2020) brings outs the need for an RTOS describing RTOS is a program that manages system resources, schedules the execution of various tasks in a system, synchronises the execution of tasks, manages resource allocations, and provides inter-task communication and messaging between the tasks. Every RTOS consists of a kernel that provides the low-level functions, mainly the scheduling, task creation, inter-task communication, resource management, etc. Most complex RTOSs also provide file-handling services, disk read-write operations, interrupt servicing, network management, user management, etc.

The operation of any operating system is to execute given tasks correctly as stipulated from the tasks priority based on the criteria of scheduling algorithm used. The scheduling algorithm plays utmost role in driving the real time system. Advancements in scheduling algorithm result in a correlated advancement in better performing RTOS directly. Hence, various algorithms have been proposed to schedule real time tasks. Buttazzo (2011) presents the following three categories that

can even be combined for better results. (e.g. Preemptive-online scheduling, preemptive-dynamic, preemptive-online-dynamic etc.)

- Preemptive vs. Non-Preemptive scheduling
- Offline vs. Online scheduling
- Fixed(Static Priority) vs. Dynamic scheduling

Preemptive-online-fixed priority scheduling is primarily used by wide range of real time systems. The Rate Monotonic Scheduling (RMS) algorithm used in FreeRTOS is static in nature prioritising tasks compared to its duration of execution (lower the execution duration means higher the priority). RMS is widely used due its easy implementation, however, its incapability of managing tasks dynamically leads to poor performance under certain circumstances resulting into missed task deadlines. In contrast, Earliest Deadline First (EDF) is a dynamic scheduling algorithm prioritising tasks with earliest deadlines. Due to its dynamic nature, the algorithm evaluates priority after each execution iteration and rebuilds the ready and running queues with new priorities. This ensures the earliest deadline task (the most important task) is handled immediately by preempting already executing task.

In RMS, task priority are based upon inverse of their period: lower the task period, higher the priority. That means the priorities are precalculated, contrasting to EDF where priorities are calculated at run time based on tasks deadlines. Furthermore, Liu & Layland (1973) proved that for set of n periodic tasks with unique period, a feasible RM schedule that will always meet task deadline exists if CPU utilisation U is below a specific bound as shown by the following scheduling condition:

$$U = \sum_{i=1}^{n} U_i = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right) \tag{3.1}$$

where C_i is the computation time, T_i is the period of task and n is the number of tasks to be scheduled. When the number of tasks, n, tends to infinity , the utilisation converges to ln(2) $\approx$ 0.6931.. or 69.32%; implying no set of tasks can be scheduled if the CPU utilisation is > 69.32%. EDF however, has a CPU utilisation bound of 100% and thus can guarantee task deadline with higher CPU utilisation than RMS. Like RMS, EDF is also proved to be optimal for preemptive uniprocessor system. EDF has the following scheduling condition:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{3.2}$$

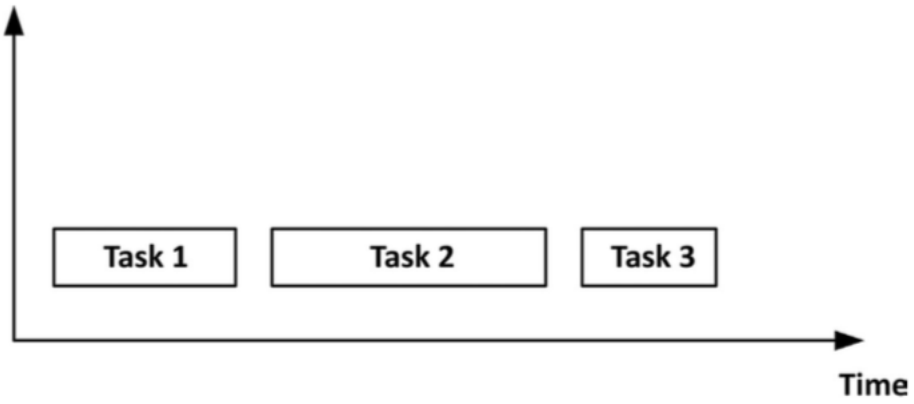

Figure 3.1: Cooperative scheduling (Ibrahim 2020)

From this, it is understandable that RMS posses issues achieving the objective and performance of real-time systems, thus the exploration of other scheduling algorithms such as EDF gains momentum.

FreeRTOS is an example of one such RTOS where native scheduler is based on Fixed Priority Scheduling(FPS). Many have proposed of implementing different FPS scheduling algorithms for FreeRTOS but only few have proposed implementing a Dynamic Priority Scheduler on FreeRTOS. Various implementations have also been proposed in user space rather than kernel space but that has its added overhead. Alternative methods for priority queue has also been researched by Pathan (2016) and implemented by Oliveira & Lima (2020).

3.2 Task Scheduling in FreeRTOS

Ibrahim (2020) illustrates that although there are many variations of scheduling algorithms in use today, three most commonly used algorithms are:

- Co-operative scheduling (Also known as non-preemptive scheduling)

  This is the simplest algorithm where tasks voluntarily give up the CPU resource when they have nothing to do and in idle state. The disadvantage in this type of implementation is that certain tasks can use extensive CPU thus not allowing other important tasks to execute when needed.

  Co-operative scheduling is preferred to schedule only non time-critical tasks.

In the Figure 3.1 we can see that Task 2 has utilised excessive CPU time leaving Task 3 to wait and miss the deadline if any,

One of the very simple way of implementing co-operative scheduling is using a conditional statement inside an infinite loop as below.

```
    Task1()
{
Task 1 code
}

Task2()
{
Task 2 code
}

Task3()
{
Task 3 code
}

nxt=1;
while(1) {
   switch(nxt)
   {
   case 1:
       Task1();
       nxt=2;
       break;
   case 2:
        Task2();
       nxt=3;
        break;
   case 3:
      Task3();
      nxt=1;
      break;
   }
}
```

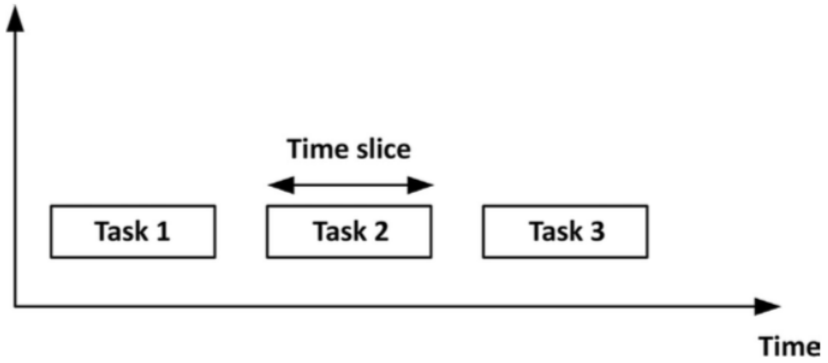

Figure 3.2: Round Robin Scheduling (Ibrahim 2020)

- Round-robin scheduling

  In round robin scheduling, we distribute the CPU time equally to all the tasks. Tasks are maintained in a circular queue and when a task's allocated CPU time expires, the task is removed.

  As show in the Figure 3.2, the time slice is equally distributed among three tasks. Round-robin scheduling has advantages like easy to implement, easy to compute response time, easy distribution of CPU share. However, the this scheduling is not suitable for real-time systems where tasks have different processing requirements.

- Preemptive scheduling

  Ibrahim (2020) explains that Preemptive scheduling is the most commonly used scheduling algorithm in real-time systems. Here, the tasks are prioritised and the task with the highest priority among all other tasks gets the CPU time more comparatively. As shown in the figure, the CPU time allocated to Task 1, Task 2, Task 3 are different.

  The priority in a preemptive scheduler can be static or dynamic. In a static priority system tasks used the same priority all the time. In a dynamic priority-based scheduler, the priority of the tasks can change during their courses of execution.

3.3 Context Switch

In multitasking, many tasks are sharing the CPU time based on the scheduling algorithm. A context switch is a procedure that a computer's CPU follows to change from one task to another while

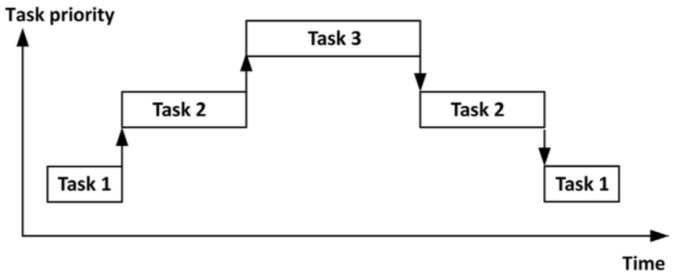

Figure 3.3: Preemptive Scheduling (Ibrahim 2020)

storing the states of a process/thread, so that it can be restored and resumes execution later. The process of saving the context of a task being suspended and restoring the context of a task being resumed is called context switching.

In FreeRTOS, As a task executes it utilises the processor/microcontroller registers and accesses memory resources (RAM, ROM etc). The task execution context comprises of these resources such as the processor registers, stack, etc.

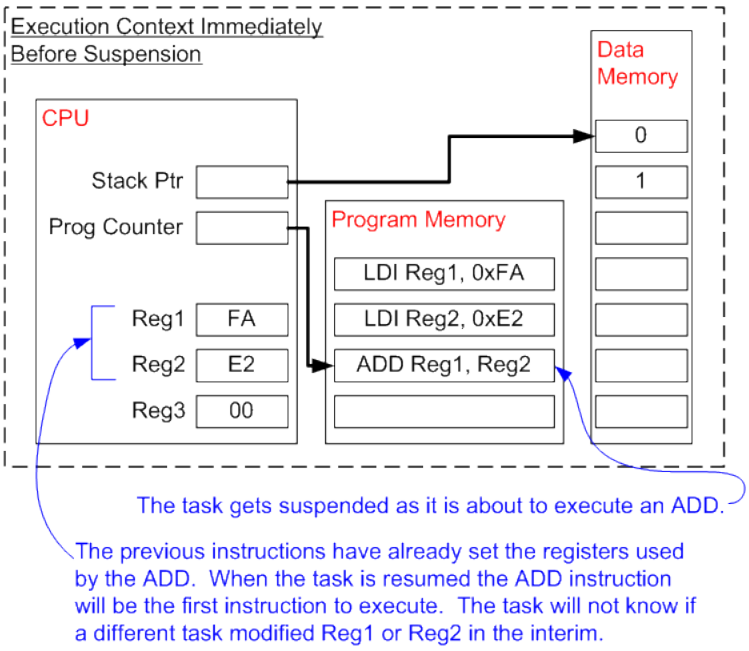

Figure 3.4: Context Switching (Barry 2016)

Barry (2016) illustrates that a task is a sequential piece of code that gets suspended and resumed by

the kernel independently. The Figure 3.4 represents a task execution context where the previously running task's context data has been saved and new task's context data is loaded in the registers. The context switching let the tasks unaware that a different task already modified the registers. In FreeRTOS, the kernel is responsible for ensuring that context switching happens smoothly.

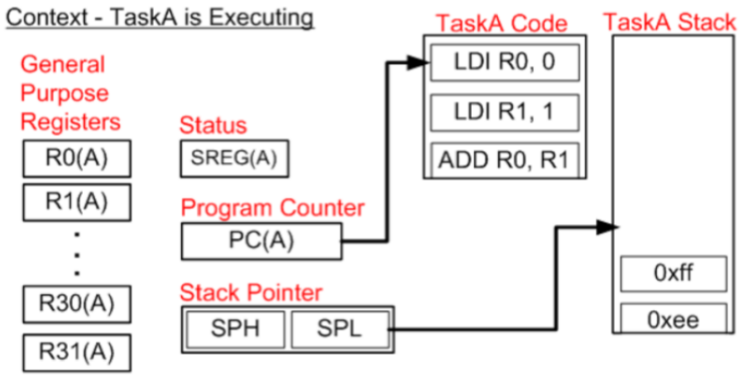

Figure 3.5: Task executing context switch (Barry 2016)

Carraro (2016) also illustrates the task stack after saving the context as presented on the Figure 3.5. As seen on the figure above, FreeRTOS implements the `portSAVE_CONTEXT()` and `portRESTORE_CONTEXT()` macros for saving and restoring the context.

In FreeRTOS, `postSAVE_CONTEXT()` and `portRESTORE_CONTEXT()` are defined as :

```
#define portSAVE_CONTEXT()
      asm volatile (
        "push r0                    nt" (1)
        "in   r0, __SREG__          nt" (2)
        "cli                        nt" (3)
        "push r0                    nt" (4)
        "push r1                    nt" (5)
        "clr  r1                    nt" (6)
        "push r2                    nt" (7)
        "push r3                    nt"
        "push r4                    nt"
        "push r5                    nt"
          :
          :
          :
         "push r30                   nt"
```

```
        "push r31                nt"
        "lds  r26, pxCurrentTCB  nt" (8)
        "lds  r27, pxCurrentTCB + 1 nt" (9)
        "in   r0, __SP_L__       nt" (10)
        "st   x+, r0             nt" (11)
        "in   r0, __SP_H__       nt" (12)
        "st   x+, r0             nt" (13)
      );


#define portRESTORE_CONTEXT()
      asm volatile (
        "lds r26, pxCurrentTCB   nt" (1)
        "lds r27, pxCurrentTCB + 1 nt" (2)
        "ld  r28, x+             nt"
        "out __SP_L__, r28       nt" (3)
        "ld  r29, x+             nt"
        "out __SP_H__, r29       nt" (4)
        "pop r31                 nt"
        "pop r30                 nt"
          :
          :
          :
         "pop r1                  nt"
        "pop r0                  nt" (5)
        "out __SREG__, r0        nt" (6)
        "pop r0                  nt" (7)
      );
```

3.4 Tick System

Software timers are important parts of any real-time multitasking operating system. The timers are used in tasks to schedule the execution of a function at a time in the future, or periodically with a fixed frequency (Ibrahim 2020).

The system tick is the time unit that OS timers and delays are based on and are typically be in the order of 1ms to 100ms. The system tick is a scheduling event causing the scheduler to run and may cause a context switch. Lowering the tick period requires system overhead significantly. So there is a trade off between timer tick resolution and CPU overhead. The timer resolution is configured

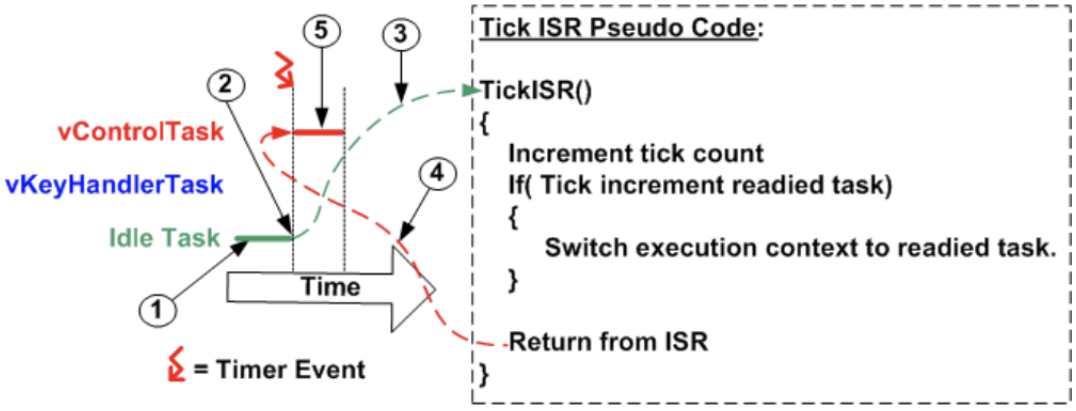

Figure 3.6: System Tick ISR (Barry 2016)

based on the application requirement.

In FreeRTOS, the system tick is implemented using a tick count variable. The tick interrupt's Interrupt Service Routine (ISR) call increments the tick count. This allows the kernel to measure time to a predefined timer interrupt frequency (Carraro 2016).

Barry (2016) describes the system tick and the resulting events/actions. Each time the tick count is incremented, the real time kernel must check to see if it is now time to unblock or wake a sleeping task. In case, the task woken during the tick ISR have a higher priority than that of the interrupted task then the tick ISR should return to the newly woken task. This effectively interrupting one task but returning to another. This situation is explained as below:

Referring to the numbers in the diagram above:

At (1) the RTOS idle task is executing.

At (2) the RTOS tick occurs, and control transfers to the tick ISR (3).

The RTOS tick ISR makes `vControlTask` ready to run, and as `vControlTask` has a higher priority than the RTOS idle task, switches the context to that of `vControlTask`.

As the execution context is now that of `vControlTask`, exiting the ISR (4) returns control to `vControlTask`, which starts executing (5).

A context switch occurring in this way is said to be Preemptive, as the interrupted task is preempted without suspending itself voluntarily.

FreeRTOS implements RTOS tick interrupt as below:

```
    /* Interrupt service routine for the RTOS tick. */
void SIG_OUTPUT_COMPARE1A( void )
{
    vPortYieldFromTick();
    asm volatile ( "reti" );
}
/*-----------------------------------------------------*/
 void vPortYieldFromTick( void )
{
    portSAVE_CONTEXT();
    vTaskIncrementTick();
    vTaskSwitchContext();
    portRESTORE_CONTEXT();
    asm volatile ( "ret" );
}
/*-----------------------------------------------------*/
```

As we can see from the above code, the first statement in `vPortYieldFromTick()` function is to save the context by calling `portSAVE_CONTEXT()` macro, then increments the tick counter. Before returning from the function, the context is restored by calling `portRESTORE_CONTEXT()` macro.

3.5 Legal, Social, Ethical and Professional issues

The project mainly focusses on the performance and efficiency improvement of the emebedded systems, it has no direct consequences related to the legal, social, ethical and professional issues. However, indirectly a positive value in social, ethical and professional areas can be realized. As we will have an improved version of the devices/systems that will yield better/efficient result.

3.6 Summary

This chapter brings out the problems and difficulties of the real-time systems. As the real-time systems are mostly in embedded devices and highly constrained settings (low power, low CPU, low memory etc), even a slight improvement in the performance make a great deal in practice. Here we discussed constraints and room for improvement in the scheduling algorithms by adopting EDF scheduling algorithm.

From FreeRTOS's implementation point of view, we have analysed and dissected some of the important aspects such as task scheduling, context switching, software timers and ticking.

Chapter 4

Requirements Specification

4.1 Earliest Deadline First Algorithm

Earliest Deadline First (EDF) is a dynamic priority scheduling algorith for real time systems. EDF selects a task based on its deadline that means it gives high priority to the tasks that has earliest deadline. In other words, priority of a task is inversly proportional to its absolute deadline. (Masoud 2021)

EDF maintains a dynamic priority based preemptive scheduling policy. That means, the priority of a task can change during its execution and the processing of any task is interrupted by a request for any higher priority task.

EDF is a task scheduling algorithm that does not asume periodicity of tasks. That means, EDF is independent of period of task (repetition of task every n time interval) and therefore can be used to schedule aperiodic tasks as well. If two tasks have the same absolute deadline then EDF selects one of them randomly.(Masoud 2021)

Masoud (2021) depicts EDF with example as below:

Task	Release time(r_i)	Execution time(C_i)	Deadline time(D_i)	Period(T_i)
T_1	0	1	4	4
T_2	0	2	6	6
T_3	0	3	8	8

Table 4.1: Example Task specification

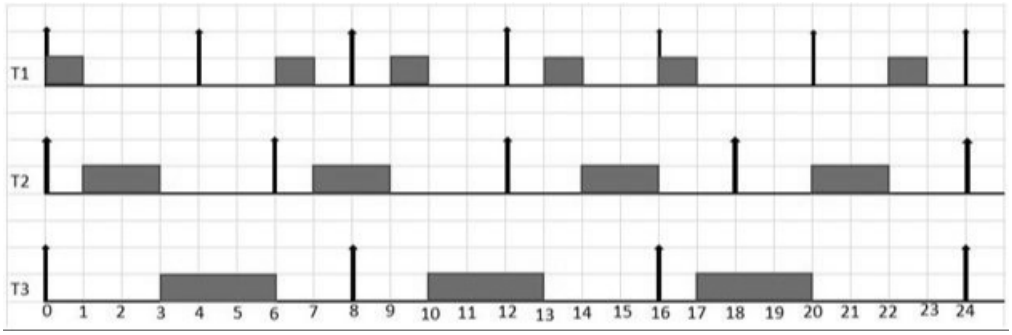

Figure 4.1: Scheduled task using EDF (Masoud 2021)

EDF can schedule any task if The processor Utilisation (U) is derived as below.

Carraro (2016) make apparent about the EDF scheduling feasibility as "A task set of periodic tasks is schedulable by EDF if any only if" Equation 3.2 is satisfied.

Processor Utilization(U) = $1/4 + 2/6 + 3/8 = 0.25 + 0.333 + 0.375 = 0.95 = 95\%$

In this example, as processor utilisation is less than 1 or less than 100% so task set is surely schedulable by EDF.

EDF scheduling as depicted in the figure above is described as below where priorities are decided based on absolute deadlines:

Case at t=0 :

- All the tasks are released.
- T1 gets executed because T1 has higher priority (T1 deadline is 4, T2 deadline is 6 and T3 deadline is 8).

Case at t=1 :

- T2 gets executed. (T1 already completed because of execution time of 1)

Case at t=3 :

- T3 gets executed (T2 already completed because of execution time is 2)

Case at t=4 :

- T1 comes in the system.

- T1 and T3 has same deadlines so randomly either T1 or T3 will get executed. In this example, T3 is chosen.

Case at t=6 :

- T2 released
- now deadline of T1 is earliest than T2 so it starts execution and after that T2 begins to execute.

Case at t=8 :

- T1 and T2 have same deadlines i.e. t=16, so ties are broken randomly and T2 continues its execution and then T1 completes.

Case at t=12 :

- T1 and T2 come in the system simultaneously so by comparing absolute deadlines, T1 and T2 has same deadlines therefore ties broken randomly and we continue to execute T3

Case at t=13 :

- T1 begins it execution and ends at t=14. Now T2 is the only task in the system so it completes it execution.

Case at t=16 :

- T1 and T2 are released together, priorities are decided according to absolute deadlines so T1 execute first as its deadline is t=20 and T3's deadline is t=24
- After T1 completion T3 starts and reaches at t=17 where T2 comes in the system now by deadline comparison both have same deadline t=24 so ties broken randomly and we choose to continue to execute T3

Case at t=20 :

- Both T1 and T2 are in the system and both have same deadline t=24 so again ties broken randomly and T2 executes. After that T1 completes it execution.

4.1.1 Advantages and disadvantages of Earliest Deadline First (EDF) over Rate Monotonic Scheduling (RMS)

Masoud (2021) outlines the advantages and disadvantages of EDF over RMS as below.

Advantages:

- Priorities are defined dynamically so no need to define offline in advance.
- Less context switching
- Processor utilisation can be up to 100%

Disadvantages

- Because of changing response time of tasks, it is less predictable
- Less control over the execution
- High execution overheads

4.2 Functional Requirements

- An implementation of EDF scheduling algoirithm by customizing the FreeRTOS kernel.
- Demo application to test and verify the EDF algorithm implementation.
- Supporting Log files generated by the system showing the working of EDF algorithm.

4.3 Non-functional Requirements

- Coding style guide
- Variable naming conventions
- Functions naming conventions
- Macros naming conventions
- Data types (`TickType_t`, `BaseType_t`, `UBaseType_t`, `StackType_t`)
- Code customization allowing porting to multiple hardwares.

4.4 Summary

This chapter elaborates the EDF scheduling algorithm with example. The advantages and disadvantages of EDF over RMS are discussed. The function and non functional requirements of the project are identified and will be achieving in the design and implementation phase further.

Chapter 5

Design and Implementation

5.1 Overview

As previously discussed, FreeRTOS uses a static priority policy based scheduler. The aim of this chapter is to present an implementation of an EDF scheduler by customising the FreeRTOS codebase (data structures and project logic).

From conceptual point of view, the EDF implementation can be done by creating a new READY List which will contain the tasks ordered by increasing deadline time, where positions in the list represent the task priorities. The highest priority tasks are at the top and the lowest priority tasks are at the bottom of the list.

Task	Deadline
T_1	3
T_2	5
T_3	9
T_4	12
..	..
IDLE	100

Table 5.1: Tasks ordered by deadline

In the Table 5.1, the head of the list contains task T1 that is closest with the deadline. The other internal functions and macros in the source codebase need to be modified to handle (add, remove, update) the new Ready list.

5.2 Earliest Deadline First Scheduling

Kumar (2020) describes Earliest Deadline First (EDF) is a dynamic priority scheduling algorithm used in real-time systems and can be used for both static and dynamic real-time scheduling. Absolute deadline is the key factor on assigning priorities to the task. The priorities are assigned and cab be changed dynamically. EDF is very efficient as compared to other scheduling algorithms in real-time systems and make can make the CPU utilisation to about 100% while still guaranteeing the deadlines of all the tasks. In EDF, if any other periodic instance with an earlier deadline is ready for execution and becomes active then any running task can be preempted. In this project, we are customising FreeRTOS to incorporate EDF scheduling algorithm.

5.3 Implementation in FreeRTOS

This section brings out the implementation details of the proposed EDF scheduler. Portion of the project codebase architecture and example code will be discussed here.

We plan to apply EDF algorithm for scheduling tasks in FreeRTOS. A task is represented by a task control block (TCB) and is maintained as a data structure in FreeRTOS.

5.3.1 Background

While customising the codebase to implement the EDF scheduler, it is essential to understand the important data structures and functions of FreeRTOS and their implementation files.

Ferreira et al. (2014) states that FreeRTOS is written mostly in C programming language, with a few assembler functions that take care of architecture-specific details. There are four main C files that represent the kernel of FreeRTOS. As shown in the figure below, the file ”tasks.c” implements most of scheduler functionalities, making use of the structure and functions defined in the file ”list.c”. The file ”queue.c” implements thread-safe queues that are used in inter-task communication and synchronisation. As FreeRTOS supports many different architectures; the directory ”portable” contains the architecture dependent code.

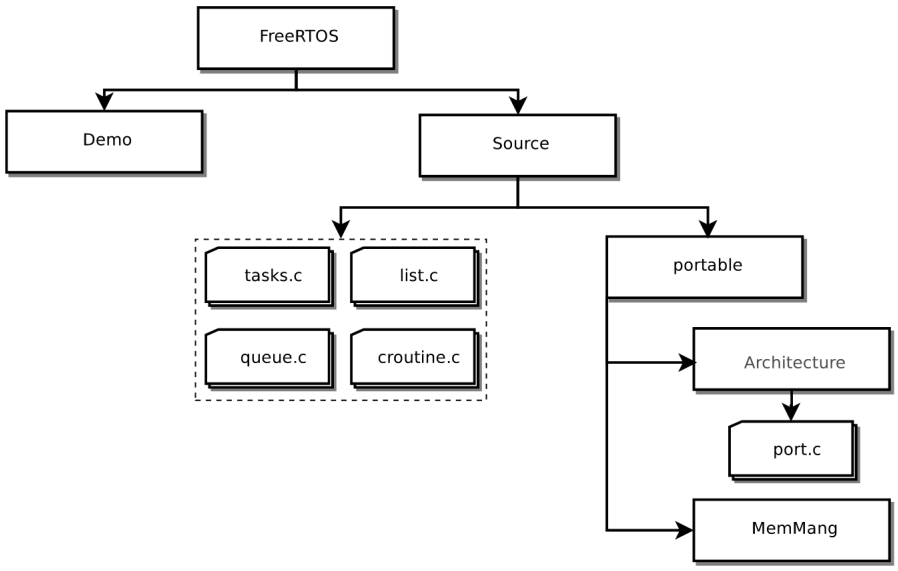

Figure 5.1: FreeRTOS file structure (Ferreira et al. 2014)

5.3.2 Data Structures

```
            /* Definition of the type of queue used by the scheduler. */
typedef struct xLIST
{
    listFIRST_LIST_INTEGRITY_CHECK_VALUE /*< Set to a known value if
        configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
    volatile UBaseType_t uxNumberOfItems;
    ListItem_t * configLIST_VOLATILE pxIndex; /*< Used to walk through the
        list. Points to the last item returned by a call to
        listGET_OWNER_OF_NEXT_ENTRY (). */
    MiniListItem_t xListEnd;            /*< List item that contains the
        maximum possible item value meaning it is always at the end of the list
        and is therefore used as a marker. */
    listSECOND_LIST_INTEGRITY_CHECK_VALUE /*< Set to a known value if
        configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
} List_t;
```

```
struct xLIST;

struct xLIST_ITEM
{
    listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE    /*< Set to a known value if
        configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
    configLIST_VOLATILE TickType_t xItemValue;   /*< The value being listed. In
        most cases this is used to sort the list in ascending order. */
    struct xLIST_ITEM * configLIST_VOLATILE pxNext; /*< Pointer to the next
        ListItem_t in the list. */
    struct xLIST_ITEM * configLIST_VOLATILE pxPrevious; /*< Pointer to the
        previous ListItem_t in the list. */
    void * pvOwner;                              /*< Pointer to the object
        (normally a TCB) that contains the list item. There is therefore a two
        way link between the object containing the list item and the list item
        itself. */
    struct xLIST * configLIST_VOLATILE pxContainer; /*< Pointer to the list in
        which this list item is placed (if any). */
    listSECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE   /*< Set to a known value if
        configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
};
typedef struct xLIST_ITEM ListItem_t;
```

5.3.3 FreeRTOS APIs and Macros

Some of the APIs/macros available in FreeRTOS are as below:

```
    void vListInitialise( List_t * const pxList ) PRIVILEGED_FUNCTION;
void vListInitialiseItem( ListItem_t * const pxItem ) PRIVILEGED_FUNCTION;
void vListInsert( List_t * const pxList,
                  ListItem_t * const pxNewListItem ) PRIVILEGED_FUNCTION;
void vListInsertEnd( List_t * const pxList,
                     ListItem_t * const pxNewListItem ) PRIVILEGED_FUNCTION;

UBaseType_t uxListRemove( ListItem_t * const pxItemToRemove )
    PRIVILEGED_FUNCTION;

#define listINSERT_END( pxList, pxNewListItem )
#define listREMOVE_ITEM( pxItemToRemove )
```

```
#define listCURRENT_LIST_LENGTH( pxList )
```

5.3.4 Configuration File (FreeRTOSConfig.h

FreeRTOS is customised using a configuration file called "FreeRTOSConfig.h". Every FreeRTOS application must have a FreeRTOSConfig.h header file in its pre-processor include path. FreeRTOSConfig.h tailors the RTOS kernel to the application being built. It is therefore specific to the application, not the RTOS, and should be located in an application directory, not in one of the RTOS kernel source code directories. (Barry 2016)

A typical truncated "FreeRTOSConfig.h" is presented below. We can have our own application-wide config settings defined in this file.

```
 #ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H

/* Here is a good place to include header files that are required across
your application. */
#include "something.h"

#define configUSE_PREEMPTION              1
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 0
#define configUSE_TICKLESS_IDLE           0
#define configCPU_CLOCK_HZ                60000000
#define configSYSTICK_CLOCK_HZ            1000000
#define configTICK_RATE_HZ                250
#define configMAX_PRIORITIES              5
#define configMINIMAL_STACK_SIZE          128
#define configMAX_TASK_NAME_LEN           16
#define configUSE_16_BIT_TICKS            0
#define configIDLE_SHOULD_YIELD           1
#define configUSE_TASK_NOTIFICATIONS      1
#define configTASK_NOTIFICATION_ARRAY_ENTRIES 3
#define configUSE_MUTEXES                 0
.....................................................
.....................................................
.....................................................
.....................................................

/* A header file that defines trace macro can be included here. */
```

```
#endif /* FREERTOS_CONFIG_H */
```

5.3.5 Implementation

5.3.5.1 Enabling/Disabling EDF scheduler

Following the FreeRTOS guideline, a global configuration variable "configUS_EDF" is added in to "FreeRTOSConfig.h" file. When the variable (configUSE_EDF) is set to 1, EDF scheduler is used, otherwise the OS uses its default scheduler.

```
#define configUSE_EDF 1 /* To use our custom EDF scheduler define with
    value 1 */
```

5.3.5.2 Managing New Ready List

The following set of actions are perfomed to accomodate the changes made to implement EDF scheduling. (Carraro 2016)

- A new Ready List is declared as below:

```
if (configUSE_EDF ==1 )
  PRIVILEGED_DATA static List_t xReadyTasksListEDF; /* Our Ready tasks
     ordered by their deadline. */
#endif
```

- Initialize the newly created "xReadyTasksListEDF" list in theprvInitialiseTaskLists() function that initialises all the task lists at the creation of task.

```
static void prvInitialiseTaskLists( void )
{
  #if (configUSE_EDF == 1 )
    vListInitialise( &xReadyTasksListEDF );
  #endif
}
```

- Updating prvAddTaskToReadyList() function that adds a task to the READY list to accommodate newly created "xReadyTasksListEDF".

```
#if (configUSE_EDF == 0 )

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

#else

  #define prvAddTaskToReadyList( pxTCB ) \
    vListInsert( &( xReadyTasksListEDF[ (pxTCB)->uxPriority ] ),
       &((pxTCB)->xGenericListItem))

#endif
```

Here, `vListInsert()` function inserts the task's TCB structure pointer into the custom "`xReadyTasksListEDF`".

- Updating the Task Control Block structure (`tskTaskControlBlock`)

  Because EDF scheduler requires every task in the Ready list has a dynamic deadline associated with it. So the `tskTaskControlBlock` structure must also hold the new data about the deadline of the task by introducing a new member variable on the `tskTaskControlBlock` structure.

```
#if (configUSE_EDF == 1 )

 TickType_t xTaskPeriod; /* the period in tick of the task */

#endif
```

  The deadline of any task in the `readyListd` is calculated as `(TASK_deadline = TICK_current + TASK_period)` and every task need to store its period value.

- Update `xTaskGenericCreate()` function

  `xTaskPeriodiCreate()` is created as a new version of `xTaskGenericCreate()`. This accepts an extra parameter `xTaskPeriod` variable in the TCB data structure. Also, before populating the new ready List, calling `prvAddTaskToReadyList()` initialises the next task deadline data on the `tskTaskControlBlock` structure.

```
BaseType_t xTaskPeriodicCreate( < regularParameters > , TickType_t
    period )
{
  ......
  pxNewTCB->xTaskPeriod = period;

  listSET_LIST_ITEM_VALUE( &((pxNewTCB)->xGenericListItem),
     (pxNewTCB)->xTaskPeriod + currentTick);

  prvAddTaskToReadyList( pxNewTCB ); /* initializes the next task
     deadline data member of pxNewTCB */

  ........

}
```

- Handling IDLE task management

  In FreeRTOS, at least one task needs to be executed at all time, because of this, the IDLE task management is mandatory. With default FreeRTOS scheduler, the IDLE task is initialised with lowest priority such that it would only be scheduled when no other task is in READY state.

  The modified `vTaskStartScheduler()` function initialises the IDLE task and inserts it into the Ready List with the initial deadline as high.

```
  void vTaskStartScheduler( void )
{
   BaseType_t xReturn;
   ......................

    #if (configUSE_EDF == 1)
    {

   tickType initIDLEPeriod = 100;

   xReturn = xTaskCreatePeriodic( prvIdleTask, "IDLE",
      tskIDLE_STACK_SIZE, (void * ) NULL, ( tskIDLE_PRIORITY |
      portPRIVILEGE_BIT ), NULL,
   initIDLEPeriod );
```

```
    ..................
    }

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

    #endif

}
```

Tasks ordered by deadline in READY List without IDLE task. (This causes problem in FreeRTOS)

Task	Deadline
T_1	3
T_2	5
T_3	9
T_4	12
..	..

Table 5.2: Tasks ordered by deadline without IDLE task

The result of the above code changes is depicted as below:

Task	Deadline
T_1	3
T_2	5
T_3	9
T_4	12
..	..
IDLE	100

Table 5.3: Tasks ordered by deadline with IDLE task

- Handling CONTEXT switch mechanism

  A context switch occurs when a suspended task with higher priority than the running task wakes up or running task is suspended. We must update the `*pxCurrentTCB` pointer data member in the TCB structure of the new running task by calling "`vTaskSwitchContext()`" function as noted below.

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

   #if (configUSE_EDF == 0)
   {
      taskSELECT_HIGHEST_PRIORITY_TASK();
   }
   #else
   {
      pxCurrentTCB = (TCB_t * )
          listGET_OWNER_OF_HEAD_ENTRY(&(xReadyTasksListEDF ) );
   }
   #endif

   .........
}
```

5.3.6 Scheduling Example

The code changes made above to accommodate EDF scheduler in FreeRTOS is supported with the following example where we will keep up the preemptive behaviour. The preemptive behaviour means that the operating system interrupts the running task and assign another task in running state.

We will consider the case how system tick event will make changes on the waiting List and ready List as shown below. For this example, assume that task T_1 and T_2 have a period of 5 and 10 respectively with the capacity (task completion duration) of 3.

Case at tick = 29

Task	Deadline
T_1	30
T_2	55
..	..
..	..

Table 5.4: Waiting List, Tick=29

Task	Deadline
T_5	40
..	..
..	..
IDLE	100

Table 5.5: Ready List, Tick=29

Case at tick = 30

Task	Deadline
T_2	55
..	..
..	..

Table 5.6: Waiting List, Tick=30

Task	Deadline
T_1	35
T_5	40
T_2	55
..	..
..	..
IDLE	100

Table 5.7: Ready List, Tick=30

When tick is 29, task T_5 is running and task T_1 is waiting next tick (tick 30) to wake up. When tick 30 occurs, task T_1 moves to the ReadyList with the deadline as:
`T1_deadline = Tick_current + T1_period = 30 + 5 = 35`

5.4 Test Application

Here, we develop a test application to test the EDF scheduler in FreeRTOS. The application creates two periodic tasks. Each task has its own period and capacity. Lets us assume the tasks are `Task_A` with `Period_A` and `Capacity_A` and `Task_B` with `Period_B` and `Capacity_B`. Here, period means in how many system tick, the task repeats. Similarly, capacity means how many system tick does the task need to complete the task (total running time in each cycle).

```
#include "main.h"
```

```
#define CAPACITY  4   /* assuming the capacity of both tasks are the same*/

#define PERIOD_A  5
#define PERIOD_B  9

void Task_A(void *pvParameters)
{

   volatile int ctr = CAPACITY;
   TickType_t xWakeTime =0 ;

   while(1)
   {

    TickType_t xTickCount = xTaskGetTickCount();
    TickType_t xTickNow;

      while(ctr !=0)
      {
         xTickNow = xTaskGetTickCount();
         if(xTickNow >xTickCount)
         {
            xTickCount=xTickNow;
            ctr--;          /* One tick completed */
         }
      }

      ctr= CAPACITY;

         vTaskDelayUntil( &xWakeTime, PERIOD_A); /* Wait till next cycle */

   }

}

void Task_B(void *pvParameters)
{

   volatile int ctr = CAPACITY;
   TickType_t xWakeTime =0 ;
```

```
   while(1)
   {

    TickType_t xTickCount = xTaskGetTickCount();
    TickType_t xTickNow;

      while(ctr !=0)
      {
         xTickNow = xTaskGetTickCount();
         if(xTickNow >xTickCount)
         {
            xTickCount=xTickNow;
            ctr--;          /* One tick completed */
         }
      }

      ctr= CAPACITY;
         vTaskDelayUntil( &xWakeTime, PERIOD_B); /* Wait till next cycle */


   }

}


void main()
{
   SystemInit();

   xTaskPeriodicCreate((TaskFunction_t) Task_A, (const char *) "Task -
       A",configMINIMAL_STACK_SIZE, NULL,1, NULL, PERIOD_A );
   xTaskPeriodicCreate((TaskFunction_t) Task_B, (const char *) "Task -
       B",configMINIMAL_STACK_SIZE, NULL,1, NULL, PERIOD_B );

   vTaskStartScheduler(); // Start the Scheduler

   while(1); // will never reach here
}
```

The test application consists of three functions ”main”, ”Task_A” and ”Task_B”. The main() function is the control entry point fr the application which calls SystemInit() for the initialisation of board. Then, two tasks are created using xTaskPeriodicCreate() assigning them with routine functions Task_A() and Task_B() and periods of PERIOD_A and PERIOD_B respectively. Followed by this vTaskStartScheduler() starts the FreeRTOS configured scheduler. At this stage, the xReadyTasksListEDF contains three tasks namely Task_A, Task_B and IDLE.

The Task_A and Task_B function provides a mock up for running the respective code for the CAPACITY tick count. Once It has executed for CAPACITY tick count then vTaskDelayUntil() pushes the task in waiting until the next cycle.

5.5 Test environment configuration

The test application developed above is used to confirm the correctness of the EDF scheduler implemented. For this purpose, we execute the application and produce the logs of the run-time task scheduling activities and compare the log data with the expected test data. To produce the run-time log file/contents, we have to redirect the output from the embedded device to host machine. Such redirection while debugging the application is possible by Semihosting technique.

Moreover, FreeRTOS provides special trace and debugging functions/macros that can be utilised as needed. We will implement traceTASK_SWITCHED_IN(), traceTASK_SWITCHED_OUT(), traceTASK_DELAY_UNTIL() and traceTASK_INCREMENT_TICK(xTickCount) trace macros.

5.5.1 Trace Macros

FreeRTOS (2022) presents that trace hook macros are a very powerful feature that permit us to collect data on how our embedded application is behaving. The document also advises that Macro definitions must occur before the inclusion of FreeRTOS.h and the easiest place to define trace macros is at the bottom of FreeRTOSConfig.h, or in a separate header file that is included from the bottom of FreeRTOSConfig.h

- traceTASK_SWITCHED_IN()

  This is called every time a task switches in. Called after a task has been selected to run. At this point pxCurrentTCB contains the handle of the task about to enter the Running state.

```
#define traceTASK_SWITCHED_IN() {
                char taskName[25];
                      getTaskName(taskName);
```

```
                        printf("Task IN: %s\n", taskName );
                    }
```

- traceTASK_SWITCHED_OUT()

  This is called every time a task switches out. Called before a new task is selected to run. At this point pxCurrentTCB contains the handle of the task about to leave the Running state.

```
        #define traceTASK_SWITCHED_OUT()
                {
                   char taskName[25];
                        getTaskName(taskName);
                        printf("Task OUT: %s\n", taskName );
                    }
```

- traceTASK_DELAY_UNTIL()

  This is called when the running task suspends itself by calling vTaskDelayUntil. Called from within vTaskDelayUntil()

```
        #define traceTASK_DELAY_UNTIL() {
                char taskName[25];
                    getTaskName(taskName);
                    printf("Task Delay: %s\n", taskName );
            }
```

- traceTASK_INCREMENT_TICK(xTickCount)

  This is called during the tick interrupt.

```
          #define traceTASK_INCREMENT_TICK(xTickCount)
                      {
                              printf("Tick Count: %d\n",xTickCount);
                        }
```

5.6 Test Results

The test result produced from the test application is verified against the expected output for the test cases as below.

- Test case :

  Assuming two tasks A and B with period and capacity as given in the table below.

Task	Deadline	Capacity
A	4	2
B	6	2

Table 5.8: Task period and capacity

Here, the processor utilisation (U) = $2/4 + 2/6 = 0.83$ (83%) which is less than 1. That means, the EDF algorithm ensures that both the tasks get executed without missing deadlines.

The test application produced the trace log contents presented in Figure 5.2.

The Figure 5.2 shows the trace logs produced by the test application. The sequence of tasks execution based on the period and capacity are received as expected. So we can confirm that the EDF algorithm is meeting the functional requirement of the project.

```
Tick Count : 1
Tick Count : 2
Task Delay : Task - A, Task Out : Task - A
Task IN: Task - B

Tick Count : 3
Tick Count : 4
Task Delay : Task - B, Task Out : Task - B
Task IN: Task - A

Tick Count : 5
Tick Count : 6
Task Delay : Task - A, Task Out : Task - A
Task IN: Task - B

Tick Count : 7
Tick Count : 8
Task Delay : Task - B, Task Out : Task - B
Task IN: Task - A

Tick Count : 9
Tick Count : 10
Task Delay : Task - A, Task Out : Task - A
Task IN: Task - B

Tick Count : 11
Tick Count : 12
Task Delay : Task - B, Task Out : Task - B
Task IN: Task - A

Tick Count : 13
Tick Count : 14
Task Delay : Task - A, Task Out : Task - A
Task IN: Task - B

Tick Count : 15
Tick Count : 16
Task Delay : Task - B, Task Out : Task - B
Task IN: Task - A

Tick Count : 17
Tick Count : 18
Task Delay : Task - A, Task Out : Task - A
Task IN: Task - B

Tick Count : 19
Tick Count : 20
Task Delay : Task - B, Task Out : Task - B
Task IN: Task - A

Tick Count : 21
Tick Count : 22
Task Delay : Task - A, Task Out : Task - A
Task IN: Task - B

Tick Count : 23
Tick Count : 24
Task Delay : Task - B, Task Out : Task - B
Task IN: Task - A
```

Figure 5.2: EDF test results

Chapter 6

Evaluation

The purpose of this chapter is to present the evaluation of the whole project components including the the testing and evaluation of the implementation code, test application and the result achieved. This chapter also reviews the achievements against the functional and non-functional requirements specified in the beginning. Evaluations, findings and recommendations for addressing deficiencies identified are included in this chapter.

The EDF and static algorithms were tested for schedulability by measuring the success ration with generated task sets with utilisation of 85%:

$$\text{Success Ratio} = \frac{\text{\# task set with no missed deadlines}}{\text{\# generated task sets}} \tag{6.1}$$

Higher the ratio, higher the success rate of scheduling algorithm.

As expected the EDF and RMS had same success ratio of 100% until the number of task reached 35. After crossing the the 35 threshold, although EDF had degraded in performance, it still outperformed RMS and managed to schedule at success rate of 60% comparatively higher than RMS with success rate of 30%. This is expected as task increased the CPU utilisation increased and RMS is known to have a bound of 65% CPU utilisation. Therefore RMS performed worse than EDF which can handle utilisation up to 100% utilisation.

6.1 Critical Review of Test Application and Results

The completion of this project work is a part of successful course completion requirement for BSc Computer Science degree course. This work is cumulative outcome of the leanings made and

knowledge acquired throughout the three year course period. Working level knowledge of Operating system, embedded systems, c programming, low level programming, project management are minimum requirement for successfully completing the project.

60 task sets with 85% utilisation were generated to measure the success ratio of the scheduling algorithm. With tasks sets below 30 both schedulers gained success ratio of 100% as the number of tasks generated exceeded 35, performance of RMS significantly dropped. On the other hand EDF maintained a 100% success ration till number of tasks reached 50. Although EDF outperformed RMS, EDF's success ratio degraded after 50 tasks. This is due to the overhead that accumulates with constant preemption of tasks at higher number of tasks.

6.2 Summary of Achievements

We manage to successfully implement EDF in FreeRTOS and perform a schedulability test using success ratio as a measure. 60 task sets with 85% utilisation were generated to measure the success ratio of the scheduling algorithm. It was clearly observed EDF outperformed RMS with higher success ration although performance of EDF degraded after 50 task sets.

Chapter 7

Conclusion

Our implemented EDF in FreeRTOS succeeded at a satisfactory standard. It is not perfect nor a failure, it successfully delivers an alternative dynamic EDF scheduling algorithm that equals FreeRTOS's static RM scheduling algorithm. We not only replace FreeRTOS's scheduling algorithm but improve its schedulability standard as the nature of EDF lets its scheduling ability surpass that of RMS. In depth discussion was conducted on dynamic over static scheduler and then feasibility of EDF over RMS scheduling policies. Upon establishing superiority of EDF over RMS, extensive implementation methodology and modification to FreeRTOS codebase were proposed and carried out. The system was then tested with periodic tasks with varying deadlines which it manage to successfully schedule correctly with total utilisation of 83% which is deemed un-schedulable under RMS.

7.1 Application

Although our system was successfully implemented, it demanded significantly learning thorough out the process. When programming tasks needs to be scheduled, it possessed major issues. FreeRTOS required the stack of the newly created tasks to be initialised as if it was already in a running state. This provided FreeRTOS the simplicity to act on the task exactly the same way as it handles tasks that have been already running. To initialise the stack correctly it required to be programmed in the hardware specific assembly language. Thus, it required me to learn ARM Assembly language which we were not expecting and it took time to fully understand the workings of it. However, we manage to use ARM Assembly programming to manually push on the registers to initialise it to make it seem it had been running for a while. This would not have been an issue and would have reduced our implementation time significantly if practice and testing were performed prior to

starting implementing. Furthermore, although we derived an implementation plan, a further rigorous planning would have eased out implementation phase with more time to perform tests and on various devices.

7.2 Future Work

The proposed EDF scheduling solution to the FreeRTOS accomplices its requirements according to its specification. It achieves a well working EDF algorithm, however, it is not rigorously tested and implemented for use outside research purposes. Further testing and modification is need to utilise this implementation at a higher level with industry level workload. RMS is usually preferred over EDF in the industry due to its predictability and lower overhead nature. EDF scheduling are known to be highly preemptive than the its static counterparts increasing context switching and with each context switch comes unavoidable overhead. Thus, further research on improving and optimising EDF actions would yield higher competitiveness with RMS.

Bibliography

Barry, R. (2016), *Mastering the FreeRTOS Real Time Kernel, A Hands-On Tutorial Guide*, Real Time Engineers Ltd.

Belagali, R., Kulkarni, S., Hegde, V. & Mishra, G. (2016), Implementation and validation of dynamic scheduler based on lst on freertos, *in* '2016 International Conference on Electrical, Electronics, Communication, Computer and Optimization Techniques (ICEECCOT)', pp. 325–330.

Buttazzo, G. C. (2005), 'Rate monotonic vs. edf: Judgment day', *Real-Time Systems* **29**(1), 5–26.

Buttazzo, G. C. (2011), *Hard Real-Time Computing Systems Predictable Scheduling Algorithms and Applications*, Springer.

Carraro, E. (2016), 'Implementation and test of edf and llref schedulers in freertos'.

Ferreira, J. F., Gherghina, C., He, G., Qin, S. & Chin, W.-N. (2014), 'Automated verification of the freertos scheduler in hip/sleek', *International Journal on Software Tools for Technology Transfer* **16**(4), 381–397.

FreeRTOS (2022), `https://www.freertos.org/index.html`. [Online; accessed 01-April-2022].

Hahm, O., Baccelli, E., Petersen, H. & Tsiftes, N. (2016), 'Operating systems for low-end devices in the internet of things: A survey', *IEEE Internet of Things Journal* **3**(5), 720–734.

Ibrahim, D. (2020), *ARM-Based Microcontroller Multitasking Projects: Using the FreeRTOS Multitasking Kernel*, Newnes.

Kase, R., Van Chu, T. & Kise, K. (2016), 'Efficient user space scheduling library for freertos', *The 78th Annual Meeting of the Information Processing Society of Japan* **5**, 09.

Kumar, S. (2020), 'Earliest deadline first (edf) cpu scheduling algorithm', `https://www.geeksforgeeks.org/earliest-deadline-first-edf-cpu-scheduling-algorithm/`. [Online; accessed 10-03-2022].

Leung, J. Y. (2004), *Handbook of scheduling: algorithms, models, and performance analysis*, CRC press.

Liu, C. & Layland, J. (1973), 'Scheduling algorithms for multiprogramming in a hard-real-time environment'.

Masoud, W. (2021), 'Earliest deadline first (edf) scheduling algorithm', `https://microcontrollerslab.com/earliest-deadline-first-scheduling/`. [Online; accessed 24-12-2021].

Musaddiq, A., Zikria, Y. B., Hahm, O., Yu, H., Bashir, A. K. & Kim, S. W. (2018), 'A survey on resource management in iot operating systems', *IEEE Access* **6**, 8459–8482.

Oliveira, G. & Lima, G. (2020), Evaluation of scheduling algorithms for embedded freertos-based systems, *in* '2020 X Brazilian Symposium on Computing Systems Engineering (SBESC)', pp. 1–8.

Páez, F. E., Urriza, J. M., Cayssials, R. & Orozco, J. D. (2015), Freertos user mode scheduler for mixed critical systems, *in* '2015 Sixth Argentine Conference on Embedded Systems (CASE)', IEEE, pp. 37–42.

Pathan, R. M. (2016), Design of an efficient ready queue for earliest-deadline-first (edf) scheduler, *in* '2016 Design, Automation Test in Europe Conference Exhibition (DATE)', pp. 293–296.

Peckol, J. K. (2019), *Embedded systems: a contemporary design tool*, John Wiley & Sons.

Pelleh, M. (2006), A study of real time scheduling for multiprocessor systems, *in* '2006 IEEE 24th Convention of Electrical & Electronics Engineers in Israel', IEEE, pp. 295–299.

Siewert, S. & Pratt, J. (2015), *Real-Time Embedded Components and Systems with Linux and RTOS*, Stylus Publishing, LLC.

Toma, A., Meyers, V. & Chen, J.-J. (2018), 'Implementation and evaluation of multi-mode real-time tasks under different scheduling algorithms', *OSPERT 2018* p. 13.

Wang, K. C. (2017), *Embedded and Real-Time Operating Systems*, 1st edn, Springer Publishing Company, Incorporated.

Xiao, P. (2018), *Designing Embedded Systems and the Internet of Things (IoT) with the ARM mbed*, John Wiley & Sons.