Scheduling policy to control interference between tasks in an LED driver
Literature study, Design and Implementation

Popa, Stefan

Award date:
2022

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Interconnected Resource-aware Intelligent Systems Research Group

Scheduling policy to control interference between tasks in an LED driver

Literature study, Design and Implementation

Stefan Popa

Supervisors:
Geoffrey Nelissen - TU/e
Erik de Wilde - Signify

Eindhoven, September, 2022

Abstract

An LED driver has multiple tasks that need to be executed in parallel. Its core functionality is that of a power supply. To implement that, several control tasks need to be executed at constant intervals. A communication protocol is also run to provide remote control capabilities. The signals for this communication protocol are captured through an interrupt. The communication interrupt is unscheduled and can interfere with control tasks. In this thesis, the current implementation is analysed. Then, three solutions to the problem are proposed. The first one is changing the scheduling algorithm of the system. The second one is using a server to manage the interrupt. The third solution is to limit the preemption of control tasks. The theoretical and practical aspects of the solutions are examined. It is concluded that only limiting the preemption of control tasks is an improvement over the current implementation.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This thesis concerns real-time scheduling policies for a real-time system. Real-time systems are computing systems that aim to create a response to events within a predictable window of time. The predictability of real-time systems is critical to their operation, ensuring that the system functions correctly. Real-time systems are often found at the border between the cyber and physical world, enabling the control of physical processes while being able to interact with the digital world. They are usually implemented with an embedded microcontroller, and their applicability ranges from small consumer electronics to safety-critical systems such as aeronautics and medical devices.

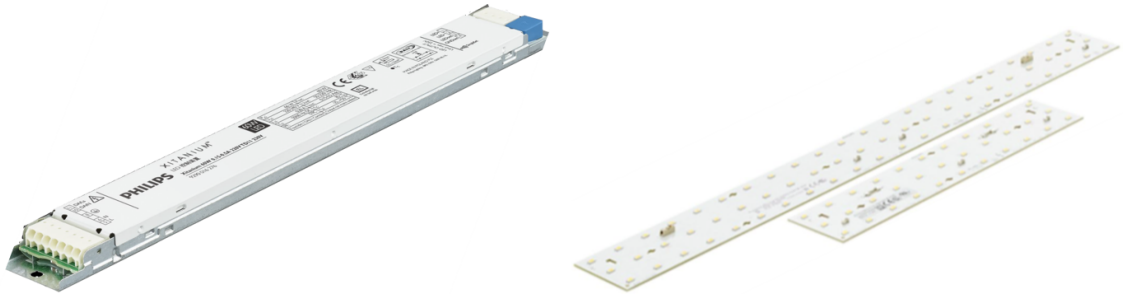
The functionality of a real-time system is often realised by several tasks. These tasks must be executed in conformance with timing requirements, which may differ for each task. One simple but common way to classify tasks is based on their deadlines. There are hard and soft real-time tasks. Hard real-time tasks' deadlines need to be met to ensure the functionality of the system. On the other hand, soft real-time tasks have deadlines that do not need to be met, but if they are missed, it affects the quality of service(QoS) of the system proportionally to the time passed after the deadline. Moreover, some tasks do not have a clear deadline, with their execution constraints coming from their purpose. To ensure the correct behaviour of the system, hard real-time tasks need to be scheduled such that they all meet their deadlines, and soft real-time tasks have their QoS maximised. Another way of classifying real-time tasks is based on their arrival patterns. Some tasks arrive periodically, while others arrive sporadically. The main focus of this thesis is to explore existing scheduling algorithms and implement an adequate solution to schedule the tasks of a lighting driver used in smart LED lighting devices.

In this chapter, the context and background for the thesis are presented, and we introduce the problem that is addressed in this thesis. Then the research goal and approach are presented, together with the main contributions of this thesis.

1.1 Context and Background

The project is done at the LED Electronics department of Signify, whose aim is to design and develop LED drivers. This is part of the technological development around light. Light is an essential part of the natural environment, which makes it an integral part of many applications. In the last decades, the advent of the light-emitting diode (LED) enabled more efficient and versatile lighting applications. A central device in many such applications is the driver. Such a driver can be seen in Figure 1.1a. The main function of the driver is to supply electrical power to the LED load. One example of such an LED load can be seen in Figure 1.1b. The load displayed in the figure is called linear due to the shape the LEDs are arranged in. The driver has to ensure that the electric current that goes to the load is stable and at the desired level, to create a qualitative lighting experience. On top of this functionality, the driver may communicate with a local area controller, which is a device that can be used by humans to configure and send commands to the

driver, and optionally receives input from a sensor and other devices, increasing the possible use cases of LED systems.



(a) A linear LED driver in its case (b) Linear LED load

Figure 1.1: Examples of an LED driver and load

In recent years, improvements in the areas of embedded computers, actuators and more accurate sensors shaped the technology used for a lighting driver. The driver was initially an analog power supply with no additional features, such as controlling the driver remotely or determining the remaining lifetime of the device. Adding them increased the usability and marketability of the driver. To implement them, a microcontroller was added to the driver. The addition of the microcontroller to the driver made possible a digital implementation of the power converters, ensuring the stable output of power. All functionality is structured in software modules, which are further structured in tasks. Two particular kinds of tasks are control and communication tasks. Control tasks are periodic tasks without a precise deadline but whose functionality depends on executing them as fast as possible. On the other hand, communication tasks have a precise deadline dictated by the protocol they implement, and if they do not meet their deadline, the protocol will not be satisfied. This thesis will study different scheduling algorithms that can be used to accommodate the timing requirements of these two types of tasks. While the problem, solution and approach are meant to be general, a particular use case will be employed to illustrate the concepts better and allow practical validation.

1.2 Use cases

1.2.1 Overview

The system considered in this thesis consists of a lighting system with one LED load. The system can be controlled remotely. It contains a light emitter(i.e., the LED load), its driver, a local area controller and a power supply. The components of the driver, as well as the interactions between them, are captured in a component diagram. The diagram is shown in Figure 1.2.

The possible interactions of the driver with the actors around it are shown in Figure 1.3. The first actor included in the system is the local area controller. Its purpose is to specify the characteristics of the light being emitted, such as dimming level or warmth. The controller passes its commands to the LED driver using DALI, which will be described in more detail in subsection 1.2.3. Apart from that, the driver can be configured through near-field communication(NFC), which is incorporated into the external storage. On the electrical input and output, the driver is supplied by mains with alternative current (AC), which is then converted to direct current (DC) to supply the LEDs.

A picture of the components of the system, as they are laid out on the workbench, can be seen in Figure 1.4. A USB hub connecting the computer to various components can be seen on the left. The computer uses an interface to the DALI protocol. The LED load can be seen on the right. In the middle, there is the LED driver that is used. It has three wired connections to the mains

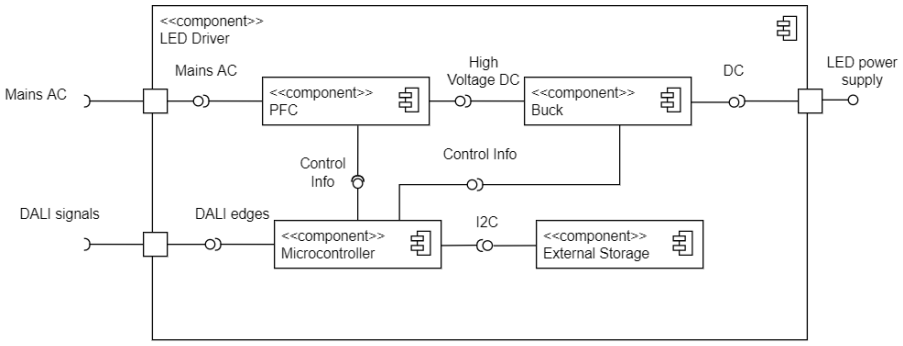


Figure 1.2: UML Component Diagram of the Driver

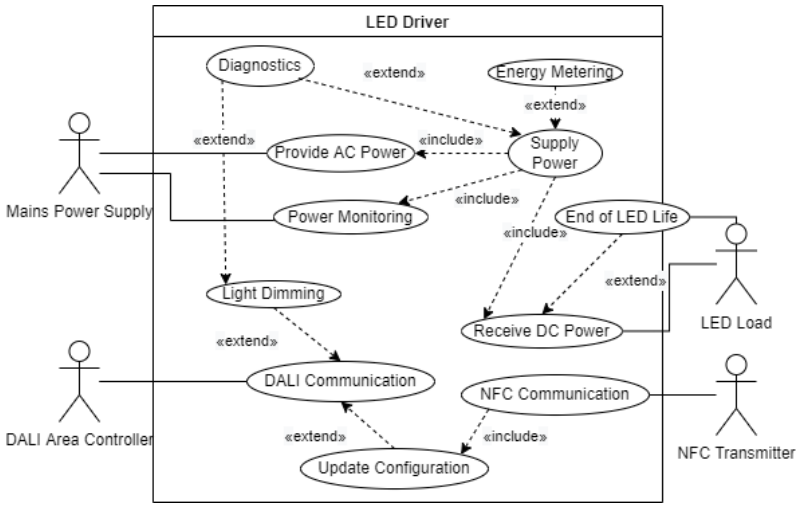


Figure 1.3: Use case diagram of the system in which the driver operates

power supply, the communication line, and the LED load. In the upper right corner, a colorimeter can be seen. It is used to measure light flicker.

1.2.2 Electrical Components

The main purpose of the driver is to supply power to the light emitters, being an AC-DC power supply. The electronic components decrease the power factor of the driver, leading to increased power consumption and even fines from the electricity supplier. The power factor of a device can be increased by a power factor corrector (PFC).

AC power is composed of a voltage and current wave, which are not necessarily in phase. The difference between the phases of the two waves creates a difference between the apparent power consumed by the device, which is the power the device draws from the mains, and the real power, which is the power that is used by the device. The power factor is the ratio between the real and apparent power, having values between -1 and 1. Having a poor power factor can be penalised by the electricity supplier through extra costs, so it is not desirable.

The circuits of the driver decrease the power factor because they contain inductors and capacitors that shift the current and voltage waves, decreasing the real power used by the driver. To increase the power factor of the driver, a correction needs to be applied, and the circuit that realises it is called power factor corrector (PFC). The PFC contains a transistor that is switched on and off to achieve the correction. However, while the PFC increases the power factor, its circuit creates DC electricity that has a very high voltage. To bring the voltage down to the levels accep-

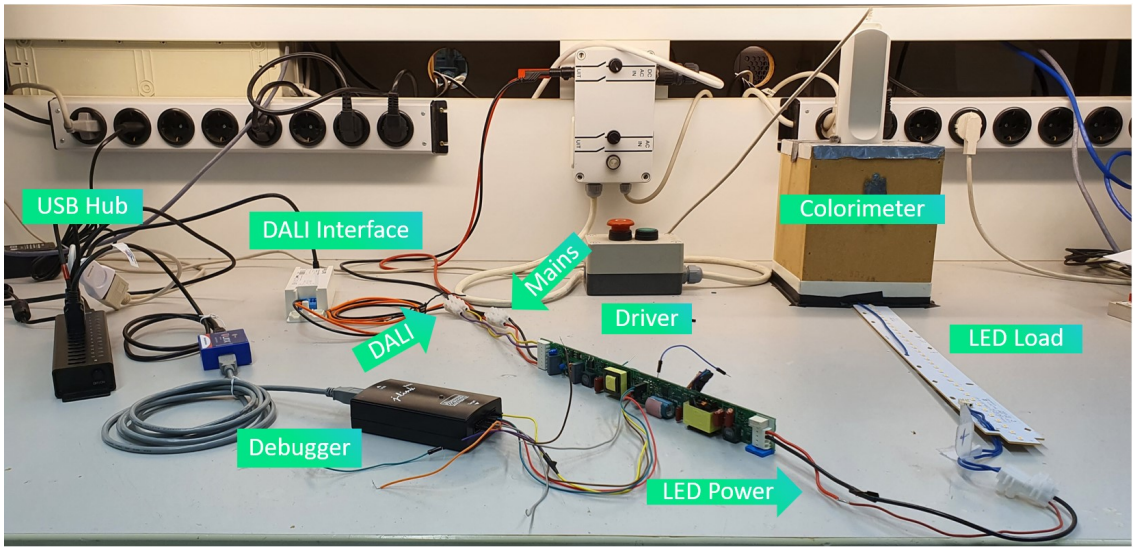


Figure 1.4: Workbench

ted by the light emitter, a buck converter circuit is used. Both the PFC and the buck converter employ transistors to control their behaviour. Their transistors are switched on and off based on a duty cycle determined by a controller, ensuring the output level of each circuit corresponds to the desired one. Both the PFC and the buck converter have a dedicated controller.

The driver contains an ARM M0 microcontroller that executes the controllers of the PFC and buck converter. The microcontroller also implements the DALI communication protocol, which allows the driver to communicate with a local area controller that plays the role of a master of a set of light emitters. Moreover, the driver has an external memory module, which communicates with the microcontroller using the I2C protocol. The memory module is used to store configuration and diagnostics data in a non-volatile form. Other features provided by the driver are monitoring of the mains, energy metering, diagnostics, and estimation of the end-of-life (EOL) of the LED load.

The emitters that produce light are light-emitting diodes (LEDs). The load controlled by the driver consists of multiple LEDs connected in parallel or series.

1.2.3 DALI communication protocol

Digital Addressable Lighting Interface [1] (DALI) is a standardised wired communication protocol dedicated to digital lighting control. The DALI protocol adheres to the master-slave design pattern. There are three types of devices: control devices, input devices, and control gears. Control devices play the role of masters, being application controllers. Input devices consist of sensors detecting the state of the environment. They can be queried by the control device. Control gears consist of lighting actuators, usually the driver of the lighting device. The control devices and the control gear are connected through a wired bus. An example of such a DALI system can be seen in Figure 1.5.

At the physical layer, connections are realised through a two-wire bus. The protocol is a digital one, using logic high and low signals across the wires. The nominal voltage of the bus is 16V, with the logic high represented by a voltage level between 9.5 and 22.5V. The logic low is represented by a voltage below 6.5V. The rise and fall time of an edge is capped at 15 us. The bus can also act as a power supply for the sensors connected to the network.

At the data link layer, messages are marked by start and stop framing. At the networking layer, a DALI network can have 64 short addresses, which may be used by either slaves or masters. Slaves can be logically grouped, and there can be up to 16 groups. Masters are able to send 16-bit

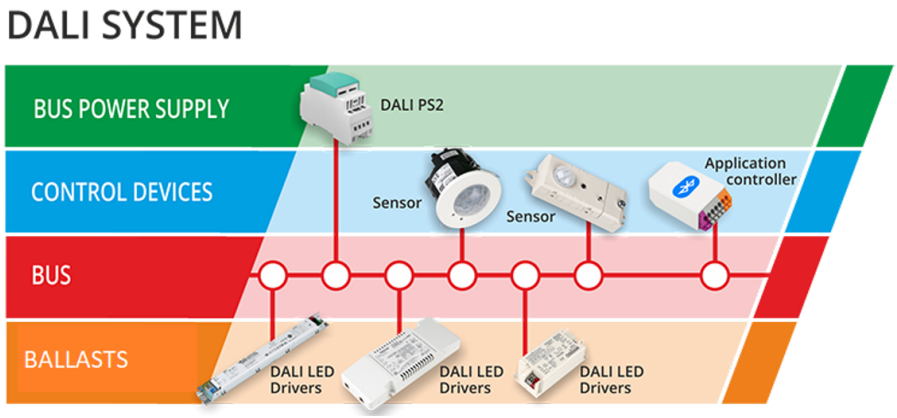


Figure 1.5: Example of a DALI system[2]

forward frames to control gear and 24-bit frames to input devices, and they receive 8-bit backward frames as a reply.

DALI also allows the definition of scenes, in which every device has a particular setting to create a lighting scene in the room they are deployed. Information about group membership and scene-setting is stored locally in each slave. Slaves can be addressed in three modes: individually, through their group or all at once through a broadcast. There are three types of commands that can be issued by masters: instructions, configuration and queries.

1.3 LED Driver and Software Components

The elements on the LED driver employed in the use case can be seen in Figure 1.6. The driver converts the AC supplied by the mains to the DC required by the LED load. Its circuits are split into four parts. The input circuit converts the current from AC to DC. The second part is the PFC circuit, which corrects the poor power factor introduced by the driver. The third part is a buck converter that lowers the voltage to the level that is required by the LEDs. This is done both to bring the voltage to the normal functioning parameters of the LED load as well as to implement light dimming to the desired level. The fourth part is the output circuit that passes current to the LEDs. The computation power is given by an XMC1402 microcontroller mounted on the driver.

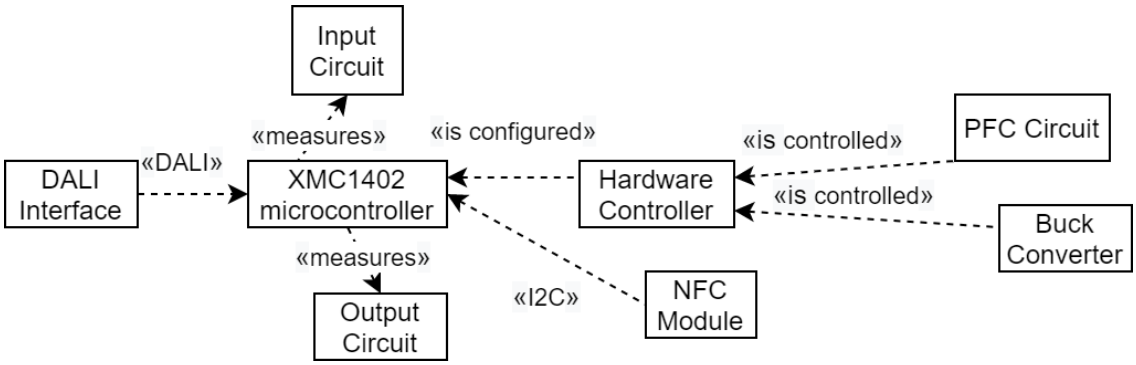


Figure 1.6: UML Class diagram of the driver

1.3.1 Controllers and other circuit-related features

The PFC and buck converter are controlled digitally. A general block diagram for the controllers can be seen in Figure 1.7. Their controllers consist of two parts. The first one is a cycle-by-cycle controller implemented by configurable hardware, switching the transistors with high frequency. This controller reacts to current fluctuations in the circuit. The second controller is a feedback controller implemented by software. The feedback is received from the bus voltage, measured at different points in the circuit for the PFC and the buck converter controllers. The bus voltage is measured by an analog to digital converter. The output of the software controller is a pulse width modulation(PWM) signal that configures the hardware controllers. The PWM signal is generated by capture and control units(CCU), which are part of the industrial control peripherals of the microcontroller. The software controllers must be executed at a fixed frequency for the quality of the light to be at an acceptable level.

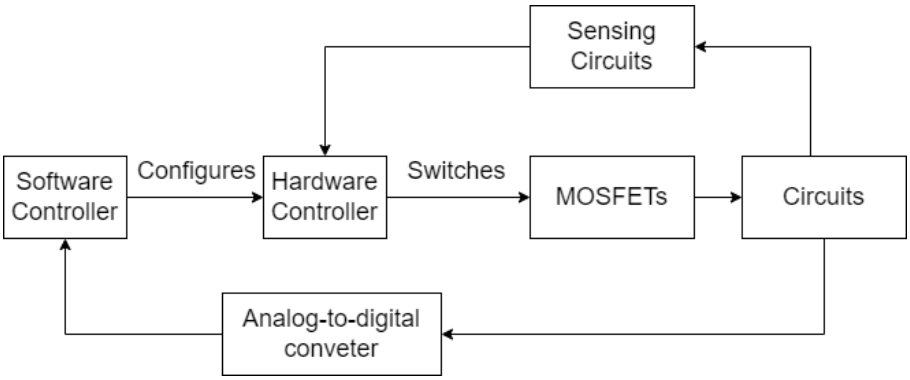


Figure 1.7: Generic block diagram of the controllers

As already said, other features implemented on the controller are mains monitoring, energy metering, diagnostics, and estimation of the EOL of the LED load. Mains monitoring serves two purposes. Firstly, the driver detects whether the power source is AC or DC, together with the frequency of AC. This makes the driver capable of using both AC and DC from the power supply, adapting the behaviour of the PFC controller accordingly. Secondly, the driver stores its configuration parameters in non-volatile memory when the power supply is stopped. To do this, it uses a narrow window of time between the moment the stop is detected and when the power actually stops. Energy metering is used to measure the energy efficiency of the driver. The diagnostics feature enables the driver to auto-detect malfunctions. Estimation of the EOL of the LED load facilitates timely replacement and continuous service for the load. These features have hard real-time requirements because their functionality relies on storing parameters of the current of the circuit at a fixed frequency. Another reason for measurements at fixed intervals is that they are not done by sampling the analog lines directly but rather by reading sampled values from dedicated registers, which are overwritten each time a new value is taken.

1.3.2 DALI Implementation

The DALI protocol is implemented on several layers. The hardware support is provided by the CCU. When receiving, the CCU captures the timestamps of rising and falling edges, and when transmitting, it times the interval between outgoing edges. The CCU communicates with the microcontroller through interrupts, which happen when either two edges are detected on the wire that connects the master to the slave or when a new edge is sent from the slave to the master.

The generated interrupt stores received edges in a queue or sets the line to the corresponding value when transmitting. The edges are picked up by a decoder, which assembles the DALI message and makes it available for an application layer interpreter, that reacts accordingly.

1.3.3 EEPROM synchronization

As mentioned in section 1.2, the driver uses an external memory EEPROM module. This is used to store non-volatile data. As the EEPROM module is external to the microcontroller of the board, reading and writing to it create a large latency. To ameliorate this, the microcontroller keeps a cache of the EEPROM in the flash memory. The content of the two is synchronised when the number of write modifications in the cache passes a threshold. Moreover, the EEPROM module can also be accessed through NFC, and when that happens, the cache needs to be synchronised. The microcontroller uses an interrupt to signal when the EEPROM has been accessed through NFC. The I2C protocol is used to communicate with the external EEPROM. The protocol is handled by the Universal Serial Interface Channel (USIC), which decreases processing time.

1.4 Current Implementation

The current firmware of the driver assigns tasks to two different schedulers. The first scheduler runs inside the process context. That scheduler is implemented as the main loop of the program. It schedules tasks that do not have hard real-time requirements. The scheduler is cooperative and non-preemptive, iterating through all events and executing the triggered events until completion. This logic can be seen in Listing 1.1. It supports both sporadic and periodic tasks. The tasks' period is maintained by software timers, which are updated with a frequency of 1 kHz.

```
for(;;) {
  update_system_time();
  for each event e {
    if(e is triggered) {
      execute_associated_callback();
    }
  }
}
```

Listing 1.1: Main Loop

Tasks which have hard real-time constraints are executed by a cyclic executive scheduler divided into twelve time slots. Tasks handled by this scheduler correspond to features related to circuit control and measurement. A timer is used to call the scheduler at a frequency of 12 kHz at the beginning of each time slot. The assignment of tasks to time slots determines the frequency at which they are executed. Since the high-frequency scheduler functions off an interrupt, and the low-frequency one is executed in process context, the former can interrupt the latter and thus ensures that its tasks are executed on time.

There are four interrupts used in the system, seen in Table 1.1. As seen from the priority assignment, the DALI protocol interrupt has a higher priority than the cyclic executive scheduler and can interfere with it. This is because DALI does not have any error correction mechanism, and therefore misses are not allowed. The interrupt is triggered when the driver is receiving a message, and it detects an edge on the DALI line or when the timer triggers when transmitting.

Function	Priority (0=highest)	Frequency
DALI Protocol	0	Sporadic
Cyclic Executive Scheduler	1	12 kHz
Cooperative Scheduler	3	1 kHz
NFC Busy	3	Sporadic

Table 1.1: System interrupts

Infineon XMC1000/4400 microcontroller The driver has an embedded microprocessor to handle communication and control the circuits that regulate the power it outputs to the LEDs. The current design uses an Infineon XMC1402 microcontroller, which contains an ARM Cortex-M0

processor. However, in future iterations, a move towards an XMC4400 with an Arm Cortex-M4 processor is considered. This would make more space and processing power available on the platform, as seen in Table 1.2. The increase in capacity opens the door to future features that will complement the existing ones, and this is part of the trend described in section 1.1. One disadvantage is that the new processor lacks some of the peripherals that the old one has, but that would be solved with other models that also use a Cortex-M4 core. One example of such a lacking peripheral is the LED brightness & color control module, which handles LED color and brightness transitions. Therefore the XMC4400 will also be considered throughout the thesis to compare the performance of the solution on two platforms as well as to create a solid base for future iterations of the product.

Feature	Processor	Processor Speed	RAM	Flash	Peripherals clock
XMC1402	Arm Cortex M0	48 MHz	16 kB	64 kB	96 MHz
XMC4400	Arm Cortex M4F	120 MHz	80 kB	512 kB	120 MHz

Table 1.2: Feature Comparison of XMC 1402 and XMC 4400

1.5 Problem Description

Multiple software tasks need to run simultaneously on the same processor: the control software, DALI and I2C communication. The controllers are crucial in ensuring that the emitted light meets the quality standards. At the same time, the DALI communication needs to be handled within the time constraints of the protocol. Otherwise, the device would not be compatible with other devices on the market and would not receive a certification. These conflicting requirements make scheduling tasks difficult.

The difference between the DALI and I2C protocols is that I2C has the advantage of having direct memory access and hardware decoders. The reception and transmission of messages at the physical layer are performed by hardware. This leaves only the application layer to the processor. As the application layer does not have strict timing requirements, it can be executed in the background as part of the cooperative scheduler. On the other hand, DALI is a domain-specific communication protocol. Because its use is not widespread, it does not benefit from hardware support. Thus, DALI messages need to be received and decoded by the microcontroller. The part with the strictest timing requirement is the reception and transmission of signals on the communication medium, i.e. on the wire.

The timing requirements of the physical layer of the DALI interrupt are different between the case when the driver is transmitting and when it is receiving messages. When receiving, the timing of incoming signals is recorded by the CCU. The DALI interrupt must read the registers of the CCU and store the recorded timing before other signals come. If other signals come before that happens, they will overwrite the registers. The timing requirements are more relaxed when the driver is transmitting messages, as it controls the timing of the signals. It must nevertheless meet the requirements of the DALI protocol.

Thus the microcontroller must handle a task with strict timing requirements. To do that, it uses an interrupt. The interrupt has the highest priority in the system. When it triggers, it

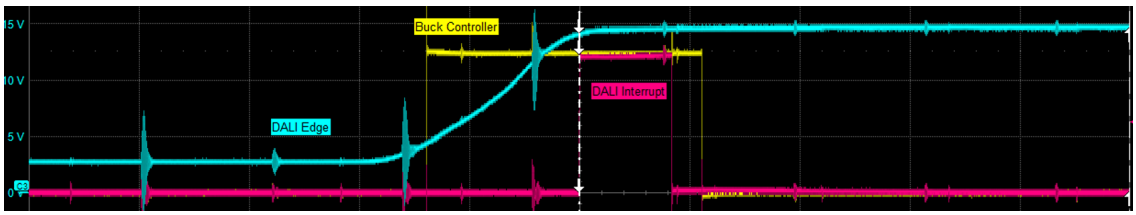


Figure 1.8: DALI interrupt interfering with the buck controller after an edge is received

interferes with control tasks from the cyclic executive. This interference can be seen in Figure 1.8. Moreover, the interrupt is sporadic and unscheduled, and there is no protection for the execution of the control tasks. Thus, it interferes with them, creating spikes in the output current, and decreasing the lighting quality. The expectation is that a more flexible and robust solution would enable the two types of tasks to execute with limited interference.

1.6 Goal and approach of the research

Given the challenges laid out in section 1.5, the goal is to introduce a new scheduling policy for the tasks that are executed on the driver such that the interference between the communication protocol and the control tasks is controlled, taking into account the real-time attributes and requirements of both task category.

This involves choosing or designing an algorithm which guarantees that the functional and QoS requirements of the lighting application and of the communication protocol are met. Moreover, the interference between tasks needs to be controlled. The algorithm will be theoretically analysed to guarantee this. Then, a practical implementation will be created to test the theoretical results.

The research question that stems from the problem description is, therefore:

> *How can a real-time scheduling policy be applied to a subset of tasks executed on an LED driver, such that all tasks meet their deadlines, and the interference between communication-related tasks and control-related tasks is controlled?*

Specifically, for the specific use case considered in this project, we must control the interference between tasks such that the flickering of the light is below a predefined threshold, the hard real-time timing requirements of the DALI communication protocol are respected, and the response time of the control tasks is minimised. The above research question sprawls other questions related to the steps that must be taken to answer the main question, reflecting the approach that will be used.

1. How can the system's tasks be modelled, and what properties and requirements can be derived from the model?
2. Which scheduling policy may produce a schedule for the tasks running on the LED driver, such that all tasks meet their deadlines?
3. What techniques can be used to control the interference between control and communication-related tasks?
4. How can the policy producing a valid schedule with the least amount of interference be implemented on the targeted specific platform?
5. How does the produced implementation compare with an existing implementation in terms of interference and response time?

1.7 Contributions

The contributions of this thesis reflect the research questions and can be summarised as follows:

1. A study and review of the state-of-the-art real-time scheduling algorithms for a single-core real-time platform
2. A scheduling policy for an LED driver, together with a theoretical analysis

3. A technique for controlling the interference between the communication and control-related tasks
4. A proof-of-concept implementation of the proposed algorithm on the targeted hardware to assess its performance and industrial applicability.
5. An evaluation of the proof-of-concept implementation

1.8 Thesis outline

The remaining chapters of this thesis are organised in the following way. Chapter 2 provides an overview of the state-of-the-art algorithms for real-time scheduling on single-sore platforms. Chapter 3 describes how the driver system can be modelled. Chapter 4 presents three different theoretical solutions together with their analysis. Chapter 5 shows how an RTOS can be evaluated for the given use case. Chapter 6 presents how a subset of the system can be ported to FreeRTOS. Chapter 7 presents different implementations of solutions that directly control the DALI interrupt. Chapter 8 contains conclusions and reflections.

Chapter 2

State-of-the-Art

This chapter will present a survey of the relevant literature and technologies related to the topic. It is presented in two parts. The first part covers literature about different scheduling algorithms and techniques. The second part presents different operating systems available on the market, which may be used as a platform for implementing the algorithms.

2.1 Scheduling algorithms

The kind of tasks and constraints that the scheduling algorithm needs to accommodate are laid out in section 3.7. There are periodic hard-real time tasks without a well-defined deadline, but with QoS inverse proportional to the jitter and interference experienced, and sporadic tasks with hard deadlines. Thus the system needs to deal with a mixed task set. This section will present different algorithms that can be used to schedule periodic hard-real time task sets on a uniprocessor platform. They provide guarantees that the tasks meet their deadlines, together with various techniques to study their jitter.

Of the few algorithms designed to schedule periodic tasks, the simplest one is the cyclic executive. It involves having the processor time divided into time slots, each executing tasks in a sequence that repeats itself. This is the method currently used on the LED driver. Its simplicity makes it highly analysable and easy to use. However, there are some drawbacks of this method, outlined in [3], which appear under certain conditions. The first one is poor management of overhead situations, in which a task exceeds its execution budget and may overrun the next timeslot. The second one is its sensitivity to changes. When a task is redesigned and requires more computation time or a higher frequency, a new scheduler needs to be computed and analysed. Another problem is that aperiodic tasks are hard to integrate into the schedule.

Before presenting other algorithms, an overview of notations is needed to establish common ground, as many papers define their own different notations. Table 2.1 summarizes the notations used in this section. Another useful notion is the concept of a necessary and sufficient condition for the schedulability of a task set under a given algorithm. If a task set is schedulable by the algorithm, it is implied that it meets the necessary condition. On the other hand, if a task meets a sufficient condition, the task set is schedulable.

Table 2.1: Notations Overview

Notation	Explanation
Γ	Set of tasks
C_i	Worst-case execution time of task τ_i
T_i	Period of task τ_i
D_i	Deadline of task τ_i, relative to its release
W_i	Worst-case response time of a task τ_i

Notation	Explanation
B_i	Best-case response time of a task τ_i
$U_i = C_i/T_i$	Processor utilization factor of task τ_i
$U = \sum_{i=1}^{n}(C_i/T_i)$	Total processor utilization for a task set
U_{lub}	Least utilization bound of an algorithm - the maximum total utilization for which a task set is schedulable given a certain scheduling algorithm.
U_p	Total utilization of periodic tasks in a task set
U_s	Server utilization

Preemptability One characteristic of scheduling algorithms is whether they are preemptive or not. A preemptive algorithm considers the tasks preemptable, meaning that they may be interrupted by higher priority tasks. A non-preemptive algorithm lets tasks execute until completion after they are selected to execute. There are also algorithms that employ limited preemption, in which tasks can only be preempted under certain conditions.

Priority-based algorithms are a class of scheduling algorithms where tasks are executed based on their priority. Although they can be used with any preemption environment, they are most commonly used assuming full-preemptability. The algorithms can be classified based on the run-time behaviour of priorities. If priorities are fixed, the algorithm is static, and if not, it is said to be dynamic.

Static priority algorithms are the easiest to implement of the two options, as they only rely on a system that can support priority scheduling and preemption. The priority assignment is given at compile-time, and they are fixed. In contrast, to support a dynamic assignment, the system also needs to recompute the new priority of tasks. Static priority algorithms are important because they can be implemented natively in all real-time operating systems.

2.1.1 Rate-monotonic scheduling

Rate-monotonic scheduling (RMS) is a static priority scheduling algorithm in which priorities are assigned based on the periodicity of each task. Tasks with a smaller period have a higher priority. It has been shown that when deadlines are equal to periods, RMS is optimal among static priority scheduling algorithms [4]. This means that if a task set can be scheduled by another static priority algorithm, then it can also be scheduled by RMS.

There are two sufficient conditions for RMS based on processor utilization. The first one is the Liu-Layland bound[4]. This bound is $U_{lub} = n(2^{1/n} - 1)$, where n is the number of tasks to be scheduled. For a large n, this converges to $\ln 2 \approx 0.69$. The second one is the hyperbolic bound, which is less pessimistic than the Liu-Layland bound[5]. According to this bound, a task set is guaranteed to be schedulable by RMS if $\prod_{i=1}^{n}(U_i + 1) \leq 2$.

2.1.2 Deadline monotonic scheduling

A variation of the RMS is deadline monotonic scheduling (DMS)[6], which is a static priority scheduling algorithm where higher priorities are assigned to tasks that have a smaller relative deadline. When deadlines are less or equal to periods, it is optimal among static-priority scheduling algorithms, meaning that if a task set can be scheduled by another static priority algorithm, DMS is also able to schedule it. When deadlines are equal to periods, DMS has trivially the same behaviour as RMS.

As DMS can handle tasks with smaller deadlines than their periods, it is more suited to schedule control tasks than RMS because if a deadline is smaller than the period of a task, the interval between different task executions has less variation. Indeed, with DMS, the jitter of tasks can be bounded[7]. The schedulability analysis technique that is used for DMS is response time analysis, which can guarantee that a task set is schedulable or not.

2.1.3 Earliest deadline first

Earliest deadline first(EDF) scheduling is a scheduling algorithm using a priority assignment in which higher priorities are assigned to tasks that have earlier absolute deadlines. Because the earliest deadlines change during run-time, the priorities of the tasks change as well, and the algorithm is classified as a dynamic priority algorithm. EDF has been proven to be optimal among dynamic-priority algorithms on a single core[8], meaning that if a task set is schedulable by any other dynamic priority algorithm, it will also be schedulable by EDF.

Under the assumption that deadlines are equal to periods, the least utilization bound of EDF is 1, meaning that it can schedule any task set that is not overloaded. However, EDF may also be used to schedule tasks whose deadlines are shorter than their periods. In this case, the analysis is conducted using processor demand[9]. This checks that tasks do not require more computation time than is actually available in an interval. The processor demand of a task τ_i in an interval $[t_1, t_2]$ is given by the sum of the computation time of instances of τ_i having their release time and deadline in the given interval. This leads to the introduction of a demand bound function (dbf), which is defined as

$$dbf(t) = \sum_{i=1}^{n} \lfloor \frac{t + T_i - D_i}{T_i} \rfloor C_i \text{for any} t \geq 0$$

This function leads to the schedulability condition for EDF with deadlines less or equal to periods: $dbf(t) \leq t, \forall t > 0$. Another schedulability test is to compute the worst-case response time for each task and compare it to its deadline. The computation of the worst case response is based on [10]. The analysis assumes that the worst-case response time of a task will appear when the other tasks have a busy period, meaning that they will keep the processor as busy as possible before the task is released.

EDF leads to a lower number of preemptions compared to a static-priority algorithm[3]. However, it involves larger run-time computations, and it is not always supported by a real-time operating system.

2.2 Scheduling sporadic tasks

The algorithms described in section 2.1 assume a periodic or sporadic task set. However, in the case of the driver's task set, the non-periodic tasks can be activated at a rate that causes problems for the rest of the tasks. To deal with tasks whose activation needs to be limited, there are several algorithms which execute non-periodic tasks while guaranteeing the deadlines of periodic tasks. This section will review the existing research on combining these two task categories.

Background scheduling The simplest solution for combining periodic tasks with non-periodic real-time tasks is to execute the non-periodic tasks while there are no available periodic tasks to execute. This approach intrinsically preserves the schedulability of the periodic tasks. Its main advantage consists of its simplicity. However, under high loads, non-periodic tasks may have large response times. As a note, this is the method that is used in the current implementation of the driver, with the cooperative scheduler acting as the background scheduler for soft and firm hard real-time tasks.

Servers One common way to manage the execution of sporadic tasks is through servers. Servers are defined as periodic tasks that execute sporadic tasks with a certain budget. Server algorithms are categorized into fixed and dynamic priority servers based on the type of algorithm used for scheduling the periodic tasks. They are used to improve the average response time of non-periodic tasks compared to background scheduling.

2.2.1 Fixed priority servers

Fixed priority servers are a class of server algorithms that are used in combination with a static priority scheduling algorithm, such as RMS and DMS. The server task does not take execution time unless there are requests from sporadic tasks, but it does receive a budget, i.e. maximum computation time allotted to them, every period. Usually, the server task has the highest priority in the system.

The algorithms presented in the following subsections are shown in their order of complexity, together with how their schedulability may be analysed and what is the maximum utilization of the server task such that the task set is still schedulable. Table 2.2 shows schedulability tests and maximum utilization of servers.

Polling Server

The polling server algorithm allocates the server task a budget every period. If there are sporadic tasks pending, then the server preserves its budget until it is selected by the scheduling algorithm to execute. Then the server executes sporadic tasks within the limit of its budget. If there are no sporadic tasks pending, the server loses its budget[11].

The server can be considered a periodic task when analysing the schedulability of a periodic task set in the presence of a polling server. Thus, the same analysis techniques used for fixed-priority scheduling can be applied, namely doing a utilization bound test or response time analysis[3].

Deferrable Server

The deferrable server is similar to the polling server, with the difference that its budget is not lost when there are no sporadic tasks pending[11]. The budget is replenished up to its limit every server period. The advantage over the polling server is the improved responsiveness of sporadic tasks.

There are also schedulability tests for the deferrable server that determine whether a task set is schedulable in the presence of a deferrable server. One of them uses the number of periodic tasks as a parameter, while another schedulability test is based on the hyperbolic bound[3].

Priority Exchange Server

The priority exchange server(PES)[11] has its budget replenished every period. If there are no sporadic tasks to be executed, the server exchanges its capacity for the execution time of the active periodic task with the highest priority. The goal of the priority exchange server is to improve the responsiveness of sporadic tasks over the polling server. Compared to the DS, PES has a slightly larger maximum utilization, but it is more difficult to implement[3].

Sporadic Server

The sporadic server[12] functions differently than the other static priority server algorithms because instead of replenishing its budget periodically, it has, by default, a full budget that is consumed when sporadic tasks arrive. The consumed budget is replenished after a time interval equal to the period of the server. Its goal is to improve the responsiveness of sporadic tasks over the polling server.

Table 2.2: Schedulability tests and maximum dimension of servers[3]

Server	Schedulability Test	Hyperbolic Bound Test	Dimensioning
Polling	$U_p \leq n\left[\left(\frac{2}{U_s+1}\right)^{1/n}-1\right]$	$\prod_{i=1}^{n}(U_i+1) \leq \frac{2}{U_s+1}$	$U_s^{max} \leq \frac{2-P}{P}$
Deferrable	$U_p \leq n\left[\left(\frac{U_s+2}{2U_s+1}\right)^{1/n}-1\right]$	$\prod_{i=1}^{n}(U_i+1) \leq \frac{U_s+2}{2U_s+1}$	$U_s^{max} \leq \frac{2-P}{2P-1}$

Server	Schedulability Test	Hyperbolic Bound Test	Dimensioning
Priority Exchange	$U_p \leq n\left[\left(\frac{2}{U_s+1}\right)^{1/n} - 1\right]$	$\prod_{i=1}^{n}(U_i+1) \leq \frac{2}{U_s+1}$	$U_s^{max} \leq \frac{2-P}{P}$
Sporadic	$U_p \leq n\left[\left(\frac{2}{U_s+1}\right)^{1/n} - 1\right]$	$\prod_{i=1}^{n}(U_i+1) \leq \frac{2}{U_s+1}$	$U_s^{max} \leq \frac{2-P}{P}$

2.2.2 Dynamic priority servers

Another category of server algorithms is dedicated to executing soft and firm real-time tasks in combination with a set of hard real-time tasks scheduled with a dynamic priority scheduling algorithm. Given that the optimal and most used algorithm in this category is EDF, the server algorithms presented below are designed and analysed having EDF in mind. In over to be able to schedule the task set, $U_p + U_s \leq 1$ has to hold.

Dynamic Priority Exchange Server

The dynamic priority exchange server[13] is an adaptation of the priority exchange server to a dynamic priority scheduling setting.

Dynamic Sporadic Server

A dynamic sporadic server[13] solution is the adaptation of the sporadic server to operate beside a dynamic priority scheduled task set.

Constant Utilization Server

The idea of the constant utilization server[14] is that the server task has a certain utilization U_s, and when a sporadic job arrives with an expected computation time of C_i, the server receives a relative deadline equal to $D_s = C_i/U_s$. A new sporadic job is budgeted for after the current deadline of the server passes.

Total Bandwidth Server

The total bandwidth server[15] is similar to the constant utilization server, with the exception that it allows budgeting for a new task even if the deadline of the server has not passed if the previous job was already executed. When budgeting for a new job with expected computation requirements of C_i, and having a deadline, the current deadline is extended: $D_s = D_s + C_i/U_s$

Constant Bandwidth Server

The constant utilization and total bandwidth servers rely on knowing the expected computation time of arriving sporadic jobs. However, that is not always the case. The constant bandwidth server[16] solves that by having a certain budget Q_s and a period T_s. When a new job arrives, the computation time allocated to it is set to Q_s, and the relative deadline of the server is set to T_s. If the task does not take the whole computation budget, the remainder is retained until the deadline. If the task exceeds its computation budget, the deadline is extended with T_s, as well as the computation time with Q_s. CBS is implemented and used as part of the `SCHED_DEADLINE` scheduler of the Linux kernel[17].

2.3 Interrupt analysis

An important aspect for the system executing on the driver is analysing the schedulability of the task set in the presence of interrupts, as the scheduler is sporadically interrupted by the DALI interrupt handlers.

An analysis technique for both static and dynamic priority scheduled task sets in the presence of interrupts has been proposed[18]. This technique is based on knowing the amount of work required by interrupts in a certain time interval.

There have also been other schedulability tests for tasks scheduled by EDF in the presence of a periodic high-priority task[19]. These tests may also be used for analysing interrupt overhead if the interrupt handler is modelled as a sporadic task.

2.4 Scheduling Control Tasks

An important topic is the scheduling of tasks that implement control algorithms. This is relevant because control tasks are designed to execute at regular intervals, and scheduling them with a deadline equal to their period may result in jitter even if they are scheduled correctly. There are two sources of jitter. The first one is the overhead of the algorithm, and the second one is the algorithm itself, as a job of the task may not be executed upon release but delayed in the limit of its deadline. This jitter may degrade the performance of the algorithm if not taken into account. The research in this area has been focused on either designing control laws that take the latency and jitter of the scheduled task into account or on scheduling the control task such that the influence of latency and jitter is as low as possible[20].

Executing control tasks through a constant bandwidth server has also been shown to be able to be a feasible solution that generates small latency and jitter[21].

2.5 Limited Preemption

Allowing the full preemption of tasks increases the schedulability of the system, but at the same time, it causes increased overhead due to frequent context switching. On the other side, a non-preemptive solution may not be able to schedule the tasks. A middle ground can be found with limited preemption solutions, which only allow preemption under certain conditions[22].

One possible solution is the preemption threshold algorithm. With this solution, each task has a preemption threshold, which functions as a second priority, and only tasks with a higher priority than the preemption threshold of the current task may preempt it. The second solution is deferred preemption, realized through either fixed or floating non-preemptive regions. With fixed regions, each task has predefined preemption points placed at specific points of their execution (e.g. after executing a specific function or a certain number of iterations of a loop), and higher priority tasks may only preempt the running task at these points. Floating regions do not define fixed points, but rather when a higher priority task is released, the preemption is delayed by a given amount of time, thereby giving an opportunity to the running task to complete its execution and cede the processor voluntarily.

2.6 Real Time Operating Systems

The algorithms presented before are implemented through a scheduler. There are already a number of schedulers provided as part of an operating system, and using them makes the development process faster and more convenient. In the next section, a selection of operating systems is presented, together with their features.

2.6.1 Requirements for the real-time operating system

Before presenting the possible options for an operating system, it is useful to have a feature list of what an operating system may provide. Some of the most important features are:

1. Support the definition of tasks with priorities. This is needed to enable priority-based scheduling policies.

2. Support software timers. As most of the tasks are periodic, software timers are needed as a means to implement their periodicity.
3. Offer task synchronization primitives. This is needed to implement precedence constraints. Examples of such primitives are:
   (a) Semaphores
   (b) Mutexes
   (c) Message Queues between tasks
   (d) Management for the priority inversion problem
4. Integration with interrupts
5. Support for dynamic scheduling algorithms
6. Support for server implementation

Besides the number of features implemented, an important non-functional aspect of the system is its licensing. In a final product, a closed-source operating system is preferred for quality and support reasons. However, in the exploratory phase, which is represented by this project, an open-source platform is more appropriate because the feasibility of the system can be determined without licencing fees, and having access to the source enables better profiling and analysis of the performance of the system.

2.6.2 Options

There is an array of operating systems available on the market. The RTOSes presented below all offer preemptive, static priority scheduling, together with task synchronization mechanisms, and are compatible with the XMC library.

FreeRTOS[23] is a free and open-source operating system supported by Amazon. It offers support for task definition with priorities and support for software timers on top of the kernel primitives. It also offers synchronization primitives and is compatible with the XMC library. The periodicity of tasks is supported through either executing the tasks in a timer task or by explicitly calling kernel primitives in a task and delaying it the desired period.

Dynamic priority scheduling is not supported by default in the kernel, but there have been various implementations in either the user space [24] or the kernel space[25]. Similarly, server algorithms are not supported by default, with existing implementations being available[24].

MicroC/OS[26] has two versions, MicroC/OS-II and III, both being open-source. Their kernel supports task definition with fixed priorities and offers task synchronization primitives together with software timers. The kernel is written in C, and it is compatible with the XMC library.

Dynamic priority scheduling algorithms are not supported, but EDF has been implemented by third parties[27]. However, there are comparatively fewer papers discussing implementations of various scheduling algorithms than for FreeRTOS.

KeilRTX[28] is an open-source operating system developed by ARM, targeted at the Arm Cortex-M processor architecture. It supports task definition with fixed priorities, software timers and offers task synchronization primitives.

The Linux kernel has also been used in real-time projects. However, its RAM requirements are too high to use on either the XMC1400 or 4400.

VxWorks[29] and EmbOs[30] are closed-source operating systems. Their use is limited by a commercial license, and they do not allow modification of the kernel, so there is only limited research on implementing dynamic scheduling algorithms or servers.

In addition to comparing the operating system based on their functional and non-functional requirements, there have also been studies about their performance. For instance, FreeRTOS has been shown to have a faster context switch time than KeilRTX but a slower message passing implementation[31]. However, other researchers[32] have found FreeRTOS to have the largest context switch time when the switch is triggered by task synchronization mechanisms.

2.6.3 Scheduler implementation

In all operating systems presented above, the scheduler is implemented in the process context, which allows the use of task synchronization and preemption. In contrast to that, tasks executed in an interrupt usually have a run-to-completion behaviour and preempt tasks that execute in the process context. This creates two separate priority spaces, which may not be a desirable situation if a process task needs to have its priority increased. There has been research into how interrupts can be used to implement all tasks and how they can be augmented with some preemption and synchronization primitives. One example of research in this direction is the Sleepy Sloth[33], which is an experimental kernel implementing this concept. Moreover, it was shown that this concept could also be implemented in FreeRTOS [34].

Chapter 3

System Model

This chapter will describe a model of the system. The aim is to gain insight into its requirements by exploring and defining how tasks are executed. In this context, the system represents the complete software that is executed on the microprocessor. The model is an abstraction of the system, which is decomposed into modules and tasks, as explained in this chapter. The first section presents the theoretical notions that are used. The second section goes over the software modules of the system. Then the tasks executed in the cyclic executive scheduler are presented.

3.1 Notions and Definitions

This section will establish the notions that will be used when describing the system model. As mentioned before, the system represents the complete software running on the microprocessor. It can be decomposed into modules and tasks. A module is a part of the software focused on only one area of functionality. Modules can be further decomposed into tasks.

A task is the sequence of actions that must be carried out to respond to an event. An event is a change in the system's state that requires a response. The event may be triggered by external or internal actions. The activation of a task is the moment when the event corresponding to that task is triggered. A task is defined by a name, a worst-case computation time and a worst-case deadline relative to the activation of the task.

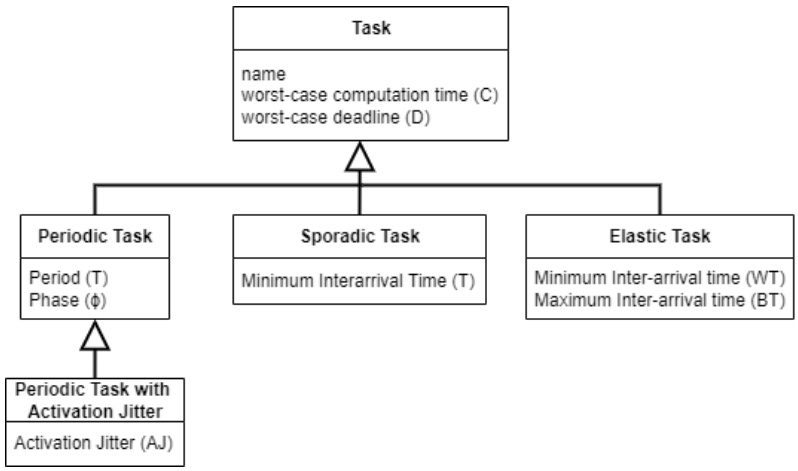


Figure 3.1: Task classification and attributes

The arrival pattern of a task is the pattern of its activations. Tasks may be classified based on their arrival patterns. This classification may be seen in Figure 3.1. Tasks that are activated at

fixed intervals are called periodic tasks. Besides the basic characteristics listed above, they have a period, which is the time between consecutive activations, and a phase, which is the time until the first activation. Periodic tasks may also have activation jitter, in which case the activation may not happen exactly at each period interval but rather in a time window after that moment. The duration of that window is the activation jitter. Alternatively, tasks may be sporadic or elastic. Sporadic tasks have a minimum inter-arrival time between consecutive activations, but after that time, they may be activated at any moment. Elastic tasks have both a minimum and maximum inter-arrival time. These different arrival patterns can be seen in figure Figure 3.2.

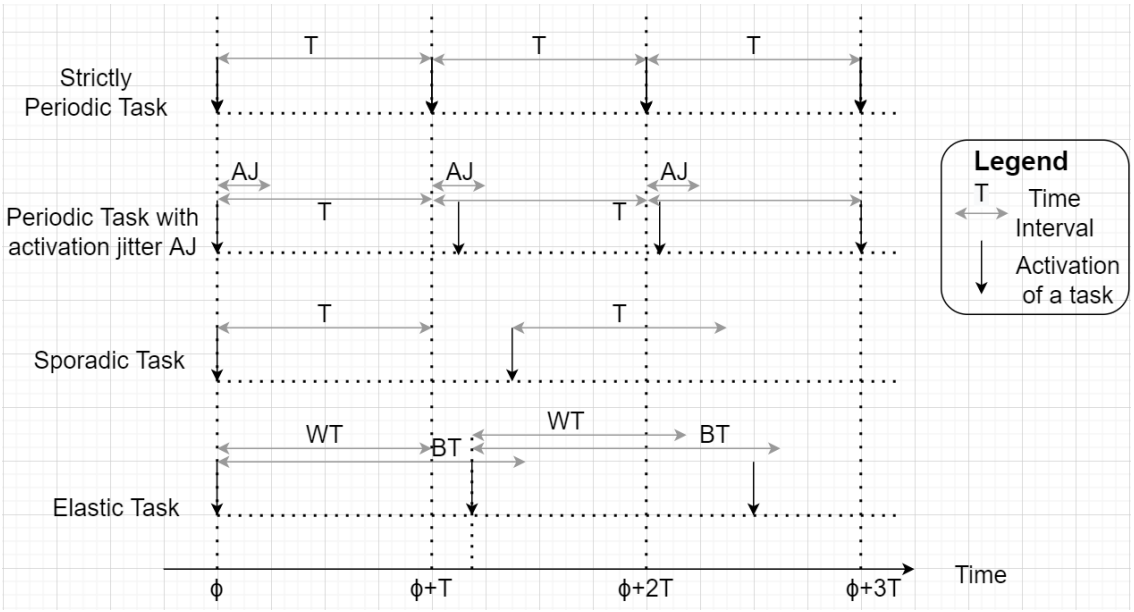


Figure 3.2: Arrival patterns visualization

At each of its activation, we say that the task must perform a job. A job is thus an instance of a task. It is defined by an activation time, an execution time, a worst-case deadline, a start time and a finalization time. The deadline can be defined with an absolute value in time or with a relative value as passed time from an event. The deadline of a job has an absolute value, while the deadline of a task is relative to its activation.

3.2 Software Modules

There are several modules in the system, and each of them decomposes into tasks. The decomposition of the system on modules is given below. Each module contains software that handles one specific functionality.

1. Mains: Monitors mains, implements zero-voltage detection and detects the frequency of the current
2. PFC: Contains the PFC controller
3. Output Power Stage (OPS): controls the buck converter.
4. Energy Metering: Handles energy metering
5. EOLL: Determines the end of life of the LED
6. Light App: Manages the control of the light

7. Gear App: Starts the driver submodules, and implements functionalities at the highest level
8. Gear Control: Manages the state of the driver at a high level
9. HAL: Manages the state of the driver's hardware
10. ACDC detect: detects whether the power supply is DC or AC
11. DALI bus: layer between the DALI driver and higher DALI logic, handles messages and events from the DALI driver
12. DALI driver: assembles the DALI messages from the edges detected on the DALI line
13. EEPROM Cache: handles the EEPROM cache and the synchronization with the NFC EEPROM
14. Sub-memory bank: Manages read and writes to certain memory banks.
15. TDC: handles the Touch-Dim Corridor protocol, an application that uses the DALI line for messages.
16. Diagnostics: Collects diagnostics data as histograms
17. Measurement: Receives input samples from various hardware signals and filters them to be used by other software modules.

Types of tasks associated with modules The software modules presented above can be further decomposed into tasks. While the specific tasks are described in the following sections, it is helpful to point out the categories of tasks with them. Tasks contained by the Mains, PFC and OPS modules directly control and monitor the electrical circuits of the driver. They are strictly periodic but do not have a defined deadline, and they must be executed as fast as possible to maintain QoS.

Tasks that record the electrical measurements sampled by the VADC unit of the XMC microcontroller are periodic tasks with hard real-time deadlines equal to their period. Measurements are used for further processing by periodic tasks with no deadlines.

The DALI communication and the TDC protocol each rely on a periodic task with a hard real-time deadline. The rest of the system's tasks are either periodic or sporadic tasks with no deadlines.

3.3 Cyclic Executive Scheduler

The first scheduler of the system is implemented using the cyclic executive pattern. It is triggered with a frequency of 12 kHz, and every time, it executes the tasks assigned to the current timeslot. The length of the cycle is 12 timeslots, repeating every millisecond. The scheduler executes tasks managing the electrical circuit and controlling the output voltage of the led driver, together with tasks recording measurements sampled by the VADC.

Table 3.1 shows the description of each task that is executed in the cyclic executive. It is important to note that PFC_vExecute and MAINS_vExecute behave differently based on the argument they are given, so they can be considered separate tasks.

Table 3.1: Tasks executed in the cyclic executive scheduler

Task ID	Task	Software Module	Description
T1	Process LED Current measurements for the buck	OPS	Handles raw current samples for the buck controller
T2	Buck Controller	OPS	Buck Control Loop

Task ID	Task	Software Module	Description
T3	Set Mains Level	Mains	Implements ZVD, and handles raw measurements on the mains voltage
T4	Execute Mains	Mains	Mains Computation task, Handles ZVD, increases a half-mains period counter generates the one second tick for gear updates the mains state
T5	Store Sample PFC	PFC	Stores a sample of the bus voltage for the PFC controller
T6	PFC Controller	PFC	Updates the state of the PFC FSM and calls its controller
T7	Process Energy Measurement	Energy Metering	Stores a raw sample for Energy Measurement
T8	Notify NTC sampled	Diagnostics	Toggles notification that driver temperature has been sampled
T9	Store measurement sample	Measurement	Stores a measurement for further processing
T10	ADC Trigger	HAL	Triggers the sampling of additional measurements in the ADC

Listing 3.1 shows how the execution of the cyclic executive scheduler proceeds. The routine is called by a periodically activated interrupt, after which the tasks allocated to the current time slot are executed. At the end of a cycle, a measurement data ready event is generated. After each time slot, the ADC is triggered, to have data ready for the next one. Note that the measurements taken when the ADC is triggered are not the ones needed for the control tasks but are used for measurement and diagnostics purposes.

```
void Interrupt_Handler(void) {
  static byte slot = 0;
  switch(slot) {
    case 0:
    <<tasks assigned to slot 0>>
    break;
    case 1:
    .
    .
    default:
    break;
  }
  if(slot < 12) {
    slot++;
  } else {
    slot = 0;
    trigger_data_ready_event();
  }
  trigger_ADC();
}
```

Listing 3.1: Cyclic Executive Scheduler workflow

The schedule of the cycle is given in Table 3.2. This schedule can be visually inspected in Figure 3.3. The figure also shows the difference between the major and the minor cycle. The major cycle is an execution of the whole schedule, while the minor cycle is the execution of the schedule in a single slot. The assignment of tasks to timeslots serves two purposes. Firstly, it enforces the periodicity of each task execution. If a task is assigned to multiple timeslots, its execution frequency increases. Hence, some of the tasks execute at a frequency of 4kHz, while others execute at 1 kHz. Secondly, the assignment takes into account dependencies between tasks, ordering their execution temporally such that dependencies are satisfied. The real-time requirement of the tasks executed by this scheduler is that they execute within the allocated timeslot.

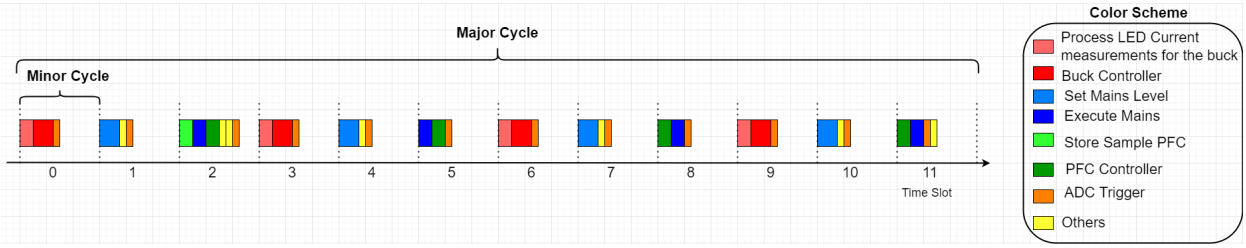


Figure 3.3: Visualization of the schedule executed by the cyclic executive

Table 3.2: Schedule of the cyclic executive scheduler

Time Slot	Task	TaskID	Dependency
0	Process LED Current Measurements for the buck	T1	-
	Buck Controller	T2	T1
	ADC Trigger	T10	-
1	Set Mains Level	T3	-
	Process Energy Measurement	T7	-
	ADC Trigger	T10	-
2	Store Sample PFC	T5	-
	Execute Mains	T4	T3
	PFC Controller	T6	T4
	Notify NTC sampled	T8	-
	Store measurement sample	T9	-
	ADC Trigger	T10	-
3	Process LED Current Measurements for the buck	T1	-
	Buck Controller	T2	T1
	ADC Trigger	T10	-
4	Set Mains Level	T3	-
	Process Energy Measurement	T7	-
	ADC Trigger	T10	-
5	Execute Mains	T4	T3
	PFC Controller	T6	-
	ADC Trigger	T10	-
6	Process LED Current Measurements for the buck	T1	-
	Buck Controller	T2	T1
	ADC Trigger	T10	-
7	Set Mains Level	T3	-
	Process Energy Measurement	T7	-
	ADC Trigger	T10	-
8	PFC Controller	T6	-
	Execute Mains	T4	T3
	ADC Trigger	T10	-
9	Process LED Current Measurements for the buck	T1	-
	Buck Controller	T2	T1
	ADC Trigger	T10	-

Time Slot	Task	TaskID	Dependency
10	Set Mains Level	T3	-
	Process Energy Measurement	T7	-
	ADC Trigger	T10	-
11	Execute Mains	T4	T3
	PFC Controller	T6	-
	ADC Trigger	T10	-

3.4 Cooperative scheduler

The cooperative scheduler executes in the main loop of the program. The scheduler employs a list of functions, events and timers. Its functions represent tasks and are connected to events. Events may be triggered from other tasks or by timers. Timers are connected to events. The resolution of the timers is one us. The loop of the scheduler checks the system time and updates the timers. It then iterates through the list of events, and for each triggered event, it calls the function connected to it.

The cooperative scheduler is designed to execute tasks that are not time-critical and can be scheduled in the background of the cyclic executive scheduler. A notable example of tasks assigned to it is the DALI communication stack. The DALI interrupt, which interferes with the cyclic executive scheduler, receives incoming edges and stores them in a queue, and when transmitting, it triggers at a constant interval to set the line to the corresponding value. Decoding and encoding edges are handled by a task executed in the cooperative scheduler. The interpretation of the assembled messages at the application layer is implemented by another background task triggered by the arrival of a message.

3.5 DALI timing requirements

To understand how the DALI protocol may be scheduled, its timing requirements need to be presented. First, the DALI stack will be presented, together with the behaviour of the interrupt that interferes with the control tasks. Then, higher-level timing requirements will be presented to determine the worst-case arrival patterns when the driver is receiving and transmitting a message.

The DALI stack can be visualized in Figure 3.4. At the software level, it is composed of 3 levels. The lowest level is represented by the DALI interrupt. The interrupt is triggered by a capture-and-compare unit, and it behaves differently based on whether the driver is receiving a message or transmitting it. When receiving, it reads captured edges and stores them in a queue. When transmitting, the CCU functions as a timer, and the interrupt is triggered when a new edge needs to be transmitted. From a scheduling point of view, the interrupt arrives sporadically, in bursts. It has a bound execution time and a higher priority than the cyclic executive interrupt, being able to preempt it. The higher level consists of a decoder that filters spikes and assembles messages. Then the application layer interprets the messages and responds to them.

The baud rate of the protocol is 2400 baud, and the bit rate is 1200 bps, as each bit is transmitted through two half-bit periods in which the line is either high or low. A 1 is encoded by a low half-bit followed by a high one, and a 0 is encoded by a high half-bit followed by a low one. The specified baud rate results in a half-bit period equal to $Te = 416.67\mu s$.

3.5.1 Reception analysis

When the driver receives a message, the CCU4 unit functions in capture mode, capturing the value of the timer when receiving a falling or rising edge. The edges are captured in pairs. The first one is the falling edges. The next edge will be a rising edge, and when it is received, the value of the timer is captured, and the timer is reset. When the timer is reset, the DALI interrupt triggers. It collects the captured values and stores them in a buffer for further processing. The CCU4 is

configured to only capture an edge if its level remains stable for at least 9.33 us, effectively making the minimum interarrival time of the DALI interrupt 18.66 us. The queue in which the edges are stored has a length of 15. If the queue is full when the interrupt attempts to write in it, it will enter an error state of at least 1400 us.

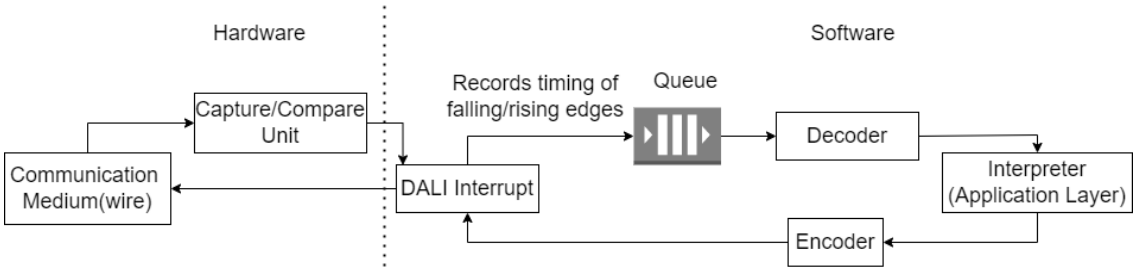


Figure 3.4: Different components of the DALI stack

At the next layer, the DALI decoder picks up the stored edges with a periodic task with a period of 1ms. The base unit of the protocol is the half-bit period, as the protocol ensures that there is a change in the voltage level of the line at least every bit period, and this guaranteed change happens in the middle of the bit. Then, an additional change may happen at the bit boundaries, if necessary. More concrete, the change happens if the next bit is the same as the previous one. The baud rate of the protocol is 2400 baud, with a bit rate is 1200 bps. A 1 is encoded by a low half-bit followed by a high one, and a 0 is encoded by a high half-bit followed by a low one. The specification sets a typical half-bit period to $Te = 416.7\mu s$, but allows it to be in the interval $333.3us - 500us$.

The decoder has two stages. The first stage is a filter for short spikes that may appear on the line. A spike is defined as two consecutive edges that are closer than a maximum time interval, which is at least $50\mu s$ according to the specification. The decoder should filter two types of spikes, single spikes that happen during a half-bit interval and double spikes found at half-bit boundaries. The second stage of the decoder assembles the DALI message from filtered edges, and if there are any abnormalities, such as edges with an unexpected length, it drives the receiver into an error state lasting at least 1400 us.

On a higher level, a forward frame has a length of 16 bits. Because the line is set to high while idling, the start bit is always a 1, making the line low at the beginning of the message. Then, the 16 data bits follow, appended by the stop condition, which is a period of where the line is high at the end of the message. Therefore the message has a length of $33Te$ or $34Te$, depending on whether there is a change on the last bit boundary. After that, there is a settling time between 2.4ms and 12.4 ms until an eventual backward frame, when the driver will be transmitting.

The maximum arrival rate will happen in a scenario where the receiver experiences the maximum amount of interrupts that do not drive it into an error, followed by entering the error state through interrupts happening as fast as possible. For single spikes, the first interval between edges may be as short as possible(limited at 9.33us), but the second one should be larger than the spike interval, so at least 50 us. This corresponds to an interrupt happening every 59.33 us. A double spike may happen at the half-bit boundary, but after that, the next edge is expected to be at least a minimum half-bit, so the worst-case scenario happens when there are single continuous spikes during a half-bit. The arrival pattern, in this case, can be seen in Figure 3.5. A second bottleneck for a continuous worst-case receiver is the queue that stores the DALI edges. If this queue fills up, the receiver will stop. The queue has a length of 15, and it is emptied every ms. As there an interrupt stores two edges at a time, the queue limits the interrupt inter-arrival time to $1000/7 = 142.85us$.

The second case of the worst-case scenario happens when edges are received at normal intervals, followed by a burst of edges on the line. The burst continues until the queue storing edges fills up, so until there are 15 stored edges. As there an interrupt stores two edges at a time, to fill up the queue would take $18.66 * 8 = 149.28us$.

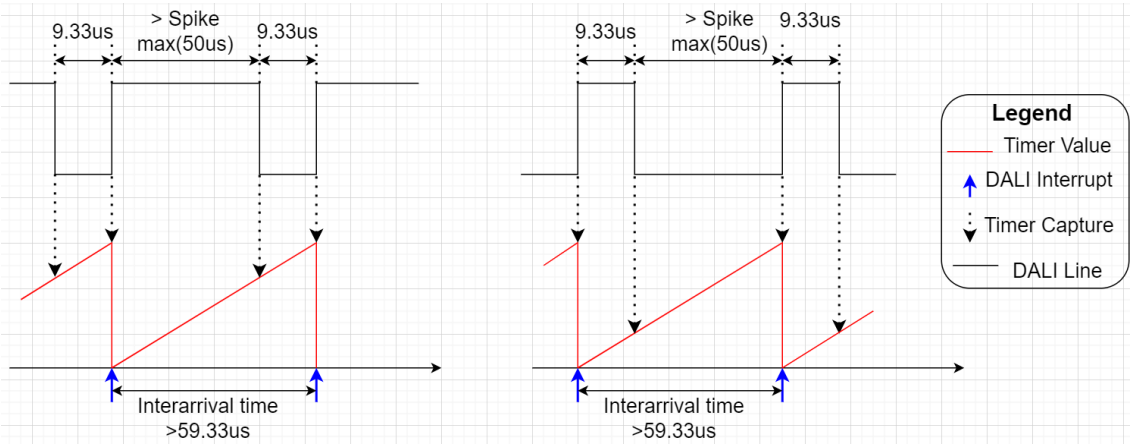


Figure 3.5: Visualization of the worst-case interarrival time for the DALI interrupt when receiving

3.5.2 Transmission analysis

When transmitting, the implementation uses the same CCU unit to trigger a timer every half-bit period to set the line to the appropriate level. The window for a correct transmission is within $[366.7, 466.7]us$. The size of this window is larger than one slot of the cyclic executive scheduler, i.e. $83.33us$, and given that no task takes the whole slot, the transmission interrupt can be scheduled in the background under the initial implementation.

3.6 Reduced System Model

While the full system model is useful for deriving requirements, a subset of the tasks will be used in the analysis so that it is more manageable while retaining the real-time characteristics. The subset that was selected consists of the control-related tasks, the tasks associated with the measurement module, the communication protocol, and the light app, which converts dimming values received through DALI into reference values for the buck controller.

Deadlines are set equal to periods in this model to get the first schedulability result. The impact of the tightness of deadlines will be analysed later. The DALI receiver, transmitter and interpreter tasks are exceptions. The deadline of the receiver is set to a value close to their execution time to be executed as fast as possible. The transmission deadline corresponds to the window of time allowed by the protocol. The interpreter task deadline is set according to the specification of the protocol and is discussed in section 3.5.

The computation time is an estimated worst-case execution time. The estimation is based on measurements taken by executing the tasks on a Duvel 90W S 21mm driver, with software version 0.1.0.21842 and hardware version 0.1. The measurements were taken by setting a GPIO pin to high just before the task started to execute and setting it low just after the task finished executing. The pin voltage level was measured using a Teledyne Lecroy waverunner 8054 500 MHz 20GS/s oscilloscope.

Table 3.3: Reduced System Model

Task	Type	Period/Minimum Interarrival time	Computation Times	Deadline
Buck Controller	Periodic	250 us	20 us	250 us
Set Mains Level	Periodic	250 us	4.65 us	250 us
Execute Mains	Periodic	250 us	13 us	250 us
Continuation on next page				

Task	Type	Period/Minimum Interarrival time	Computation Times	Deadline
Store Sample PFC	Periodic	1ms	6.23 us	1 ms
PFC Controller	Periodic	250 us	28 us	250 us
Store Sample Measurement	Periodic	1 ms	8.26 us	1 ms
Measurement Task	Periodic	1 ms	95 us	1 ms
DALI Interrupt (Reception)	Sporadic	142.85 us	7.35 us	18.66 us
DALI Interrupt (Transmission)	Sporadic	416 us	3.52 us	100us
DALI Decoder	Periodic	1 ms	26.65 us	1 ms
DALI Interpreter	Sporadic	43.33 ms	6.6 us	5ms
Lightapp Task	Periodic	5 ms	3.42 us	5 ms

3.7 Summary of the system model

There are multiple types of tasks in the system. The first one is strictly periodic tasks with deadlines equal to their period. Another category consists of tasks that are periodic and have to be executed with reduced jitter but do not have a determined deadline. There are also sporadic communication tasks with hard deadlines dictated by the constraints of the DALI protocol. Moreover, there are a number of dependencies between different tasks.

Chapter 4

Theoretical analysis

This chapter presents a theoretical analysis of the existing system, together with potential solutions to the interference between control tasks and the communication driver. Three potential solutions are considered: using a priority-based scheduling algorithm, using a server to control the firing rate of the communication driver, and limiting the preemption of control tasks when the communication driver has to execute.

4.1 Initial system

The initial system is the current existing implementation of the driver's tasks and how they are scheduled. More precisely, this analysis will look into the properties of the cyclic executive scheduler and of the DALI interrupt, as they are tasks with real-time requirements.

The first part of the chapter will be focused on the initial system to establish a baseline for comparison. The analysis will present the worst-case response time of tasks of interest. The response time is considered to be the time elapsed from the activation of a task to its finalization. The activation of a task is defined differently for sporadic and periodic tasks. For sporadic tasks, it is the time of the event that triggers it. For periodic tasks, it is at the beginning of the current period, the moment in which the task becomes available to be executed. The finalization time is the end of its computation time for both types of tasks.

The analysis is focused on the worst-case response time. Another interesting measure is the response time jitter of tasks, which is the difference between the best-case and worst-case response time. For the initial system, the best-case response time is the execution time of each task. Given that they do not overrun their slots, tasks in the cyclic executive scheduler do not interfere with each other, and in the best-case, the DALI interrupt does not trigger. As the DALI interrupt has the highest priority, when it is triggered, it will immediately begin to execute.

For the worst-case response time analysis, the DALI interrupt will be treated as a sporadic task, with a worst-case execution time of $C_D = 7.5us$, a minimum inter-arrival time of $T_D = 18.66$, and a deadline equal to its period. Its worst-case response time is equal to its best case, $C_D = 7.5us$.

Tasks of the cyclic executive scheduler only suffer interference from the DALI interrupt in the worst case. They are interleaved by the scheduler, and as long as they do not overrun their assigned slot, they do not interfere with each other. This is indeed the case, as the worst-case interference of the DALI interrupt on a slot of $T_S = 83.33us$ is equal to the number of times the DALI interrupt may trigger in that time interval multiplied by the execution time of the interrupt. This amounts to $\lceil \frac{T_S}{T_D} \rceil * C_D = \lceil \frac{83.33}{18.66} \rceil * 7.5 = 37.5us$. This means that as long as the total execution time of the tasks assigned to a slot is shorter than $83.33 - 37.7 = 45.83us$, it will not overrun to the next slot.

The worst-case execution time for a task with computation time C_I is equal to C_I plus the interference from the DALI interrupt. The worst case happens when the DALI interrupt starts to execute upon the activation of the task. This can be visualized in Figure 4.1. The interference is the

maximum number of DALI interrupts that happen in that interval multiplied by its computation time. The task will be executed in the background of the DALI interrupt. The task will execute in chunks of $T_D - C_D$, needing $\frac{C_I}{T_D - C_D}$ of these chunks to finish executing. The number of DALI interrupt executions is, therefore, the ceiling of the number of chunks the task needs.

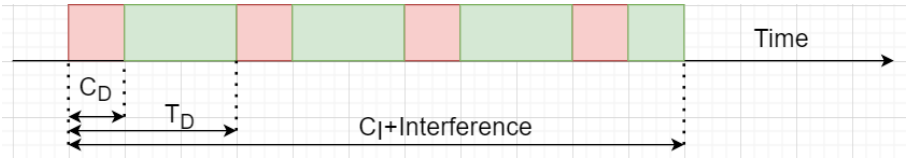


Figure 4.1: Visualization of the worst-case response time for a slot

The exact formulas can be found in Table 4.1. The application of the formulas to the tasks scheduled in the cyclic executive scheduler that are present in the reduced system model can be seen in Table 4.2.

Task	Best-case response time	Worst-case response time
DALI	C_D	C_D
Control	C_i	$C_i + \left\lceil \frac{C_i}{T_D - C_D} \right\rceil * C_D$

Table 4.1: Response time formulas for the initial system

Task	Best-case response time(us)	Worst-case response time(us)	Response time jitter(us)
Buck Controller	20	34.7	14.7
Set Mains Level	4.65	12.0	7.35
Execute Mains	13	27.7	14.7
Store Sample PFC	6.23	13.58	7.35
PFC Controller	28	50.05	22.05
Store Sample Measurement	8.26	15.61	7.35

Table 4.2: Response times for the initial system

4.2 Proposed solutions

The analysis presented in section 4.1 considers the original system. Under the original implementation, the DALI interrupt is executed as soon as it is triggered. However, the DALI interrupt does not need to be executed immediately. Its deadline is larger than its computation time. This is true in its worst-case activation pattern, which occurs when the driver is receiving a message. It is also true when the driver is transmitting, and in that case, the deadline is even more relaxed. Therefore, there is room to reduce the interference suffered by the control tasks.

Three different solutions will be studied. The first solution is changing the underlying scheduling algorithm from using a cyclic executive to using a priority-based scheduling algorithm. Using a priority-based algorithm has two potential benefits. The first one is that the difference between the deadline and the computation time of the DALI interrupt would be exploited inherently. The second benefit is that it improves non-functional aspects of the system, such as maintainability and extendability of the system because adding new tasks or modifying existing tasks would not

require generating a new schedule for the cyclic executive. The new schedule would be created automatically by the scheduling algorithm at runtime.

Priority-based scheduling algorithms can assign priorities in two ways: static or dynamic. A static assignment means that priorities are assigned before the scheduler begins to execute and remain fixed. Therefore, a task with a high priority will always be allowed to preempt a lower priority task. A dynamic assignment means that tasks change their priority during execution based on a chosen criteria. One algorithm from each category will be evaluated as a potential solution to the interference between the DALI interrupt and control tasks.

Static priority algorithms have the advantage of being simple to implement. Because of this, they are supported in multiple operating systems. Furthermore, they have a smaller scheduling overhead compared to dynamic priority assignments. This is due to the fact that their queue of ready tasks is based on the assigned static priorities. Thus, they can select the highest priority task with an associated complexity of O(1).

On the other hand, the queue for dynamic priority algorithms needs to keep track of a runtime attribute of the tasks. To select the highest priority task, the associated priority is O(log(n)), where n is the number of tasks. This makes their implementation more involved, and therefore there is limited support for them in operating systems. However, they generate fewer context switches compared to static priority algorithms. Moreover, they can cope with higher processor utilization.

The second proposed solution is the use of a server to manage the DALI interrupt. This solution has the advantage of being independent of the underlying scheduling algorithm. This means that it will be compatible with the existing cyclic executive, as well as any potential future alternative. Regarding the scheduling of the DALI interrupt, the server will control the amount of interference of the interrupt suffered by control tasks.

The third proposed solution is the use of limited preemption for the control tasks. That is, the DALI interrupt will preempt control tasks only if the control task cannot complete its execution within a predefined amount of time. This solution exploits the longer deadline of the DALI interrupt compared to its computation time by delaying the execution of the interrupt until a convenient moment. This has the potential of lowering the interference of the DALI interrupt on control tasks, while ensuring that the timing requirements of the DALI protocol are met.

4.3 Deadline-Monotonic Scheduling

Deadline monotonic scheduling is a static priority scheduling algorithm. This means that the priority assigned to tasks remains fixed during the execution of the scheduler. The deadline monotonic algorithm assigns higher priorities to tasks having shorter deadlines. Taking the reduced system model yields the priority assignment from Table 4.3. The computation time and the deadline of the DALI interrupt is considered to be the ones of its worst-case arrival pattern, $C_D = 7.5us$ and $T_D = 18.66us$. Each task needs to have a unique priority assignment. Therefore, tasks that have the same deadline are enumerated on different levels. The assignment used in the analysis can be seen in Table 4.4.

4.3.1 Analysis

In this section, we apply several schedulability tests to the task set. Schedulability tests are used to determine whether the task set can be scheduled by the deadline monotonic scheduling algorithm. Moreover, the response time jitter of the DALI interrupt and of the control tasks is also of interest. It will determine whether this algorithm can ameliorate the interference suffered by the control tasks. We use two different tests, a utilization test and a response time test.

Utilization test

The first schedulability test is the utilization test. The test assesses whether there is enough computation time available for the tasks to finish their execution before their deadlines. The test

Priority	Tasks	Deadline (us)
0	DALI Interrupt	18.66
1	Buck Controller Set Mains Level Execute Mains PFC Controller	250
2	Store Sample PFC Store Sample Measurement Measurement Task DALI Decoder	1000
3	DALI Interpreter Lightapp Task	5000

Table 4.3: Deadline monotonic priority assignment

Priority	Task	Deadline(us)
0	DALI interrupt	18.66
1	Buck Controller	250
2	Set Mains Level	250
3	Execute Mains	250
4	PFC Controller	250
5	Store Sample PFC	1000
6	Store Sample Measurement	1000
7	Measurement Task	1000
8	DALI Decoder	1000
9	DALI Interpreter	5000
10	Lightapp Task	5000

Table 4.4: Deadline monotonic analysed priority assignment

is sufficient yet not necessary. This means that a successful test guarantees the schedulability of the task set, but failing the test does not mean that the task set is not schedulable. The test consists of evaluating the expression given below[3]:

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{1/n} - 1), \forall \text{ tasks } \tau_i$$

For the analysed task set, this evaluates to $0.79463 \leq 0.713557132$. As this is false, the test fails. The test was established by Liu-Layland for the rate monotonic scheduling algorithm. The rate monotonic algorithm assigns higher priorities to tasks that have a shorter period. In its original form, the denominator in the sum is the period of the task. Its adaptation for the deadline monotonic algorithm uses the deadline of the task instead. This makes the test pessimistic because tasks that have a deadline shorter than their period are represented by a higher utilization than they actually have.

Response time test

The second schedulability test is a response time analysis. For this test, both the worst-case and best-case response time of each task is computed. Then the worst-case response time is compared to the deadline of the task. If it is higher than the deadline, the test fails. The best-case response time is computed to be able to determine the tasks' response time jitter. This gives a way to evaluate the solution against the current implementation. The test is sufficient and necessary, as it uses all attributes of the tasks, such as their deadlines, priorities, and computation time.

The worst-case response time occurs in a critical instance. The critical instance for deadline monotonic is when all tasks are released at the same time. The worst-case response time of a task is composed of its computation time and the amount of interference it suffers from higher priority tasks. The interference from a task is equal to the number of jobs the task has multiplied by its computation time. Then the sum over all higher priority tasks is the total interference. The number of jobs a task with period T_i releases in an interval t is equal to $\lceil \frac{t}{T_i} \rceil$.

The DALI interrupt has the highest priority with deadline monotonic priority assignment. It will not suffer any interference in this scenario. In the actual implementation, the DALI interrupt has different behaviour depending on the state of the communication protocol. Therefore, its worst-case interference on the other tasks can be computed in a way that is more close to reality.

Its worst-case computation time is given when the task is executed at its minimum inter-arrival time. The implementation of the protocol stops its execution for a set amount of time upon high utilization, to limit its interference on other tasks. Therefore, the maximum interference in a set

period of time is achieved when the interrupt executes as often as possible without triggering the stop of the protocol. This is followed by triggering it as fast as possible at the end, until it is stopped.

In subsection 3.5.1, it is shown that when receiving, the protocol can receive up to 7 DALI interrupts in a ms. Then, in the last ms, the protocol will stop upon the 8th interrupt. These constraints are given by the queue used to store edges for further processing. Two cases can be distinguished here. If the time interval is shorter than a forward frame, the end of the interval will feature interrupts happening as fast as possible. When the queue is filled, the interrupt is disabled. If the time interval is longer, then the reception should not trigger an error state to allow a backward frame to be transmitted. When transmitting, the DALI interrupt is considered to happen once every Te. Maximum interference happens while receiving. Thus, a message starts to be received at the beginning of a critical instant. The algorithm used for computing the DALI interference is shown in Listing 4.1. It returns the interference that happens up from the beginning of a message until a time t.

```
DALI_interference(t):
  I = 0
  forward_frame_end = 34 * Te_max
  backward_frame_start = forward_frame + settling_forward_backward
  backwad_frame_end = backward_frame_start + 18 * Te
  period = backwad_frame_end + settling_backward_forward

  while (t >= 0) {
    if (t <= forward_frame_end) {
      I = I + ⌈t/1000⌉ * 7 * C_DALI_reception
      I = I + min(⌊(t−⌊t/1000⌋*1000)/18.66⌋, 8) * C_DALI_reception
    } else {
      I = I + ⌈forward_frame_end/1000 * 7⌉ * C_DALI_reception
    }
    I = I + ⌈min(max(0,t−backward_frame_start),backwad_frame_end)/Te⌉ * C_DALI_transmission
    t = t - period
  }
  return interference
```

Listing 4.1: DALI interference computation

The worst-case response time of tasks can be computed using the algorithm shown in Listing 4.2. The computation is an iterative process. The worst-case response time is initialized with the smallest possible value. The smallest value is the computation time of the task, as it cannot be completed faster than that. Then the interference from higher priority tasks is added to the computation time. The sum is considered to be the new worst-case response time. After that, the process is repeated until the worst-case response time does not increase anymore.

```
compute_wc_response_times(Γ) {
  response_times = []
  for each τ_i ∈ Γ {
    wr = C_i
    wr_next = C_i + DALI_interference(wr) + Σ_{j=0}^{i−1} ⌈wr/T_j⌉ C_j
    while(wr ≠ wr_next) {
      wr = wr_next
      wr_next = C_i + DALI_interference(wr) + Σ_{j=0}^{i−1} ⌈wr/T_j⌉ C_j

      if (wr_next > D_i)
        break
    }
    response_times.append(wr)
  }
}
```

Listing 4.2: Worst-case response time computation

```
compute_bc_response_times(Γ) {
  response_times = []
  for each τ_i ∈ Γ {
    br = wr(τ_i)
    br_next = C_i + Σ_{j=0}^{i−1} (⌈br/T_j⌉ − 1) C_j
    while(br ≠ br_next) {
      br = br_next
      br_next = C_i + Σ_{j=0}^{i−1} (⌈br/T_j⌉ − 1) C_j

    }
    response_times.append(br)
  }
}
```

Listing 4.3: Best-case response time computation

For the best-case response time computation, an optimal instant is considered. The optimal instant occurs when the task suffers minimal interference from the other tasks. The optimal instant occurs when the task is executed just before the simultaneous release of the other tasks. The best-case interference of a sporadic task is when it does not trigger. Thus, the DALI interrupt is considered to not happen in the optimal instant.

The computation of the best-case response time of a task is iterative. The best-case response time is initialized with the largest response time a task can have. Thus, the initial value is the worst-case response time. The next value is computed by summing the computation time of the task with the interference of higher priority tasks for the current best-case response time. As the starting point is the worst-case response time, and there is less interference in the optimal instant, the current best-case response time will decrease. The computation is stopped when the response time is not decreasing anymore. The algorithm can be seen in Listing 4.3.

The results of the analysis can be seen in table 4.5. It can be seen that all tasks meet their deadlines. However, the jitter suffered by control tasks is higher compared to the initial implementation. Therefore, using the deadline monotonic scheduling algorithm does not improve the original system scheduling solution. This is mainly due to the short deadline of the DALI interrupt, which makes it the highest priority task under the DM scheduling algorithm.

Task	Computation Time	Best case response time	Worst case response time	Response time jitter	Deadline
Buck Controller	20	20	108.2	88.2	250
Set Mains Level	4.65	4.65	120.2	115.55	250
Execute Mains	13	13.0	140.55	127.55	250
PFC Controller	28	28.0	175.9	147.9	250
Store Sample PFC	6.23	6.23	182.13	175.9	1000
Store Sample Measurement	8.26	8.26	190.39	182.13	1000
Measurement Task	95	95.0	351.04	256.04	1000
DALI Decoder	26.65	26.65	377.69	351.04	1000
DALI Interpreter	6.6	6.6	384.29	377.69	5000
Lightapp Task	3.42	3.42	387.71	384.29	5000

Table 4.5: Response times(us)

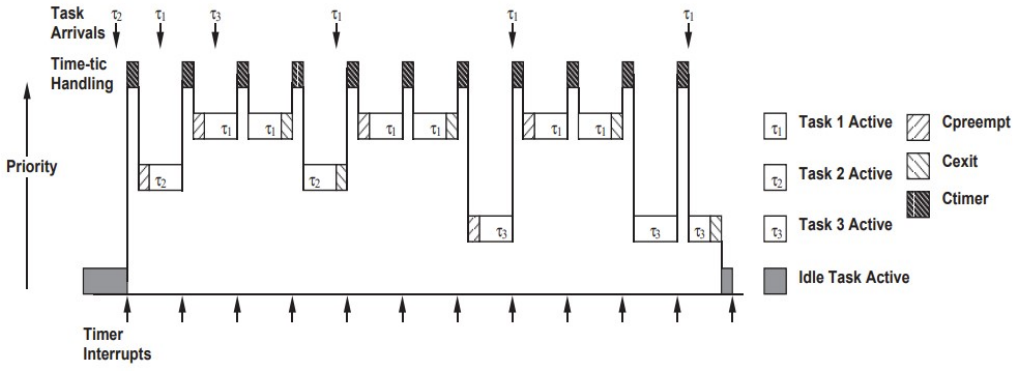


Figure 4.2: RTOS overhead[35]

4.3.2 Analysing the overhead of the RTOS

It has been noticed that the deadline-monotonic priority assignment produces more jitter than the current implementation. At the same time, the tasks meet their deadlines. A priority-based algorithm has some secondary advantages related to non-functional attributes. Therefore, it may

be advantageous to use if those non-functional attributes are important or desirable and the control tasks can cope with a higher jitter. The next step in the analysis is investigating if the task set is schedulable in the presence of a real-time operating system (RTOS).

The overhead of the RTOS can be included in the theoretical analysis[35]. The overhead is composed of 3 elements. The operating system model that is used is shown in Figure 4.2. It uses a time-triggered function to keep track of executing tasks, called the RTOS tick. When a task preempts another task, it generates some overhead due to the context switch. Moreover, when the task finishes its execution, it needs to reschedule itself to trigger at its next period.

Table 4.6: RTOS overhead

Component	Overhead
RTOS tick	4 us
Preemption	13 us
Task exit	15 us

To estimate the overhead of the RTOS, the overhead of FreeRTOS on an Arm Cortex-M0 processor of the driver has been measured, and can be seen in Table 4.6. The RTOS tick has a variable execution time, depending on the number of tasks that are activated at that tick. To simplify the analysis, an average value will be used in this analysis. The period between OS ticks is $T_{tic} = 83.33us$.

To determine whether the task set remains schedulable in the presence of an operating system, the response time analysis is redone. The algorithm used to compute the worst-case response time taking into account the RTOS overhead is shown in Listing 4.4. It is similar to the original one, but it adds the RTOS tick overhead to the interference suffered by tasks, as well as the preemption and exit overhead to the computation time of the task. The terms representing the overhead are coloured in red.

```
compute_wc_response_times(Γ) {
  response_times = []
  for each τ_i ∈ Γ {
    wr = C_i + C_preempt
    wr_next = C_i + C_preempt + DALI_interference(wr) + ⌈wr/T_tic⌉ * C_timer + Σ_{j=0}^{i-1} ⌈wr/T_j⌉ (C_j + C_preempt + C_exit)
    while(wr ≠ wr_next) {
      wr = wr_next
      wr_next = C_i + C_preempt + DALI_interference(wr) + ⌈wr/T_tic⌉ * C_timer + Σ_{j=0}^{i-1} ⌈wr/T_j⌉ (C_j + C_preempt + C_exit)
      if (wr_next > D_i)
        break
    }
    response_times.append(wr)
  }
}
```

Listing 4.4: Response Time Analysis Algorithm, taking into account the RTOS overhead

The analysis with the RTOS overhead shows that only the `Buck Controller`, `Set Mains Level`, `Execute Mains`, `Store Sample PFC` and `Store Sample Measurement` tasks meet their deadlines. Therefore the task set may not be schedulable with an operating system, when using deadline monotonic scheduling.

4.4 Earliest Deadline First

The second priority-based algorithm that is analysed is earliest deadline first(EDF). As a priority-based algorithm, it assigns priorities to tasks. In contrast to deadline monotonic scheduling, it is a dynamic priority algorithm. This means that priorities change during the execution of the scheduler. The algorithm assigns higher priorities to tasks with earliest deadline.

The algorithm is considered an improvement to the initial solution because it takes advantage of the slack of the DALI interrupt. Furthermore, it is more flexible than deadline monotonic, as the DALI interrupt will not have the higher priority at all times, only when it is the task with the earliest deadline.

For the analysis of the task set, there are two aspects of interest. The first one is whether the task set is schedulable with EDF. This means that the tasks scheduled with EDF would be able to

meet their deadlines. The second point of interest is the response time of tasks. This is important because it enables a comparison with the original implementation.

We use two analyses for EDF scheduling. The first one uses processor demand as a means to evaluate the task set's schedulability. The second one is the computation of the worst-case response time of tasks. Then it is compared to tasks' deadlines.

4.4.1 Processor Demand

The first analysis is conducted using processor demand[3]. More precisely, what it is checked is that at no point, the tasks demand more computation time than is actually available. To determine this, a function to compute the maximum demand is used. This function is called the demand bound function. It returns the processor demand up until a time t. This function can be seen below. The analysis assumes that tasks are released synchronously, at $t = 0$.

$$dbt(t) \stackrel{\text{def}}{=} \sum_{i=1}^{n} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor * C_i$$

The task set analysed for EDF is shown in Table 3.3. It is considered that the DALI interrupt is receiving a message, because that is when it has the highest utilization. Because tasks' priorities are dynamic, it is not known when preemptions are happening. Therefore, the more realistic DALI interference function used in the analysis of deadline monotonic cannot be used in this case.

A task set is schedulable if $\forall t \geq 0$, $dbf(t) \leq t$. This means that the computation time demand is not higher than the actual time available. The formula is required to be true at all points in time. However, the test points can be reduced to the set of points in time when tasks have their absolute deadline. Those are the points when tasks' demand changes. The set of test points is extended up to the point when tasks repeat their behaviour.

The formula for the set of test points is[3]:

$$D = \{d | d = D_i + j * T_i, \tau_i \in \Gamma, j \in N \& d < min(H, max(D_i, L^*)\}, \text{ where}$$

$$L^* = \frac{\sum_{i=1}^{n}(T_i - D_i)U_i}{1 - U}, U_i = \frac{C_i}{T_i}, U = \sum_{i=1}^{n} U_i \text{ and } H = \text{greatest common multiple of } T_i, \forall \tau_i \in \Gamma$$

For the analysed task set, $L^* = 22.72$, H is at least 43330, and the maximum deadline is 5000. Therefore, test points are the set of absolute deadlines smaller than 5000.

The results of the analysis can be seen in Table 4.7. It can be seen that at all test points, the demand bound function evaluates to less than t. This means that tasks do not demand more computation time than is available.

4.4.2 Worst case response time analysis

The other analysis of interest is the response time analysis. The first step is determining the worst-case computation time for all tasks. The worst-case response time computation is based on [10]. The analysis assumes that the worst-case response time of a task will appear when the other tasks have a busy period. A busy period means that they will keep the processor as busy as possible. The busy period is found when all tasks are released at the same time. However, the worst-case computation time does not happen when it is released at the same time as the other tasks. Due to the fact that EDF assigns a higher priority to tasks that have earliest deadlines, a task may suffer a higher interference when it is released after a bit of time since the simultaneous release of the other tasks. The analysis tests different points in the busy period to determine what is the worst-case computation time of the tasks. There is no exact best-case response time computation algorithm under EDF. However, the computation time of tasks may be used as an approximation for it.

The results of the analysis are given in Table 4.8. The analysis confirms that the task set is schedulable. However, it leads to higher interference than the current implementation. As EDF

Table 4.7: EDF Analysis Results

t	$dbt(t)$	$dbt(t) < t$
18.66	7.35	True
161.66	7.35	True
250	80.35	True
304.66	87.7	True
447.66	95.05	True
500	160.7	True
590.66	168.05	True
733.66	175.4	True
750	241.05	True
876.66	248.4	True
1000	450.19	True
1019.66	450.19	True
1162.66	464.89	True
1250	530.54	True
1305.66	537.89	True
1448.66	545.24	True
1500	610.89	True
1591.66	618.24	True

t	$dbt(t)$	$dbt(t) < t$
1734.66	625.59	True
1750	691.24	True
1877.66	698.59	True
2000	900.38	True
2020.66	907.73	True
2163.66	915.08	True
2250	980.73	True
2306.66	988.08	True
2449.66	995.43	True
2500	1061.08	True
2592.66	1068.43	True
2735.66	1075.78	True
2750	1141.43	True
2878.66	1148.78	True
3000	1350.57	True
3021.66	1357.92	True
3164.66	1365.27	True
3250	1430.92	True

t	$dbt(t)$	$dbt(t) < t$
3307.66	1438.27	True
3450.66	1445.62	True
3500	1511.27	True
3593.66	1518.62	True
3736.66	1525.97	True
3750	1591.62	True
3879.66	1598.97	True
4000	1800.76	True
4022.66	1808.11	True
4165.66	1815.46	True
4250	1881.11	True
4308.66	1888.46	True
4451.66	1895.81	True
4500	1961.46	True
4594.66	1968.81	True
4737.66	1976.16	True
4750	2041.81	True
4880.66	2049.16	True

assigns priorities to earliest deadlines, the deadlines of the tasks have a great influence in the amount of interference the tasks have. Thus, it may be interesting to explore assigning more constrained deadlines for tasks involved in the control functionality. Unfortunately, constraining the deadlines of these tasks does not improve their worst-case response time. Their deadlines would still be between the one of the DALI interrupt and of the tasks with deadlines of 1000 or larger. This makes the relative priorities of tasks the same. Thus constraining deadlines does not reduce the worst-case response time.

Table 4.8: Response times when using EDF

Task	Computation Time	Best case response time	Worst case response time	Response time jitter	Deadline
Buck Controller	20	20.0	73.0	53.0	250
Set Mains Level	4.65	4.65	73.0	68.35	250
Execute Mains	13	13.0	73.0	60.0	250
Store Sample PFC	6.23	6.23	216.49	210.26	1000
PFC Controller	28	28.0	73.0	45.0	250
Store Sample Measurement	8.26	8.26	216.49	208.23	1000
Measurement Task	95	95.0	216.49	121.49	1000
DALI Receiver	7.35	7.35	7.35	0.0	18.66
DALI Decoder	26.65	26.65	216.49	189.84	1000
DALI Interpreter	6.6	6.6	226.51	219.91	5000
Lightapp Task	3.42	3.42	226.51	223.09	5000

4.5 Server

A server is used to manage a number of tasks. It ensures their interference on the rest of the tasks is limited. A server has a computational budget for the managed tasks. Ideally, the budget would be as large as possible, to serve all requests as soon as possible. However, the maximum limit for the budget is the maximum interference control tasks may suffer, such that the quality

requirements are met. For the analysed use case, the maximum interference was determined to be 30 us in a 250 us interval. The second parameter of the server is its period. A shorter period distributes the load more evenly, while a longer period allows for fewer preemptions for the tasks executed in the server.

Applied to the use case, the server can be used to limit the interference of the DALI interrupt to the levels accepted by control tasks. However, there is a caveat to this situation. If DALI interrupt arrives when the budget of the server is depleted, the edges it is supposed to process may be missed. This is compensated by the fact that the worst-case activation rate of the DALI interrupt has a low chance of happening in practice.

4.6 Limited Preemption

The interference of the DALI interrupt on control tasks can be limited by deferring the preemption of the latter when the interrupt is triggered. This limited preemption is implemented through deferral. The interrupt is executed after a set period of time, or when the control task is not running, whichever happens first. This can be seen in figure Figure 4.3, which depicts both the scenario in which the communication interrupt is executed before the control task completes, as well as the one in which it is executed at the end. This allows the running task to complete its execution and cede the processor voluntarily. The duration of the deferral can be set to the largest delay the communication protocol can take. As this interval is may be significantly larger depending on the state of the communication, it may be advantageous to change it accordingly.

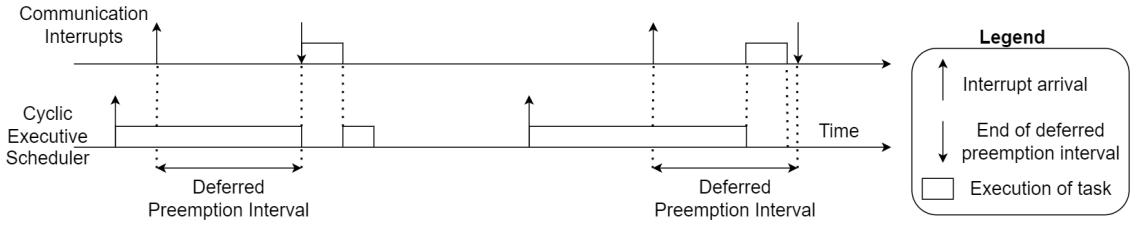


Figure 4.3: Limited Preeemption Design

4.6.1 Response time

The response time of the DALI interrupt and of the control tasks is influenced by the length of the deferrable interval. This interval is denoted as Def. There are three scenarios in which the DALI interrupt may trigger. If the control task is not executing, the DALI interrupt will be executed immediately. Its response time is its computation time C_D. The second scenario is the interrupt triggering when there is more than Def until the end of the control task. Then, it will be delayed by Def and executed. Its response time in this case is $C_D + Def$. The third scenario is when the control task finishes execution before the end of the Def interval. Then, the interrupt will be executed at the end of the control task. In this scenario, the response time of the DALI interrupt is between $[C_D, C_D + Def]$. Thus, it is guaranteed that the interrupt will start to execute at most Def after its release. This allows the communication protocol to meet its deadline, given that it may be delayed.

Using limited preemption, the time interval in which a control task may be interrupted is reduced from C_i to $C_i - Def$, where C_i is the computation time of the control task, and Def is the deferrable interval. Therefore, the maximum interference suffered by a control task is $\left\lceil \frac{C_i - Def}{T_D - C_D} \right\rceil * C_D$. This translates to a reduction in interference.

This solution can be easily integrated with the cyclic executive design patterns, as well as with a priority-based scheduling algorithm. Moreover, it is an improvement over the initial implementation.

4.7 Conclusions of theoretical analysis

After doing the theoretical analysis, it can be concluded that changing the underlying scheduling algorithm does not help in the analysed use case. While tasks meet their deadline, their worst-case response time is increased significantly. This can be seen in Figure 4.4. There are two reasons for this situation. The first is that the DALI interrupt is scheduled with a high priority with both priority-based algorithms. This is due to its short deadline. The second reason is that control tasks suffer interference from other tasks than the DALI interrupt, which is not the case for the initial system.

However, there are other advantages to using a priority-based scheduling algorithm. These are related to non-functional attributes of the system, such as maintainability and extendibility. These are important for the management of the system. Moreover, the algorithms are most often used with an operating system. The operating system provides additional benefits, such as protection for the memory stack. Therefore, we will further analyse the applicability, benefits and limitations of using an operating system through practical experiments in the next chapters.

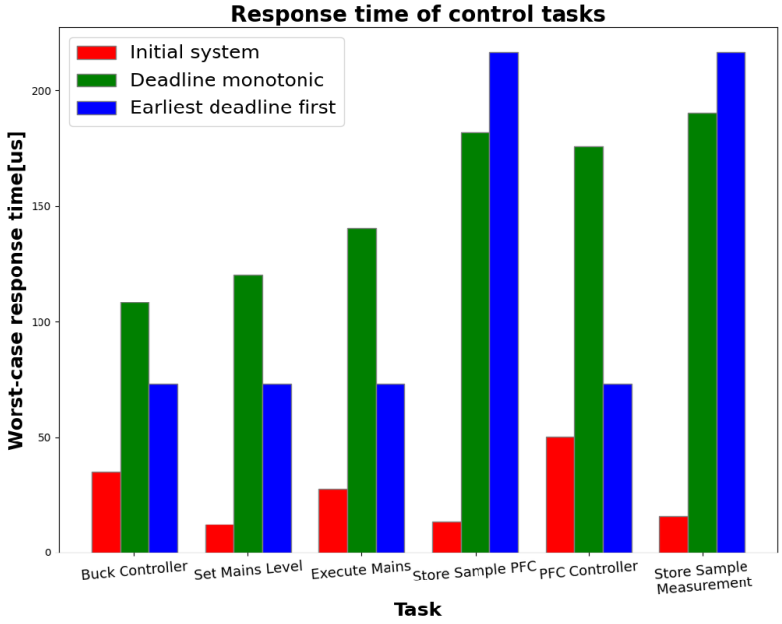


Figure 4.4: Worst-case response time for control tasks

Using a server to manage the DALI interrupt is not an ideal solution for the use case. This is due to the fact that, ideally, no DALI interrupts may be missed. However, the worst-case activation rate has a small probability of occurring in practice. Therefore, it makes sense to test a server implementation that protects the control tasks from interference.

Limiting the preemption of the control tasks is the most promising solution. It is able to reduce the interference caused by the DALI interrupt. Moreover, it ensures that the timing requirements of the DALI protocol are met. This can be done by setting a proper amount of time by which the preemption is deferred.

Chapter 5

Analysis of an RTOS Overhead

This chapter discusses experiments conducted to evaluate the overhead of FreeRTOS. The overhead is critical to determine how feasible it is to run an operating system on the driver. The results presented below were used to estimate the overhead used in subsection 4.3.2.

5.1 Experiments

The overhead of the operating system is the execution time spent on kernel operations. It falls under two categories, scheduling and message passing overhead. The former is analysed in subsection 5.1.1 and the later in subsection 5.1.2 and subsection 5.1.3. The tests are split in three different categories, focusing on the overhead of context switching, RTOS task notifications and semaphores.

The results of the evaluation of FreeRTOS are given below. The experiments were conducted on a microcontroller with a Cortex M0 processor and one with an M4. The differences are a result of the different processor speeds (48MHz vs 120 Mhz), as well as the existence of optimized kernel calls that use additional instructions only present on an M4 core.

5.1.1 Context switch with delayUntil

Most real-time systems need to implement tasks with periodic activation. An example is the case of the buck converter control task in our use case. There are two common ways of implementing periodic activation in FreeRTOS. The first one is to use the `delay()` function, which blocks the tasks until a certain number of ticks has passed since the function was called. This solution is not very accurate however because in the case the task is preempted by another task for a number of ticks before the function is called, that delay will not be considered for the next activation. The second and preferred solution is to use the `delayUntil()` function. It schedules the task to be activated at a number of ticks since the last activation, not since the function is called.

The experiments are related to the RTOS tick computation. The RTOS tick is called at a fixed frequency. When it is called, it updates the system time. It then checks whether there are any tasks that become active at the current tick. If that is the case, the RTOS tick switches in the task with the highest priority. The `delayUntil()` function is called at the end of the task and schedules the next activation of the task.

The experiments performed related to the context switch and the overhead of the `delayUntil()` function are presented in Table 5.1. The sequence diagram of the first experiment can be seen in Figure 5.1. It shows how the task interacts with the RTOS. Two intervals are measured through toggling pins. The first one is the interval between the RTOS tick until a task is resumed. The second interval is from the moment the task calls `delayUntil` until the idle task of the RTOS is resumed. The other experiments are variations on this experiment. They add background tasks

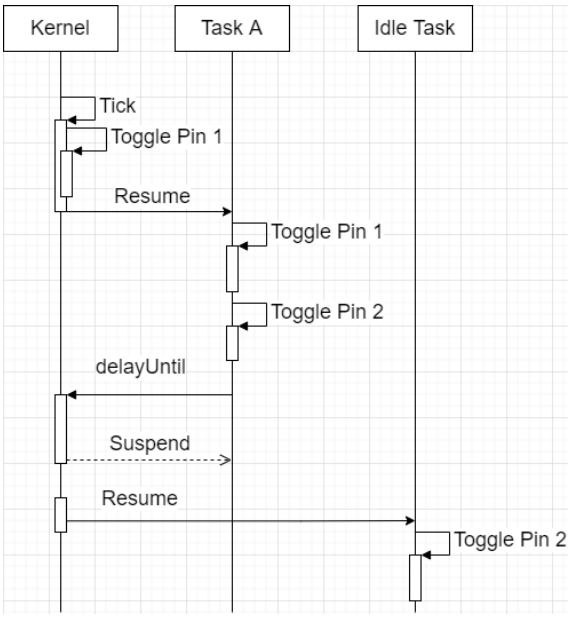


Figure 5.1: Experiment 1 with context switch

to evaluate the effects of loading the RTOS. The last experiment measures the RTOS tick without any running task.

The results of the experiments are shown in Table 5.2. They show that the operating system overhead is significant when compared to the frequency of the cyclic executive. That frequency generates slots of 83.33 us. The scheduling overhead would be, in this case, about a quarter of that slot.

ID	Short description	Motivation
1	task A periodic	Measure delay with one task
2	task A periodic; task B suspended	Measure influence of the suspended queue
3	task A periodic; task B continuous	Measure overhead with multiple tasks activated
4	task A periodic	Measure overhead when lower priority tasks are ready at the same RTOS tick
	a) 1 periodic background task	
	b) 2 periodic background tasks	
	c) 2 periodic background tasks each with different priorities, both lower than A	
5	task A periodic, 6 periodic background tasks	Measure overhead with multiple tasks
6	No tasks executing	Measure the RTOS tick with no tasks

Table 5.1: Experiments with delayUntil() API

5.1.2 Semaphores

Semaphores are a way to share variables and states provided by FreeRTOS. They are similar to a mutex in that regard. They are important in the context of the use case as they can be used to implement events. Tasks may be activated when certain events happen. A task may wait on a semaphore for a certain event to happen. As task dependencies can be represented through events, semaphores are a way to implement said dependencies.

ID	Measurement	M0 (us)				M4 (us)			
		avg	min	max	sdev	avg	min	max	sdev
1	Task resumption	12.87	12.86	12.89	0.004	3.57	3.57	3.57	0.000
	Task suspension	16.73	15.92	17.58	0.544	4.92	4.91	4.92	0.004
2	Task resumption	12.88	12.86	12.89	0.004	3.49	3.49	3.49	0.000
	Task suspension	17.09	16.44	17.74	0.513	4.72	4.49	4.94	0.148
3	Task resumption	12.88	12.86	12.89	0.004	3.50	3.49	3.50	0.000
	Task suspension	16.02	15.9	16.14	0.082	4.51	4.51	4.51	0.000
4 a	Task resumption	15.96	15.94	15.98	0.005	4.55	4.55	4.55	0.000
	Task suspension	16.66	16.64	16.68	0.006	4.57	4.57	4.57	0.000
4 b	Task resumption	19.04	19.01	19.06	0.006	5.52	5.52	5.52	0.000
	Task suspension	16.66	16.64	16.68	0.005	4.57	4.57	4.57	0.000
4 c	Task resumption	19.04	19.02	19.06	0.006	5.52	5.52	5.52	0.000
	Task suspension	16.66	16.64	16.68	0.006	4.57	4.57	4.57	0.000
5	Task resumption	31.36	31.32	31.40	0.011	9.45	9.45	9.45	0.000
	Task suspension	16.66	16.64	16.68	0.006	4.57	4.57	4.57	0.000
6	RTOS tick	3.95	3.94	3.95	0.001	0.866	0.866	0.866	0.000

Table 5.2: Overhead of delayUntil() API

Semaphores can be declared as either a binary or a counting semaphore. A binary semaphore can have two values, 0 and 1, and two operations: giving and taking. When a semaphore is given, its value is set to 1, and tasks waiting for it are activated. When it is taken, its value is set to 0. If its value was already 0, the task that tried to take the semaphore is suspended.

The overhead of a binary semaphore is composed of two components. The first one is the interval between the moment semaphore is given, until tasks waiting on it are activated. The second one is the interval between the moment a task signals that it waits on a semaphore, until the kernel suspends it and chooses another task.

Counting semaphores can have any non-negative value. They also feature the giving and taking operations. In addition to them, they can also return the value they hold without modifying it. This is the third component of their overhead, specific to them.

Two types of experiments were performed for semaphores. The first type aimed at determining the overhead of calling the API. The second type of experiment tested the responsiveness of the RTOS when semaphores are used to signal events. They measure the time between triggering the event and resuming the task associated with that event. A second measurement is done to determine the time between the finalization of the task and the resumption of other tasks. The task finalizes by trying to take the semaphore and waiting for the next event to trigger. The sequence diagram of the first experiment with a semaphore as an event can be seen in Figure 5.2. Here, an interrupt represents an event triggering. It gives the semaphore that task A is waiting on, and the kernel resumes the task as a consequence. Then, task A finishes its execution and takes the semaphore. This operation is not successful, as the semaphore cannot the taken at the moment, and the kernel suspends task A. Then, the idle task of the operating system is resumed. The two intervals mentioned are measured by means of toggling two pins.

ID	Short description
1	Counting semaphore give
	Counting semaphore take
2	Binary semaphore give
	Binary semaphore take
3	Get current value of counting semaphore

Table 5.3: Experiments with semaphore API

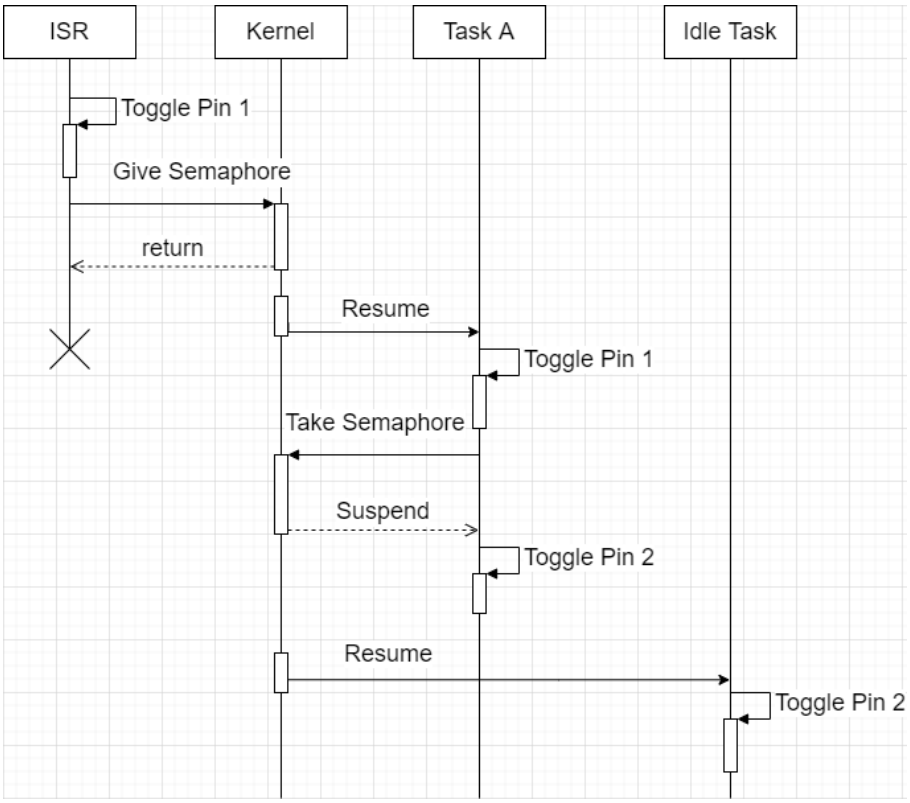


Figure 5.2: Experiment 1 with semaphore as event

The next experiments are variations of the first one. In experiments 2-4, the environment of the RTOS is changed by adding different background tasks. In experiment 5, task A is not triggered by an ISR, but by another task B. Moreover, as there are two versions of semaphores in FreeRTOS, binary and counting, the experiments are redone for both of them.

The results with the overhead of the API are given in Table 5.5. The results of using semaphores as events are given in Table 5.4. It can be seen that changing the environment by adding multiple tasks does not significantly influence the measured times. However, the resumption of task A takes more time. The results show that semaphores are too slow to use for control tasks, which have strict timing requirements. However, they can be used to implement events for the rest of the tasks, as they do not have the same strict requirements.

1	Task A woken by ISR	Response time when there is nothing else running
2	Task A woken by ISR; Task B suspended	Response time with a suspended task
3	Task A woken by ISR; task B continuous, lower priority	Response time with a running task
4	Task A woken by ISR; 6 continuous tasks	Response time with multiple running tasks
5	Task A woken by periodic task B	Response time when triggered from another task

Table 5.4: Experiments with semaphores as events

ID	Measurement	M0 (us)				M4 (us)			
		avg	min	max	sdev	avg	min	max	sdev
1	Counting Semaphore Given	6.37	6.36	6.37	0.002	1.75	1.75	2.27	0.024
	Counting Semaphore Takes	3.91	3.90	3.91	0.001	1.10	1.10	1.34	0.013
2	Binary Semaphore Given	6.36	6.36	6.37	0.002	1.84	1.84	1.84	0.000
	Binary Semaphore Takes	3.90	3.90	3.91	0.001	1.10	1.10	1.10	0.000
3	Get count of semaphore	2.66	2.66	2.67	0.001	1.10	1.10	1.10	0.000

Table 5.5: Overhead of semaphore API

	ID	Measurement	M0 (us)				M4 (us)			
			avg	min	max	sdev	avg	min	max	sdev
Binary	1	Task woken	16.90	16.57	21.37	1.145	4.92	4.90	5.98	0.134
		Task suspension	35.12	33.46	41.25	1.980	9.58	9.24	11.17	0.289
	2	Task woken	17.18	16.57	31.52	2.031	4.95	4.89	8.22	0.240
		Task suspension	35.22	33.44	50.44	2.901	9.36	9.06	11.85	0.368
	3	Task woken	16.93	16.56	21.37	1.172	4.89	4.87	5.95	0.147
		Task suspension	33.91	32.96	39.37	1.926	9.47	9.35	10.95	0.276
	4	Task woken	16.95	16.56	21.37	1.205	4.98	4.94	6.08	0.186
		Task suspension	33.96	32.93	42.21	2.180	9.63	9.52	11.10	0.271
	5	Task woken	18.79	18.77	18.81	0.006	4.88	4.88	4.88	0.000
		Task suspension	33.68	33.66	33.73	0.011	9.22	9.22	9.22	0.000
Counting	1	Task A resumption	16.91	16.55	21.37	1.161	4.93	4.91	5.99	0.116
		Task suspension	35.104	33.43	41.20	1.161	9.65	9.39	11.31	0.318
	2	Task woken	17.04	16.55	31.48	1.747	4.93	4.87	9.32	0.296
		Task suspension	35.271	32.97	49.85	2.817	9.63	9.31	13.74	0.435
	3	Task woken	16.95	16.56	21.39	1.216	4.85	4.82	5.90	0.180
		Task suspension	33.75	32.96	39.34	1.768	9.72	9.60	11.17	0.280
	4	Task woken	16.94	16.56	21.36	2.256	5.05	5.02	6.10	0.156
		Task suspension	34.03	32.94	42.15	2.256	9.70	9.60	11.90	0.293
	5	Task woken	18.79	18.77	18.82	0.006	5.09	5.09	5.09	0.000
		Task suspension	33.70	33.67	33.74	0.011	9.12	9.12	9.12	0.000

Table 5.6: Responsiveness of semaphores as events

5.1.3 Task Notifications

FreeRTOS provides multiple means of waking blocked tasks on the basis of an event. If the task to be resumed is known beforehand, the fastest means is to notify the task directly using task notifications. They are an alternative to semaphores. Compared to them, the woken task is coupled with the triggering function. This represents a trade-off between speed of execution and abstraction. In the analysed use case, the relations between tasks are known before runtime. This enables the use of task notifications.

The experiments involving task notifications are the same ones as the ones used to evaluate semaphores as events. They are described in Table 5.4. The difference is that task notifications are used instead of semaphores. The results of the experiments are shown in Table 5.7. They show a clear reduction in execution time compared to semaphores.

ID	Measurement	M0 (us)				M4 (us)			
		avg	min	max	sdev	avg	min	max	sdev
1	Task woken	13.90	13.58	18.39	1.082	4.37	4.31	5.40	0.230
	Task suspension	13.06	12.04	18.47	1.114	3.55	3.28	4.87	0.247
2	Task woken	13.92	13.59	18.39	1.131	4.28	4.22	5.30	0.227
	Task suspension	13.04	12.04	18.53	1.055	3.65	3.24	4.66	0.210
3	Task woken	13.88	13.59	18.39	1.052	4.29	4.24	5.31	0.223
	Task suspension	11.91	11.56	16.51	0.952	3.35	3.28	4.35	0.183
4	Task woken	13.92	13.6	18.38	1.093	4.32	4.24	5.40	0.244
	Task suspension	12.05	11.55	20.41	1.744	3.40	3.25	5.59	0.458
5	Task woken	15.06	14.79	19.60	1.040	4.04	4.04	4.04	0.000
	Task suspension	12.42	12.16	17.15	1.006	3.50	3.50	3.50	0.000

Table 5.7: Overhead of task notifications

Chapter 6

Porting the Use Case to FreeRTOS

This chapter describes how the system may be ported to FreeRTOS and how a static priority algorithm can be used to schedule a selection of tasks from the system. The aim is to determine to what extent all the functionalities could be implemented with the intended behavior on an out-of-the-shelf real-time operating system. Having both a bare metal and OS implementation of the analyzed solutions allowed us to measure the overhead of using an operating system.

6.1 Functionality selection

The port implements a subset of the functionality available in the driver. The first reason these functions were selected is that they represent the core functionality of the driver. This subset consists of the buck controller, the tasks of the measurement module, and the DALI communication protocol. The buck controller maintains the output current at a stable level, enabling the use of the LED load. The measurement module is used by the controller to store, filter and retrieve measurements. DALI communication enables remote control and data access.

The second reason for the selection of these features is that they constitute different task types. They show how the operating system can handle different use cases. The buck controller is a periodic task that has to be executed as fast as possible with minimal jitter. It was originally executed in the cyclic executive scheduler. The measurement module contains periodic tasks with deadlines equal to their period. The DALI communication protocol is implemented through a sporadic interrupt that either collects incoming edges. The DALI decoder is called periodically. The DALI interpreter is called when a full message is received.

The focus of the implementation is the integration of the DALI interrupt and decoder with a control task. Therefore the application layer of the DALI protocol is simplified. The port does not use all types of messages that can be interpreted by the application layer. Only broadcast messages are accepted and processed. They are used to set the value of the light level.

6.2 RTOS selection

The operating system used in the experiment is FreeRTOS. There are three reasons for this selection. The first one is that it is free and open-source. This makes it faster and cheaper to test it on the driver, because it does not require a licence. The second reason is its large support and documentation. This makes debugging and understanding its implementation easier. The third reason is that it supports priority-based scheduling, and offers task synchronization primitives. These features enable porting the use case.

6.2.1 FreeRTOS Setup

Task scheduling in FreeRTOS is based on the execution of a periodic tick. Its role is to measure time for the real-time kernel, track when tasks need to be activated and select the highest priority task to be run. The frequency of the operating system tick is set to 12kHz. This frequency matches the granularity of the cyclic executive scheduler. The default total heap size for the whole system was reduced from 10000 words to 5000. The reduction was needed in order to fit in the RAM of the XMC1400.

6.3 Task priority assignment

The implementation explores how static priority scheduling can be used, since it is the base for multiple scheduling algorithms. Tasks are assigned a fixed priority, and higher priority tasks may preempt lower priority tasks. This method is supported by multiple real-time operating systems.

The priority assignment does not follow any particular algorithm, such as deadline monotonic, but rather mimics the tasks' assignment to the cyclic executive and cooperative schedulers in the original implementation of our use case. This was done in order to introduce as few modifications as possible when testing whether the microcontroller is able to run an operating system. Thus, tasks originating from the cooperative scheduler are scheduled in the background(i.e., are assigned low priorities), while tasks coming from the cyclic executive scheduler are given the highest priority.

6.4 Task design

Periodic FreeRTOS tasks are implemented as seen in Listing 6.1. They use the `vTaskDelayUntil` function to create periodicity. This function was chosen over `vTaskDelay` because the latter delays a task from the moment it is called, and the former delays the task from the last wake time.

```
void vTask(void *pvParameters) {
  TickType_t xLastWakeTime;
  // set the period of the task
  const TickType_t xFrequency = task_tickinterval;
  //get the initial wake time
  xLastWakeTime = xTaskGetTickCount();

  while(1) {
    //execute the task
    Task_vExecute();
    //delay the task until its next activation
    vTaskDelayUntil(&xLastWakeTime, xFrequency);
  }
}
```

Listing 6.1: FreeRTOS Generic Task Implementation

The features selected for implementation are mapped to FreeRTOS tasks. The first feature is the buck controller. The actions needed for the controller correspond to tasks T1 and T2 from Table 3.2. As T2 is dependent on T1, and they have the same frequency, they are mapped to the same task and executed one after the other. To achieve the desired frequency of 4 kHz, its period is set to 3 operating system ticks.

The functions of the measurement and of the DALI modules are called with a frequency of 1 kHz. They are implemented as separate tasks in FreeRTOS. When a DALI message is decoded, a third task is called to interpret it. The DALI decoder task calls the interpreter task through task notification. The architecture of the task design can be seen in Table 6.1.

Task	Type	Period(Rtos Ticks)	Priority(1 = lowest)
Buck Controller	Periodic	3	2
Measurement	Periodic	12	1
DALI Decoder	Periodic	12	1
DALI Interpreter	Notified task	-	1

Table 6.1: Architecture of RTOS port of the use case

6.5 Initialization sequence

After designing the architecture of the tasks, the next step is writing the initialization sequence. In the original system, the initialization was performed by higher level tasks. These monitor the state of the driver from a higher level. They are used to separate the management of different software modules from their implementation. Because they involved tasks from software modules not included in this port, such as the PFC controller, they were also not included. Thus, in the new implementation, the initialization had to be done manually.

The initialization sequence was checked by inspecting state variables with a debugger. Another way to validate the sequence is to check that the hardware cycle-by-cycle buck controller behaves as expected. It controls the output current of the driver. It is crucial that it works correctly. Otherwise the driver may suffer damage in tests with high voltage. To verify the hardware controller, the initialization code was run on a bootkit board with an XMC1402 microcontroller connected to a buck converter made of discrete electronic components.

6.6 Alternative implemenation

Porting each task from the original system to its individual task has some disadvantages. The periodicity of the tasks is realized manually through kernel calls. This adds boiler plate code. As this is repeated for all of them, it increases the risk of error which is not a good practice.

Periodic and sporadic tasks are implemented differently using kernel callback functions. While this is not a disadvantage in itself, in the cooperative scheduler, there is a common mechanism to define both types of tasks. This is achieved by having all tasks triggered by events. Events can be triggered by either the expiration of timers or through a direct call. Thus, the porting of tasks to FreeRTOS is not straightforward, because a different paradigm is used.

Another disadvantage is that each task do not use a common stack. Each task is allocated a RAM section to be used as stack. Even though tasks are not executed at the same time, they occupy RAM simultaneously, increasing the memory footprint of the system.

FreeRTOS features a timer task. The timer task keeps track of several software timers and callback functions. Tasks can be defined as callback functions and attached to a timer. The timer task also enables pending a call to a callback function. This provides a way to represent events. The generic structure of a callback function can be seen in Listing 6.2.

```
void vTimer_callback_function(TimerHandle_t xTimer) {
  Task_vExecute();
}
```

Listing 6.2: FreeRTOS Generic Timer callback function implementation

The advantage of this solution is that it defines tasks similarly to the cooperative scheduler, easing their porting. The second advantage is a reduced RAM footprint, as there is only one task control block created to manage multiple functionalities. However, the callback functions of the timer task are executed in a cooperative way. Therefore, only tasks at a single priority level may be places under its management.

In the implementation of the use case, the buck controller is still defined as a separate task, because it has a higher priority compared to the other tasks. The rest of the tasks can be defined as callback functions for the timer task.

6.7 Results

The aim of the experiments conducted with the FreeRTOS-ported system is to evaluate its overhead on the microcontroller. A secondary goal was to examine how tasks behave in the environment of an RTOS. The execution of the buck controller, measurement and DALI decoder task can be seen in Figure 6.1. The figure shows a critical instance when all tasks are released at the same time. The delay between the RTOS tick and the execution of the buck controller task is constant, being around $6\mu s$. After the execution of the buck controller task, it suspends itself and calls the operating system scheduler. The scheduler then sets its next execution, and then chooses the next task to execute from the ready queue. This operation takes around $15\mu s$. The version using the timer task offered by FreeRTOS has a similar overhead, as it internally uses the same underlying primitives.

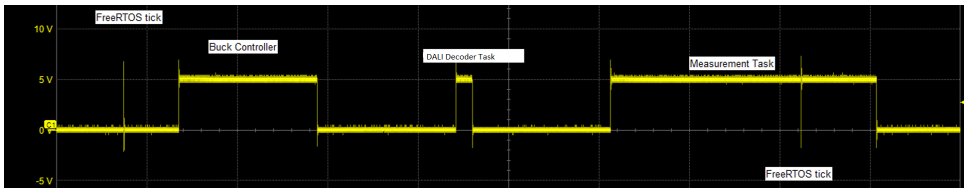


Figure 6.1: Execution of the tasks compared to the RTOS tick

The DALI interpreter can receive broadcast messages and change the reference value for the buck software controller. The reception of a DALI message can be seen in Figure 6.2. The DALI communication line is shown in blue in the figure. Its idle state is up, at 15 volts. When bits are transmitted, it goes from the idle state, to using Manchester encoding. The pink line shows the activations of the DALI decoder task. When the DALI line is set to 0, it shows spikes due to electrical interferences. After the DALI message is transmitted, it can be seen that the DALI decoder task is executed, calling the DALI interpreter task. Its execution can be seen on the yellow line.

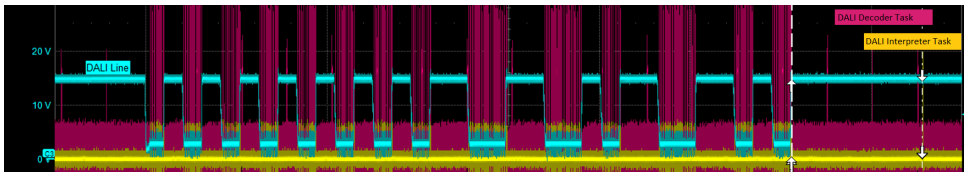


Figure 6.2: Reception of a DALI message

The experiments show that using FreeRTOS to implement a functionality subset is feasible. However, its overhead is too high to be used for scheduling multiple control tasks. Its use would only be feasible in scheduling soft real-time tasks that were scheduled in the cooperative scheduler. Because they have the same priority and do not have strict timing requirements, it is not advantageous to use an operating system instead of the original implementation.

Chapter 7

Improving QoS by Controlling Interference and Response Jitter

Using a priority-based scheduling algorithm to schedule the DALI interrupt as a regular task does not improve its interference on the control tasks. Therefore, another way to control that interference is needed. In this chapter, different server implementations and one solution for limiting preemption were developed around an interrupt. These were designed to be independent of an operating system and of an underlying scheduling algorithm. These implementations are presented and the impact they have when used on the original system is shown.

7.1 Jitter interference on controllers

One important aspect for a potential solution to the interference suffered by control tasks is the effect of jitter on them. More precisely, this section aims at determining the amount of jitter under which control tasks still deliver a high quality of service(QoS). The relation between jitter and the QoS delivered by controllers is determined through practical experimentation.

The experiments concerning jitter were aimed at measuring the amount of flicker from the LED load. This is an indirect measure of the QoS offered by the driver. The jitter was introduced in the buck controller, as that is the last stage in the electrical pipeline, making the effect of jitter more pronounced. If jitter were introduced in the PFC controller, the buck controller would be able to ameliorate the jitter's effect.

7.1.1 Delay types

There are two types of delay that can be introduced and that affect the quality of the controller. The first one is the delay between the release of the task and the reception of the input. The second one is the delay between the input and output of the controller. These delays occur when other tasks, most importantly the DALI interrupt, preempt a control task. The delays can be visualized in Figure 7.1.

Static delays To examine the effect of the delays, they are artificially introduced through making the task busy wait at the 2 points in time. The busy wait is introduced through a while loop. The interference is determined by measuring the flicker of the light emitted by the LEDs. The flicker is measured by using a Klein K-1 colorimeter with software provided by the manufactured, KLEIN K Colorimeter. The set-up used for the flicker measurement can be seen in Figure 7.2.

Variable Delays To further explore the effect of jitter, a variable jitter was introduced in the buck controller. The jitter was between 2us, which was the overhead of generating the random number, and a maximum delay set for that experiment. Jitter was introduced in both the release

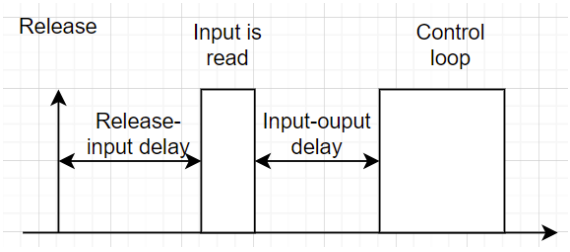


Figure 7.1: Potential delays in the control task

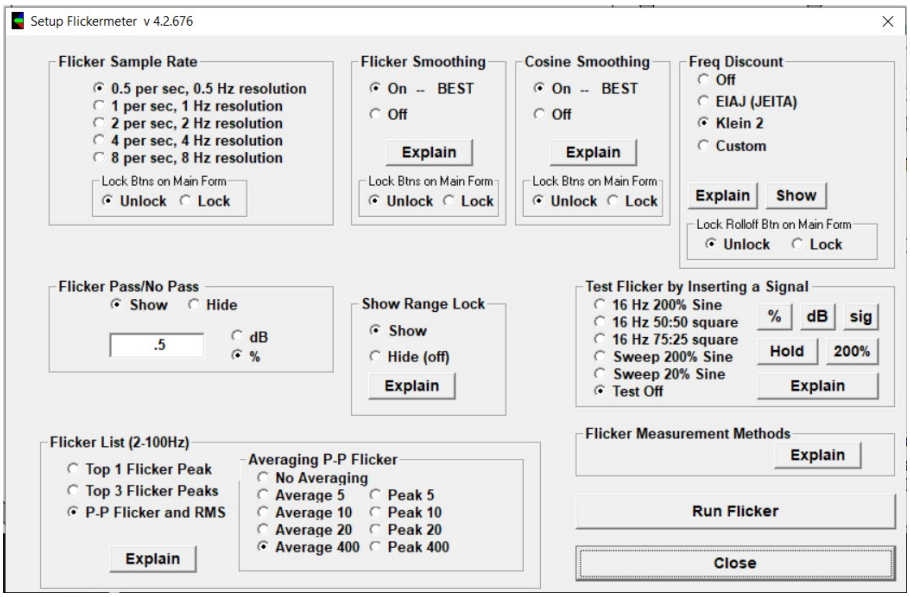


Figure 7.2: Flicker measurement settings

of the task, as well as between the input and output of the controller, and the driver was running the whole system, connected to AC.

7.1.2 Static Jitter

The first set of experiments was conducted by introducing a fixed delay in the buck controller running as the sole task in the system. The results of the flicker measurements can be seen in Figure 7.3. For each combination, the average flicker value measured in percentage is given, together with the peak value. The goal is to have the flicker never pass 0.05. It can be seen that the delay between the input and the output of the control loop has the biggest influence on the light quality.

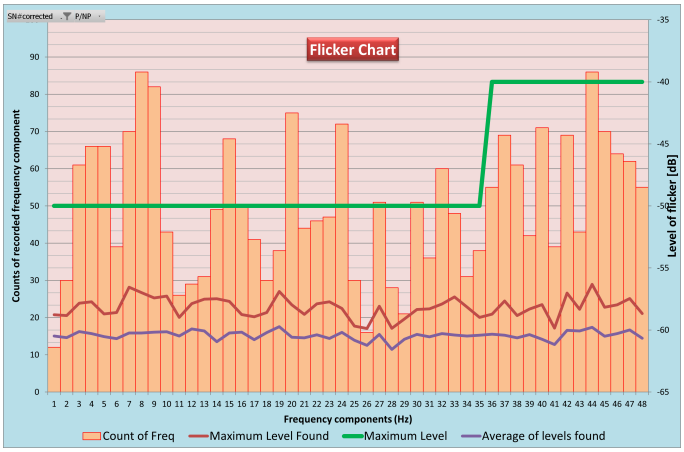
7.1.3 Variable Jitter

To further explore the effect of jitter, a variable jitter was introduced in the buck controller. The jitter was between 2us, which was the overhead of generating the random number, and a maximum delay set for that experiment.

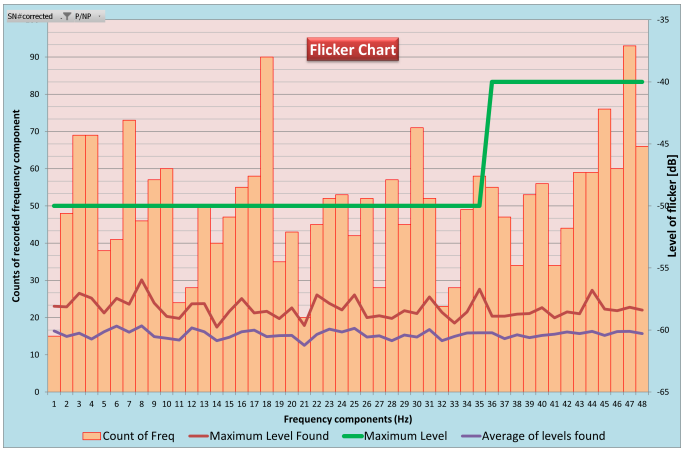
Some results are plotted in Figure 7.4. The plot can be interpreted in the following way. The colorimeter records flicker by taking measurements as fast as possible in a 5-minute period. Each measurement represents flicker at a given moment. The flicker is a wave that can be decomposed into frequency components. Each component has a certain amplitude, expressed in dB. For each measurement, the frequency with the highest amplitude is taken as representative. The plot shows

Release_input_delay/input_output_delay	0	5	10	15	20	25	30	35	40	45	50	55	60
0	0,048	0,063	0,046	0,056	0,058	0,048	0,061		0,059		0,058		0,063
5	0,024	0,024	0,022	0,041	0,037	0,038	0,041		0,36		0,045		0,045
10													
15	0,057		0,063		0,057		0,053						
20	0,041		0,044		0,039		0,03						
25	0,056												
30	0,031												
35			0,05										
40			0,036										
45													
50							0,053						
55							0,029						
60													
	0,058												
	0,038												

Figure 7.3: Flicker measurements results



(a) Flicker with no jitter

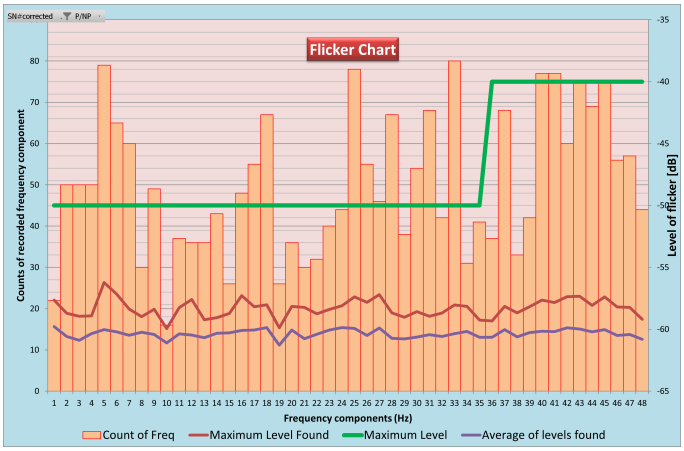


(b) Flicker with variable input-output jitter of 10 us

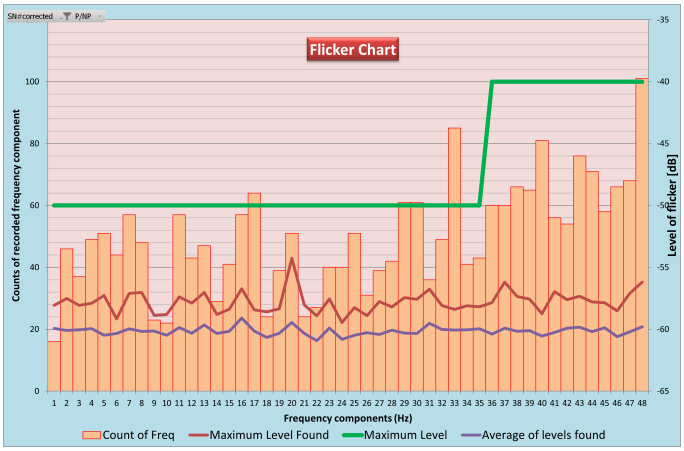
the set of highest frequencies, one for each measurement. The bars represent the histogram of frequencies. It shows the distribution of the highest frequencies. The count for each frequency can be seen on the left vertical axis. Overlaying over the histogram are 3 line plots. The purple line represents the amplitude average for each frequency. The red line represents the maximum amplitude found for each frequency. The green line represents the maximum acceptable level of flicker for each frequency. The right vertical axis represents the amplitude values for each line plot.

Variable jitter was introduced in both the release of the task, as well as between the input and output of the controller, and the driver was running the whole system, connected to AC.

From the results, there is enough room in the system for jitter, without a significant impact on the flicker. However, the system was only tested with jitter on the buck controller, and with a maximum delay that would not cause an overflow of the task in another slot of the cyclic executive scheduler. Therefore, a smaller maximum value of jitter needs to be considered when looking for a solution. Combined with the value obtained from analysing static jitter, the maximum amount of jitter acceptable in a control task is estimated to be 30us for every period of a controller, which is equal to 250 us.



(c) Flicker with variable input-output jitter of 80 us



(d) Flicker with both release and input-output jitter of 120 us

Figure 7.4: Flicker with variable controller jitter

7.2 Server

One method of controlling the interference of the communication protocol on the controllers is through a server. A server is a dedicated task that limits the interference of sporadic jobs. It can be used to manage the execution of the DALI interrupt. Servers accept or deny sporadic jobs based on a predefined budget. The implementation assumes that the sporadic jobs are non-preemptive, and they can only be managed by disabling them. This mirrors the fact that interrupts of the communication protocol cannot be preempted once they start to execute. Another assumption that was made is that the jobs managed by the server cannot interrupt each other. This assumption eliminates the case when the server manages interrupts coming from communication protocols with different priorities.

Fixed-priority servers are usually executed with the highest priority. Thus, they do not need information about periodic tasks. They have a simple implementation, which consists of 3 callback functions. The first one is the server period callback, whose purpose is to replenish the server's budget. The second and third ones are job arrival and finalization callbacks, which manage the budget of the server, and disable new job arrivals if the budget is exhausted.

The overhead of the operating system exceeds the computation time of the interrupt. Thus, managing the interrupt with a server implemented as an RTOS task would only increase the interference on control tasks. The server is implemented independently of the operating system. It uses a separate hardware timer to keep track of time. Moreover, the server can be integrated with the complete use case, enabling the comparison of results with the original implementation. To reduce hardware requirements, only one hardware timer is used.

The server's budget is the computation time allowed for sporadic tasks in a server period. If the worst-case computation time of the managed tasks is known, the budget can be defined as the number of jobs to be admitted in that period. Either way, the budget is stored in an integer. The capacity of a server is the remaining budget at any given time. When a sporadic job arrives, the server's capacity needs to be reduced. If the budget is a number of jobs, the reduction can be done at the job's arrival. If the budget is defined as computation time, the reduction needs to be done at the end of the job, as its computation time is assumed to not be known beforehand. When the budget of the server is exhausted, the source of the sporadic jobs is disabled. In the analysed use case, that source is represented by the DALI interrupt. The server has custom implementations depending on the definition of the budget, to minimize overhead. For instance, if the budget is only one job, upon a job arrival, the interrupt is disabled immediately. This is done without keeping track of the server's capacity, because the budget is known to have been exhausted upon the job arrival.

It is assumed that jobs may not interrupt each other. The assumption is not relevant for the implementation if the budget is a number of jobs. This is the case because the server only has to keep track of the number of admitted jobs, which is the same. However, if the budget is defined as computation time, the reduction of capacity happens at the end of the job. If the possibility of jobs interrupting each other is not taken into account, the same computation time may be deducted multiple times from the server's capacity. To counteract this, the server keeps track of multiple arrival times, and upon the finalization of a job, it takes into account the computation time of jobs that interrupted it. The computation time is equal to the finalization time minus the arrival time, and minus the sum of the difference between finalization and arrival times of the interrupting jobs. For the example, in Figure 7.5, the computation time of $T1$ is $f1 - a1 - [(f2 - a2) + (f4 - a4)]$. To hold the computation time of the interrupting jobs, as well as multiple arrival times at once, a stack is used.

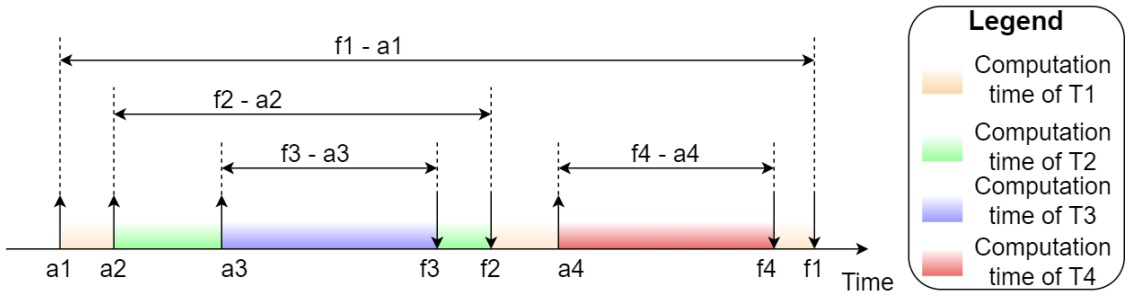


Figure 7.5: Computation times of interrupting jobs

The nested vectored interrupt controller(NVIC) is an important aspect for the server's implementation. It handles interrupts on the ARM Cortex-M processors. If an interrupt is disabled by software and cannot be executed immediately, the NVIC sets a pending flag for that interrupt. A server can take advantage of this feature to know if sporadic jobs were activated while its budget is exhausted.

7.2.1 Server Variants

There are several fixed-priority server algorithms available, suited for different applications. However, their implementations are quite similar. They can all be implemented using the three callback functions described in the previous section. Therefore, it is interesting to evaluate their behaviour on the XMC1400 microcontroller. Thus, they were implemented and tested. In this subsection, the implementation aspects specific to each algorithm are discussed.

Polling and Deferrable Servers

The polling and deferrable servers are both periodic servers. Their budget is replenished to its maximum value every period. In the implementation, this is done with the server interrupt callback function. The difference between the two is the way the budget is consumed. The budget of the polling server is intended for immediate consumption. Thus, if there are no pending jobs at the beginning of a period, the budget is lost for that period. The budget of the deferrable server is left available for the duration of the period. Thus, it can serve jobs immediately upon arrival, in the limit of its budget.

The polling server replenishes its budget only if there are pending jobs when it is activated. This can be checked using the pending interrupt flag set by the NVIC. If there are no pending jobs upon finalization, interrupts are disabled, which corresponds to making the budget 0.

Sporadic Server

The sporadic server is not a periodic server, in contrast to the polling and deferrable servers. It replenishes its budget only if a job has consumed it beforehand. The replenishment happens one server period after the consumption began. This can be seen in Figure 7.6.

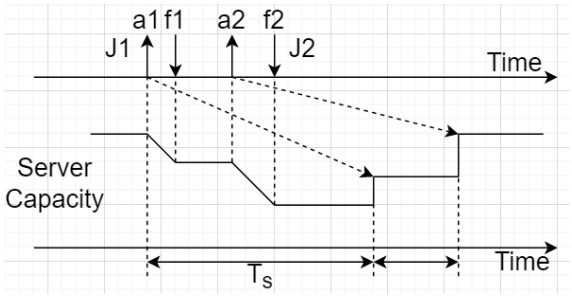


Figure 7.6: Operation of the sporadic server

There is only one hardware timer that is used to time the next replenishment. After the arrival of the first job, the timer is loaded with the server period. However, the time between further replenishments is stored as deltas from the last replenishment in a queue. If the server timer is running upon arrival, the replenishment time is computed by subtracting the sum of further replenishments, the remaining time until the timer expires, and the computation time of the server period callback function. This calculation can be seen in Figure 7.7. If the server uses a time budget, the amount to be replenished is put together with the replenishment time in the queue. If the budget is a number of jobs, the amount to be replenished is one, because each replenishment corresponds to one job.

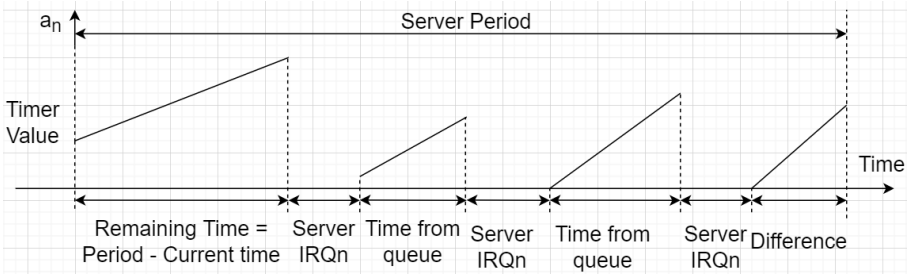


Figure 7.7: Server Replenishment Time

7.2.2 Overhead

The server is used to manage the DALI interrupt. Because of this, the overhead of the server is critical. The overhead of the different server implementations is measured by simulating a burst of jobs. The results are given in Table 7.1. It can be seen that the overhead of the server increases as the assumptions concerning jobs are lifted. The smaller overhead is obtained in the case where the server admits only one job. For a budget of multiple jobs, the server also needs to keep track of its budget, which increases its computation time. If the budget is defined as computation time, the server also needs to compute the time spent by each job. Furthermore, if jobs can interrupt each other, the server needs to keep track of the interrupts in a stack, which adds to its overhead.

Table 7.1: Server implementation overhead

Algorithm	Budget	Arrival Function(ns)	Finalization Function(ns)	Server Function(ns)
Periodic	1 job	496	-	433 - 951ns
	n jobs	455 - 1380	372 - 1300	413 - 1070
	time	414	992 - 2810	454 - 1170
	time nested	518	1490 - 3350	455 - 1170
Deferrable	1 job	579	-	723
	n jobs	496 - 847	-	869
	time	352	744 - 1170	993
	time nested	580	1210 - 1570	993
Sporadic	1 job	725	-	725
	n jobs	869 - 2960	-	1220 - 3480
	time	620 - 3430	910 - 2310	1880 - 3840
	time nested	972- 4690	1790 - 2620	2540 - 6040

The server variant that was chosen to be integrated with the original system is the sporadic server. This is due to the fact that it does not produce any overhead when the interrupt is not active. The budget of the server was set to 1 job. This can be done as the computation time of the interrupt is bounded. Moreover, setting the budget to 1 job has the least overhead. The activation of a DALI interrupt under the server can be seen in Figure 7.8. The activation is followed by the replenishment of the server budget.

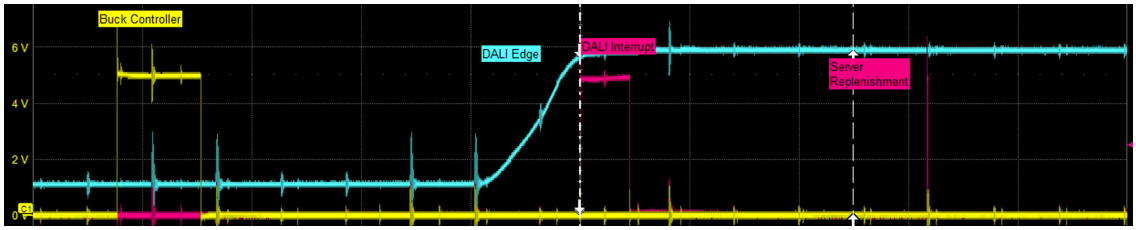


Figure 7.8: Activation of DALI interrupt under a sporadic server

The maximum interference allowed by the server was estimated to be 30 us in any 250 us. As the maximum computation time of the interrupt is about 7.5 us, the server can allow at most 4 interrupts. Thus, the period of the server was set to 250/4=62,5us.

Two measurements were taken to evaluate the effects of using a sporadic server. The first one is the response time of the DALI interrupt and of the buck controller. The second measurement is the light flicker of the LED load. For these measurements, 4 different situations were taken into account. Firstly, the server was used, but there were no messages sent. In the second experiment, stress messages were sent as fast as possible. The messages altered between one setting a register, requiring no response, and one that requests the value of the set register, to trigger a response from the driver. The same set-up was used for the system with no server, to compare the results.

7.2.3 Response time

Response time is crucial for the buck controller, as it is a measure of the interference is suffers from the DALI interrupt. Response time measurements are given in Table 7.2. They were taken when the system was under stress of DALI messages, as described in the previous section. It can be seen that the server does not increase the response time of the buck controller. At the same time, the response time of the DALI interrupt is increased by the overhead of the server, as expected.

Function	Min	Max	Mean	Sdev
Buck Controller without server	21.28	33.91	21.79	0.59
Buck Controller with server	21.26	33.82	21.79	0.66
DALI Interrupt without server	3.96	9.1	7.21	0.85
DALI Interrupt with server	4.52	11.08	8.26	0.93

Table 7.2: Response times with server (us)

7.2.4 Light Flicker

Another important measurement is the light flicker of the LED load. It is an indirect measure of the impact of servers on the quality of service provided by the buck controller. Light flicker was analysed by a colorimeter, taking multiple measurement in a span of 5 minutes. Each flicker measurement was decomposed, showing the amplitude of its composing frequencies. Then frequencies with the largest amplitude value measured through the complete experiment were recorded, and used as an estimate for the buck controller performance.

Frequency (Hz)	Maximum peak(dB)
48	-66,3
47	-66,9
43	-67,49
45	-67,85
46	-68,38
44	-68,6
42	-68,65
33	-68,67
41	-68,71
30	-69,09
Avg	-68,064

(a) Server without DALI messages

Frequency (Hz)	Maximum peak(dB)
48	-66,13
44	-66,34
45	-66,4
47	-67,59
46	-67,79
37	-68,06
43	-68,14
42	-68,45
41	-68,79
32	-68,96
Avg	-67,665

(b) Server with DALI messages

Frequency (Hz)	Maximum peak(dB)
47	-64,1
45	-66,51
39	-66,82
46	-67,11
48	-67,57
44	-68,1
38	-68,37
42	-68,39
25	-68,94
43	-68,95
Avg	-67,486

(c) No server without DALI messages

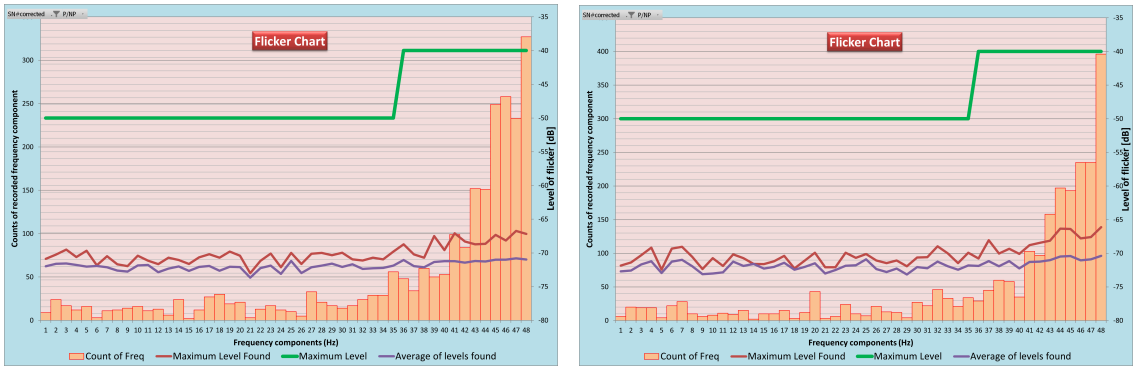
Frequency (Hz)	Maximum peak(dB)
47	-66,74
41	-67,08
48	-67,18
45	-67,32
39	-67,51
46	-68,16
42	-68,32
44	-68,64
36	-68,72
43	-68,73
Avg	-67,84

(d) No server with DALI messages

Table 7.3: Light Flicker Tests of Server

The top 10 frequencies with the highest amplitude registered for each experiment can be seen in Table 7.3. Moreover, Figure 7.9 shows a plot of the biggest amplitude frequencies when the system is under stress.

Despite the fact that DALI messages were sent at maximum speed, there were no spikes on the line, so interrupts were spaced by $Te = 416us$. The result is that the server does not slow down the interrupts in this setup, but rather only adds overhead, which can be seen in the flicker. On the other hand, this also shows that the overhead does not increase the flicker over the acceptance threshold.



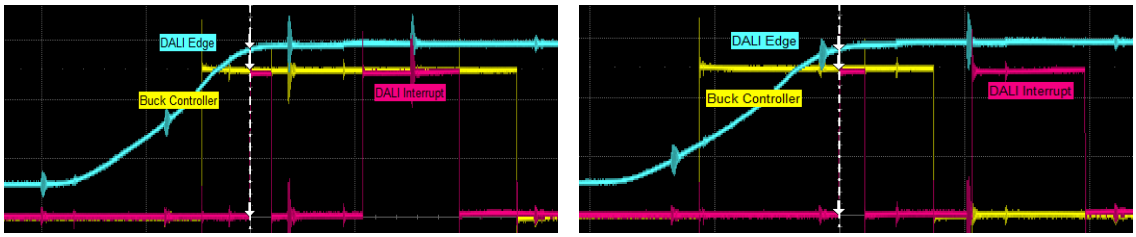
(a) No server with DALI messages

(b) Server with DALI messages

Figure 7.9: Influence of server on the system

7.3 Limited Preemption

Limiting the preemption of the control tasks can be achieved by delaying the execution of the DALI interrupt in certain conditions. If a control task is running when the interrupt is triggered, the interrupt is delayed by a set amount of time. There are 2 cases in this scenario. The first case is when the amount of time passes before the control task finishes its execution. Then the DALI interrupt will preempt the control task. The second case is when the control task completes its execution before the amount of time passes. The DALI interrupt will be executed afterwards, and the control task will not be preempted. These two scenarios can be seen in Figure 7.10.



(a) The buck controller does not finish its execution before the delayed interval passed and is preempted

(b) The buck controller finishes its execution, and the DALI interrupt executes afterwards

Figure 7.10: Activation of the DALI interrupt under limited preemption

The implementation of limited preemption requires a way to measure time. To that end, a hardware timer is used. This makes the implementation independent of an operating system. There are three callback functions necessary. They are grouped inside a C module. The first one is called from the communication interrupt when it starts to execute. It signals that it was released. This function determines if the interrupt is to be delayed or allowed to execute. If it is delayed, it will start the hardware timer. The timer triggers an interrupt which pends a new execution of the communication interrupt. The running task calls a function to signal that its execution has finished, and this function stops the hardware timer and executes the communication interrupt if it is pending. Besides that, the solution requires a shared variable through which the control task signals that it is running. That function stops the hardware timer, and it pends a DALI interrupt, to be executed. A visualization of when functions are called, in both scenarios, is shown in Figure 7.11.

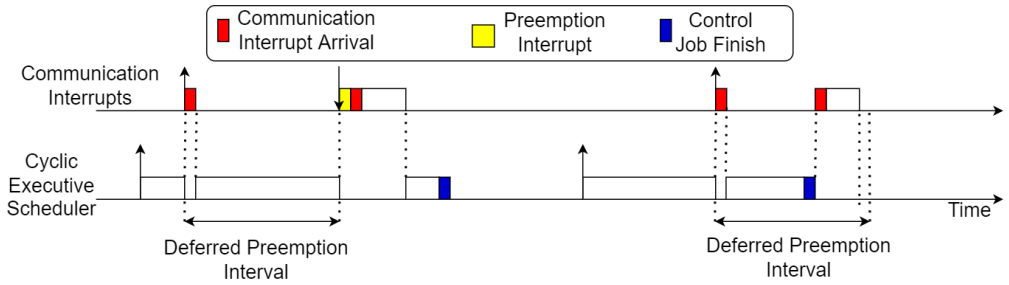


Figure 7.11: Limited Preemption Implementation

7.3.1 Overhead

Because the control tasks and the DALI interrupt are time sensitive, the overhead of the implementation is important. The overhead measurements can be seen in Table 7.4. These measurements can be used to set-up the delay interval when the implementation is integrated with the original system. The situation in which the delay interval matters is when the DALI interrupt needs to be triggered while the control task is still executing. In that case, two executions of the arrival function and one of the preemption interrupt will take place. This makes a maximum overhead of $1263 + 1263 + 415 = 2941(ns)$. The minimum interarrival time of the DALI interrupt is 18.66 us, with an execution time of about 7.5us. With an overhead of 2.91us, the maximum deferrable interval that ensures that there are no lost edges is 8.219us. To leave room for errors, an interval of 8us was chosen. When the driver is transmitting a message, the deadline of the DALI interrupt is 100us, with an execution time of 3.5 us. To meet this deadline, the delay interval is set to $100 - 2.941 - 3.5 = 93.559 \approx 93us$.

Function	Min	Max	Mean	Sdev
Arrival	412	1263	575	30
Finalization	804	2293	808	39
Preemption Interrupt	413	415	414	0

Table 7.4: Limited Preemption Overhead (ns)

7.3.2 Response time

An interesting measure of the impact of the limited preemption solution is the response time of the control tasks and of the DALI interrupt. The response time was measured when receiving and transmitting DALI messages, with limited preemption activated and deactivated. The measured control task was the buck controller. The results can be seen in Table 7.5. It can be seen that in the worst case, the response time of the buck controller is higher when using limited preemption. This is due to the fact that in the worst case, limiting the preemption just delays the execution of the DALI interrupt, so the controller experiences interference from both the implementation of limited preemption, as well as from the DALI interrupt. However, the mean response time was similar. The response time of the DALI interrupt is increased. This is to be expected, and it is one of the effects of the algorithm.

7.3.3 Light Flicker

Light flicker is an indirect measure of the quality of service provided by the buck controller. The test procedure for limited preemption is the same as for the server implementation, testing without and with DALI messages sent as fast as possible. Two variants of limited preemption were tested. The difference between the two is whether they are aware of the state of the communication

Function	Min	Max	Mean	Sdev
Buck Controller without limited preemption	21.28	33.91	21.79	0.59
Buck Controller with limited preemption	21.44	37.44	21.88	0.93
DALI Interrupt without limited preemption	3.96	9.1	7.21	0.85
DALI Interrupt with limited preemption	10.15	21.62	9.865	4.182

Table 7.5: Response times with limited preemption (us)

protocol. The first variant is not aware of that, and it uses the 8us delay interval all the time. The second variant is aware of the state of the protocol, and changes the delay amount based on that. Thus, when the driver is transmitting a message, the delay interval is changed to 93us.

The results can be seen in Table 7.6. Furthermore, Figure 7.12 shows the largest and the average amplitude for each frequency, when the system is under DALI message stress. Figure 7.12 shows the case in which there is no limited preemption, and the one when the state is not taken into account. The figures correspond to Table 7.6d and Table 7.6e.

The results indicate that there is no influence on the flicker when there are no DALI messages received, which is to be expected. When the system is under stress, there is a clear improvement when using limited preemption, of up to 0.8 dB. Moreover, when comparing the stateless and stateful implementations, there is no significant difference between them. It is interesting to note that the stateless solution has a lower average over the top ten frequencies with the largest amplitude, but the stateful implementation has a lower maximum peak amplitude.

While the reduction in light flicker is visible, the overhead of the implementation is significant compared to the computation time of the DALI interrupt. This reduces the impact this solution has. Moreover, this points to the fact that the original implementation is already quite performant.

Frequency (Hz)	Maximum peak(dB)
47	-64,1
45	-66,51
39	-66,82
46	-67,11
48	-67,57
44	-68,1
38	-68,37
42	-68,39
25	-68,94
43	-68,95
Avg	-67,486

(a) Preemption not limited without messages

Frequency (Hz)	Maximum peak(dB)
48	-66,01
44	-66,56
47	-66,91
46	-67,45
45	-67,71
41	-67,95
43	-68,17
32	-68,62
39	-68,75
40	-69,07
Avg	-67,72

(b) Stateless limited preemption without messages

Frequency (Hz)	Maximum peak(dB)
47	-66,62
48	-66,68
46	-67,27
41	-67,69
43	-67,72
27	-67,79
42	-68
45	-68,13
44	-68,27
26	-69,3
Avg	-67,747

(c) Stateful limited preemption without messages

Frequency (Hz)	Maximum peak(dB)
47	-66,74
41	-67,08
48	-67,18
45	-67,32
39	-67,51
46	-68,16
42	-68,32
44	-68,64
36	-68,72
43	-68,73
Avg	-67,84

(d) Preemption not limited with messages

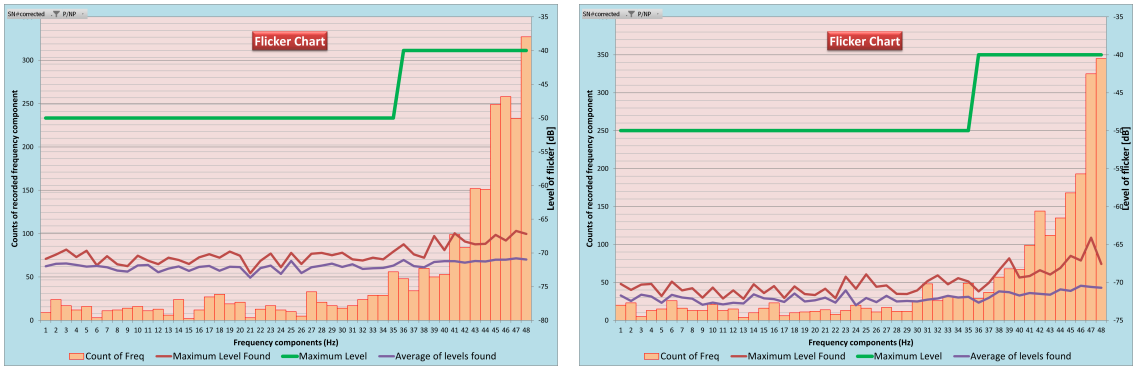
Frequency (Hz)	Maximum peak(dB)
48	-66,54
47	-67
45	-67,64
46	-67,88
41	-68,19
44	-68,39
43	-68,75
40	-69,37
34	-69,43
42	-69,6
Avg	-68,279

(e) Stateless limited preemption with messages

Frequency (Hz)	Maximum peak(dB)
45	-67,1
48	-67,26
47	-67,3
44	-67,81
43	-67,94
46	-68,24
37	-68,66
42	-68,73
32	-68,83
41	-68,95
Avg	-68,082

(f) Stateful limited preemption with messages

Table 7.6: Light Flicker Tests of Limited Preemption



(a) Flicker without limited preemption

(b) Flicker with limited preemption

Figure 7.12: Influence of limiting preemption on the system

Chapter 8

Reflection and Conclusion

This chapter contains the evaluation of the solutions as well as reflections on the applicability of the results. The problem investigated was the interference between a task that implements the communication protocol and tasks that need to be executed periodically and with as little jitter as possible. The maximum jitter accepted by a controller in a period of 250 us was estimated to be 30 us, as shown in section 7.1. The concrete use case is the integration of the DALI protocol with control tasks executed on a smart lighting driver. The communication protocol has to meet its timing requirements, and its driver requires processor time as it does not benefit from direct memory access or hardware decoders. Control tasks are affected by two types of jitter. The first one is release jitter, which happens when a task begins to execute later than its activation. The second type is response time jitter, which makes the time between collecting input and releasing its output variable.

8.1 Research questions

In the beginning, a number of research questions were formulated with the goal of answering them. A list of the questions can be found below, together with the answer provided by this thesis.

1. How can the system's tasks be modelled, and what properties and requirements can be derived from the model?

   The system can be modelled as a set of tasks. There are multiple types of tasks in the system. There are strictly periodic tasks with deadlines equal to their period. Other types of tasks are periodic and have to be executed with reduced jitter but do not have a determined deadline. There are also sporadic communication tasks with hard deadlines dictated by the constraints of the DALI protocol. Moreover, there are a number of dependencies between different tasks. The system model is useful in the theoretical analysis that was undertaken.

2. Which scheduling policy may produce a schedule for the tasks running on the LED driver, such that all tasks meet their deadlines?

   Deadline monotonic and earliest deadline first scheduling policies both produce a schedule in which the tasks meet their deadline, but the jitter of control-related tasks may be very large in the worst-case scenario. That said, using such a scheduling algorithm may have other non-functional advantages such as improved maintainability and extendability.

3. What techniques can be used to control the interference between control and communication-related tasks?

   Server algorithms and limited preemption are potential techniques to control this interference. However, for the analysed use case, only limited preemption yields a reduction in interference.

4. How can the policy producing a valid schedule with the least amount of interference be implemented on the targeted specific platform?

   Server algorithms and limited preemption can be implemented with the use of a hardware timer inside a C module. Deadline monotonic scheduling can be implemented with an operating system. However, this implementation showed too much overhead for our specific use case.

5. How does the produced implementation compare with an existing implementation in terms of interference and response time?

   The interference, measured indirectly in terms of light flicker, is improved by 0.8 dB when using limited preemption. The response time of the control tasks has not seen a significant improvement on average. This is due to the additional execution time added by the implementation in the worst-case scenario.

8.2 Summary of the investigated solutions

Several potential solutions were evaluated. They aimed at controlling the way the DALI interrupt triggers. The goal was to control the interference between the interrupt and control tasks.

8.2.1 Scheduling Algorithms

The first attempt was to change the scheduling algorithm used for the control tasks. The expectation was that a different scheduling policy would be able to take advantage of the difference between the execution time of the DALI interrupt and its deadline. Two possible policies were analysed theoretically. They are both priority-based algorithms.

Priority-based algorithms use priorities to choose the running task. A static priority algorithm keeps priorities fixed at run-time. A dynamic priority algorithm can change the priorities of tasks depending on the situation. The algorithms that were examined were a static priority algorithm, deadline monotonic scheduling, and a dynamic priority algorithm, earliest deadline first.

The result of the investigation was that under both of them, control tasks exhibited greater jitter than with the original implementation. This was due to tasks suffering interference from both the DALI interrupt and other tasks in the system. Ultimately, the short deadline of the DALI interrupt prevents those scheduling algorithms from improving the situation.

8.2.2 Server

Another possible solution is to place the DALI interrupt under the management of a server. While this ensures that the interference of the DALI interrupt on the control tasks is limited, the nature of the communication protocol limits its applicability in this use case. All DALI interrupts need to be executed before another one arrives, as new captured edges overwrite old values. Thus, in the situation when the server would actually limit the number of DALI interrupts, there is the risk of not implementing correctly the communication protocol. When the server does not limit the number of DALI interrupts, it just adds unnecessary overhead to the system.

There are situations in which a server would be needed, but that depends on the nature of the managed tasks. Tasks managed by a server should have soft or firm real-time requirements. One example would be tasks whose jobs may be skipped or that require only a number of jobs to be executed in any time interval. Another example would be tasks that have an execution time depending on their input data to prevent them from holding the processor for too much time.

One potential application of servers in the use case is to partition the computation time of the cooperative scheduler for different categories of features. This ensures that features can be added to the product without starving the already existing ones. It also provides a way for managing the existing features, allocating more computation time to the ones with higher priority.

8.2.3 Limited Preemption

The most promising solution is to limit the preemption of control tasks by the DALI interrupt. This allows the control tasks to finish their execution as long as the DALI driver is guaranteed to meet its constraints. If there is enough room for delay, the DALI interrupt will be executed as soon as the current running control task finishes. Moreover, the amount of time the DALI interrupt is delayed can be adjusted based on the state of the driver. This takes advantage of the larger deadline of the DALI interrupt when the driver is transmitting.

The improvement in light flicker was only marginal. This is because the DALI interrupt has a small computation time compared to the implementation, making the overhead significant. Thus, the implementation overhead offsets the theoretical reduction in light flicker. This solution is best used when a communication protocol cannot miss any incoming edges and does not have mechanisms ensuring redundancy, such as checksums for error correction or detection.

8.3 Future Work

There are several aspects of the work that may be interesting to be continued in a future investigation. The first one is the effect of different phasing of tasks. Currently, tasks are assumed to be released at the same time during what is called a critical instant. This assumption was made in order to make the solution more generally applicable. However, tasks can be designed to have different phasing. This means that tasks with the same period may not necessarily interfere with each other, as they would be interleaved. Introducing more constraints, one of which is fixing the tasks' phasing, may reveal that a solution involving a different scheduling algorithm is feasible. It is therefore an interesting point of investigation.

The second point of improvement is investigating the effect of a more powerful processor. It has been shown that the Cortex-M4 processor is faster. Redoing the experiments involving the measurement of light flicker can provide valuable data. However, those require a driver using such a powerful processor, which was not available at the time of writing.

The third aspect is the investigation of the scheduling behaviour of a communication protocol that benefits from error correction or detection. Such a protocol may be able to afford missing jobs, which changes its real-time properties. Therefore it would be valuable to investigate the applicability of the solutions presented in this thesis for such a protocol.

Bibliography

[1] IEC Central Secretary, "Digital addressable lighting interface - Part 101: General requirements - System components", en, International Electrotechnical Commission, Standard IEC 62386-101:2014+AMD1:2018, 2018. [Online]. Available: https://webstore.iec.ch/publication/63236 (visited on 23/12/2021).

[2] "DALI devices". en. (21st Mar. 2022), [Online]. Available: https://www.nvcuk.com/media/technical/library/dali-devices-16_9_cropped.png.png (visited on 21/03/2022).

[3] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 3rd. Springer Publishing Company, Incorporated, 2011, ISBN: 1461406757.

[4] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment", *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973, ISSN: 0004-5411. DOI: 10.1145/321738.321743. [Online]. Available: https://doi.org/10.1145/321738.321743.

[5] E. Bini and G. Buttazzo, "Rate monotonic analysis: The hyperbolic bound.", *Computers, IEEE Transactions on*, vol. 52, pp. 933–942, Aug. 2003. DOI: 10.1109/TC.2003.1214341.

[6] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks", *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, 1982, ISSN: 0166-5316. DOI: https://doi.org/10.1016/0166-5316(82)90024-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0166531682900244.

[7] N. Audsley, A. Burns and A. Wellings, "Deadline monotonic scheduling theory and application", *Control Engineering Practice*, vol. 1, no. 1, pp. 71–78, 1993, ISSN: 0967-0661. DOI: https://doi.org/10.1016/0967-0661(93)92105-D. [Online]. Available: https://www.sciencedirect.com/science/article/pii/096706619392105D.

[8] W. A. Horn, "Some simple scheduling algorithms", *Naval Research Logistics Quarterly*, vol. 21, pp. 177–185, 1974.

[9] S. K. Baruah, L. E. Rosier and R. R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor", *Real-time systems*, vol. 2, no. 4, pp. 301–324, 1990.

[10] M. Spuri, "Analysis of deadline scheduled real-time systems", Ph.D. dissertation, Inria, 1996.

[11] J. P. Lehoczky, L. Sha and J. K. Strosnider, *Enhanced aperiodic responsiveness in hard real-time environments.* Conference contribution, Dec. 1987. [Online]. Available: http://www.scopus.com/inward/citedby.url?scp=0023534382&partnerID=8YFLogxK.

[12] B. Sprunt, L. Sha and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems", *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, Jun. 1989, ISSN: 1573-1383. DOI: 10.1007/BF02341920. [Online]. Available: https://doi.org/10.1007/BF02341920.

[13] M. Spuri and G. C. Buttazzo, "Scheduling aperiodic tasks in dynamic priority systems", *Real-Time Systems*, vol. 10, pp. 179–210, 2004.

[14] Z. Deng, J.-S. Liu and J. Sun, "A scheme for scheduling hard real-time applications in open system environment", in *Proceedings Ninth Euromicro Workshop on Real Time Systems*, 1997, pp. 191–199. DOI: 10.1109/EMWRTS.1997.613785.

[15] Spuri and Buttazzo, "Efficient aperiodic service under earliest deadline scheduling", in *1994 Proceedings Real-Time Systems Symposium*, 1994, pp. 2–11. DOI: 10.1109/REAL.1994.342735.

[16] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems", in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, 1998, pp. 4–13. DOI: 10.1109/REAL.1998.739726.

[17] A. BALSINI, "Novel resource management mechanisms for real-time scheduling in the linux kernel", 2015.

[18] K. Jeffay and D. Stone, "Accounting for interrupt handling costs in dynamic priority task systems", in *1993 Proceedings Real-Time Systems Symposium*, ser. 1993 Proceedings Real-Time Systems Symposium, 1993, pp. 212–221. DOI: 10.1109/REAL.1993.393497. [Online]. Available: https://doi.org/10.1109/REAL.1993.393497.

[19] J. A. Santos Jr. and G. Lima, "Sufficient schedulability tests for edf-scheduled real-time systems under interference of a high priority task", in *2012 Brazilian Symposium on Computing System Engineering*, 2012, pp. 131–136. DOI: 10.1109/SBESC.2012.33.

[20] C. Lozoya, M. Velasco and P. Marti, "The one-shot task model for robust real-time embedded control systems", *IEEE Transactions on Industrial Informatics*, vol. 4, no. 3, pp. 164–174, 2008. DOI: 10.1109/TII.2008.2002702.

[21] A. Cervin and J. Eker, "The control server: A computational model for real-time control tasks", in *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, 2003, pp. 113–120. DOI: 10.1109/EMRTS.2003.1212734.

[22] G. C. Buttazzo, M. Bertogna and G. Yao, "Limited preemptive scheduling for real-time systems. a survey", *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2013. DOI: 10.1109/TII.2012.2188805.

[23] "FreeRTOS". (), [Online]. Available: https://www.freertos.org/index.html.

[24] R. Kase, *Efficient scheduling library for freertos*, 2016.

[25] G. Oliveira and G. Lima, "Evaluation of scheduling algorithms for embedded freertos-based systems", in *2020 X Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2020, pp. 1–8. DOI: 10.1109/SBESC51047.2020.9277851.

[26] "MicroC/OS". (), [Online]. Available: https://weston-embedded.com/micrium-kernels.

[27] K.-M. Cho, C.-H. Liang, J.-Y. Huang and C.-S. Yang, "Design and implementation of a general purpose power-saving scheduling algorithm for embedded systems", in *2011 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC)*, 2011, pp. 1–5. DOI: 10.1109/ICSPCC.2011.6061645.

[28] "Keil RTX". (), [Online]. Available: https://www2.keil.com/mdk5/cmsis/rtx.

[29] "VxWorks". (), [Online]. Available: https://www.windriver.com/products/vxworks.

[30] "EmbOs". (), [Online]. Available: https://www.segger.com/products/rtos/embos/.

[31] Y. Mazzi, A. Gaga and F. Errahimi, "Benchmarking and comparison of two open-source rtoss for embedded systems based on arm cortex-m4 mcu", *Indian Journal of Science and Technology*, vol. 14, no. 16, pp. 1261–1273, 2021.

[32] I. Ungurean and N. C. Gaitan, "Performance analysis of tasks synchronization for real time operating systems", in *2018 International Conference on Development and Application Systems (DAS)*, 2018, pp. 63–66. DOI: 10.1109/DAAS.2018.8396072.

[33] W. Hofer, D. Lohmann and W. Schröder-Preikschat, "Sleepy sloth: Threads as interrupts as threads", in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011, pp. 67–77. DOI: 10.1109/RTSS.2011.14.

[34] S. Pinto, J. Pereira, D. Oliveira *et al.*, "Porting sloth system to freertos running on arm cortex-m3", Jun. 2014. DOI: 10.1109/ISIE.2014.6864903.

[35] D. Katcher, H. Arakawa and J. Strosnider, "Engineering and analysis of fixed priority schedulers", *IEEE Transactions on Software Engineering*, vol. 19, no. 9, pp. 920–934, 1993. DOI: 10.1109/32.241774.