Politecnico di Torino

Master's degree in electronic engineer

Master's degree Thesis

Core Concurrency in multi-core system and scheduler development for STRED architecture

Supervisors

Prof. Maurizio Martina

Luca Longhi @STMicroelectronics

Candidate

Klides KABA

October 2024

Abstract

The growing demand for high computational performance and energy efficiency has made multi-core architectures a fundamental component in modern computing systems. From mobile devices to data center servers, multi-core systems allow for the execution of multiple operations simultaneously, improving speed and processing capacity without increasing the clock frequency. However, managing concurrency for resource access can present complex challenges that require innovative and optimized solutions.

This thesis aims to address two main objectives in the context of multi-core and single-core architectures, with a particular focus on concurrency management and performance optimization. The first objective is the development of methodologies for managing concurrency in a specific multi-core architecture within a SER-DES (serializer-deserializer) environment. This process begins with the analysis of a specific industrial case of interest and then attempts to generalize the solutions found to improve the efficiency and robustness of concurrent operations. In this first case, the goal is to explore various synchronization algorithms and inter-core communication strategies, evaluating their robustness and criticalities.

The second objective is the development of a scheduler for the STRED_L architecture (property of STMicroelectronics), a software component that, based on the type of implemented algorithm, manages the order and duration with which processes get access to the CPU, allowing for efficiency optimization and responsiveness of the system. This part will be developed for the single-core configuration, which is able to work at a frequency four times higher than the multi-core mode, which ultimately is a quad-core.

The main question we want to answer in this case is: Is a quad-core system, where up to four tasks can work in parallel, better than a single-core system with scheduling, where operations are executed in series but can run at a speed four times higher? Which of these solutions allows for higher performance, and, most importantly, how significant is this difference between them? By answering this, this work aims to make some considerations about which one can ultimately provide the best balance in terms of area, performance, power consumption, and cost.

The developed scheduler will be tested in specific industrial scenarios and will feature a Round-Robin algorithm. The analysis will start with simple and limited solutions and will gradually become more complex, covering some possible cases of industrial application interest.

Contents

1. Introduction

This thesis was conducted at STMicroelectronics under the supervision of Luca Longhi, inside the SW/FW team led by Stefano Antoniazzi.

The work environment for this project is represented by the S112, a SerDes application system developed by STMicroelectronics. The focus of this work will be on the microcontroller subsystem within this environment, which has the capability to work in both single-core and quad-core configurations.

The first part of this thesis will concentrate on the multi-core mode of the microcontroller, addressing a specific problem related to resource sharing and synchronization mechanisms. Given that the four cores run in parallel, access to shared resources must be carefully managed. The aim is to explore synchronization techniques between the cores to develop a more general and efficient approach.

The second part of the thesis will focus on designing a scheduler for the single-core configuration, specifically for the STRED_L architecture, which is the core implemented in the microcontroller produced by STMicroelectronics. The aim is not only to create a more dynamic environment for the single core, making it more suitable for real-time operations, but also to compare its performance with the quad-core configuration. It is noteworthy that the quad-core configuration operates at only one-fourth of the nominal frequency.

By estimating the impact of the overhead introduced by the scheduler, we can then compare the two configurations in terms of performance and make some considerations on the pros and cons of single-core versus quad-core setups. Without a scheduler, the comparison would be skewed, as the advantages of the multi-core configuration would be too significant. The multi-core setup not only offers superior performance by running different jobs in parallel but also provides better responsiveness, making it more suitable for tasks with strict deadlines.

By addressing these objectives, this thesis aims to enhance the efficiency and functionality of the microcontroller subsystem in both single-core and multi-core configurations.

The documentation will begin with an introduction to the architecture, with a particular emphasis on the specific blocks utilized in this project. The subsequent chapter will address the resource-sharing problem in the quad-core configuration within a specific case of interest. Various approaches will be presented, discussing their pros and cons, until the selection of the final solution.

The methodology for core synchronization will aim to achieve the highest performance while exploiting the idle states of the cores.

The final chapter will focus on the secondary objective of this work: the development of a scheduler for the STRED_L core architecture. The primary goal will be to implement a scheduler capable of performing context switches between tasks effectively, thereby creating a more responsive system. The work will begin from simple solutions until a more complete and complex one. The last sections will then provide the results of the context switch impact on performances.

1.1 Brief description of a Serializer/Deserializer (SerDes) system

Since the work environment is represented by the S112 SerDes application, it is pertinent to provide a brief description of a SerDes system, even though it is not necessary for the comprehension of the involved project.

A SerDes environment stands for Serializer/Deserializer. It is a functional block used in high-speed communications that can convert between serial data and parallel interfaces, and vice versa. The primary use of a SerDes is to enable data transmission over a single line, thereby minimizing the number of I/O pins and interconnects in chips where large data transactions are required and there are constraints regarding the number of I/O pins [10].

A basic SerDes system is composed of two functional blocks:

1. **Parallel-In Serial-Out (PISO) Block**: Also known as the parallel-to-serial converter, this block functions as the transmitter of the system. It converts parallel data into a serial data stream for transmission over a single line.
   Serial-In Parallel-Out (SIPO) Block: Also referred to as the serial-to-parallel converter, this block functions as the receiver of the system. It converts the incoming serial data stream back into parallel data [9].

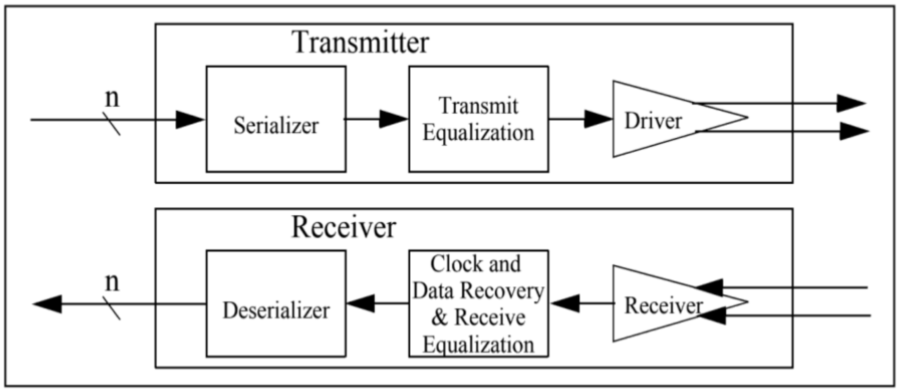


Figure 1: block diagram of a typical high-speed SerDes [1].

Given the very high-speed transactions involved, it is essential for these systems to implement equalization techniques for data recovery. Equalization helps to mitigate signal degradation and ensures accurate data transmission and reception [1].

2. Architecture description

Due to confidentiality and secrecy obligations, the detailed architecture of this advanced SerDes system cannot be fully disclosed. However, to facilitate understanding, it is sufficient to highlight some general characteristics of the system and the key components involved. An overview of the system will be provided in the following chapter, while a more detailed description of the various functional blocks will be given when discussing the specific parts of the work in which they got involved.

As reported in its user manual, the S112 is a multi-channel SerDes system composed of multiple Data Slice Quads (DS-Quad), each of which is composed of:

- **4 Data Slices**: they are the same physical instance and are mainly composed of 3 blocks: a high-frequency Clock generation block, the Serializer which corresponds to the Transmitter, and the Deserializer which represents the receiver.

- **A micro subsystem** shared among the 4 Data Slices that assists in the calibration of the SerDes and offers an improvement in flexibility of use.

The other fundamental block of the Quad is the *Clock Slice Twin*. This is an aggregation of:

- **Clock Slice (CS)**: also in this case they are the same exact physical instance and are dedicated to the distribution of the clock through all the quad.
- **Clock Slice Auxiliary (CSA):** it contains auxiliary blocks like a thermal sensor, an auxiliary ADC and impedance Compensation [3].

2.1 Clock Slice Auxiliary

A particular focus will be dedicated to the Clock Slice Auxiliary block, as it has been the primary subject of the first part of this work. As previously mentioned, this block is responsible for impedance compensation, includes a thermal sensor for calibration and adaptation purposes, and features an auxiliary ADC, among other functionalities.

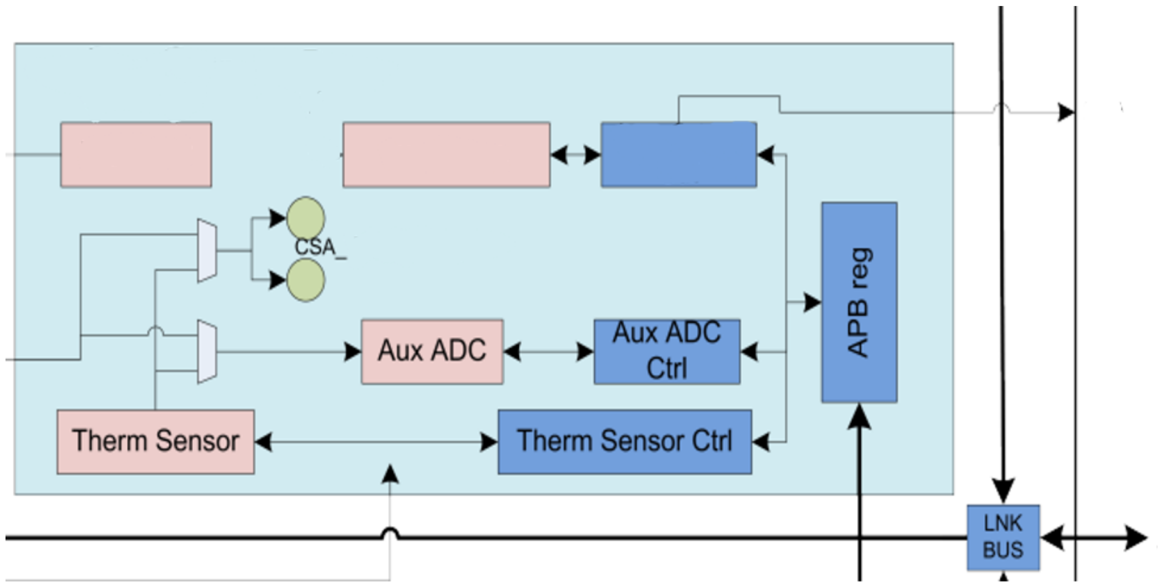


Figure 2: CSA block diagram.[3]

The most important blocks to focus on here are the Thermal Sensor and the Auxiliary ADC. Both components are connected to control registers, which can be accessed by the microcontroller and utilized to configure and set these blocks. For example, these control registers allow the microcontroller to turn on the Thermal Sensor or start the ADC [3].

2.2 Micro sub-system architecture

The micro sub-system integrated in S112 SerDes is composed of:

- A very low power core processor, the STRED_L architecture.
- A program RAM.
- A data RAM.
- An interrupt controller for managing interrupts.
- 1 UART
- 1 LINK bus that allows to access the clock Slice memory map or the memory map of others Quad-slices.
- Multicore and single core capability that can be manually configured based on the type of application.

This Micro is able to configure and control multiple Data Slices in parallel, thanks to the multicore configuration, to the timers that are available to support this task [3].

Below a Block diagram of the sub-system is reported.

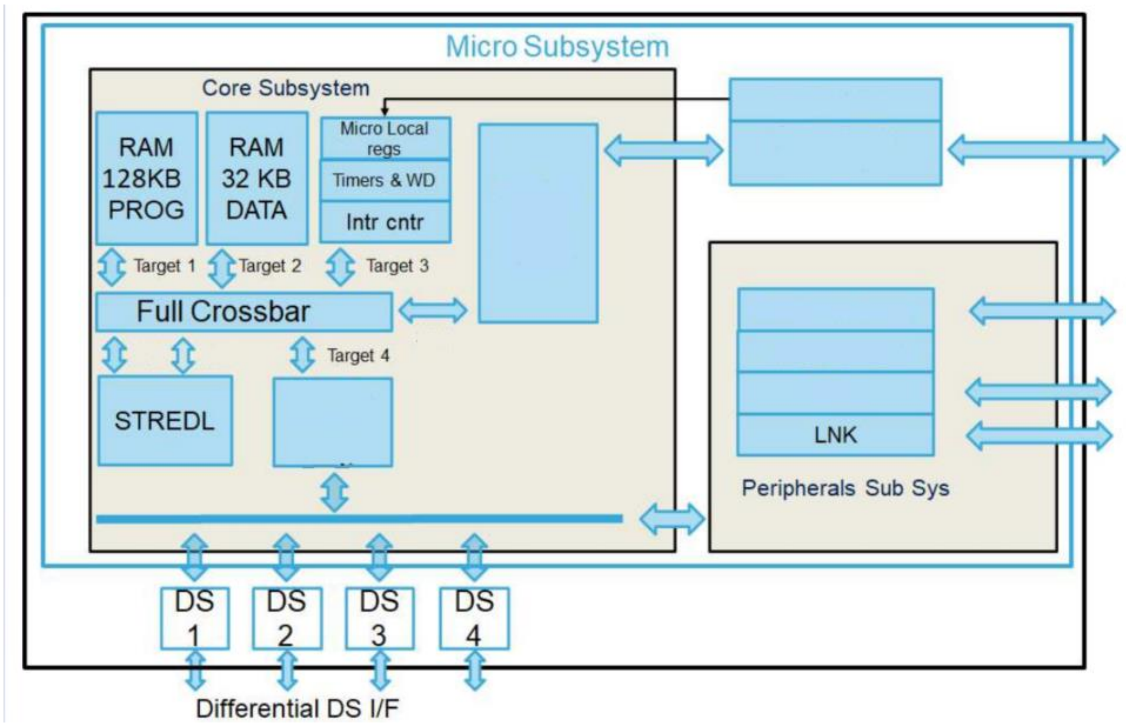


Figure 3: Block diagram of the micro subsystem in S112(single core-configuration) [4].

As the diagram shows, the code memory and data memory are connected to the core through a 32-bit Full Crossbar, along with all other peripherals. The crossbar functions as an arbiter, managing access requests from all connected devices. The necessary blocks showed in the image will be explained more in detail later, as follows the description of the specific activities in which they got involved.

In the multi-core configuration, all four STRED_L core become active, sharing all peripheral resources among them and getting referred to as *core0*, *core1*, *core2*, *core3* [3].

3. Thermal Sensor reading in multicore.

The Data Slices blocks are highly complex, comprising various groups of analog and digital components that must communicate at very high speeds. To ensure proper signal restoration and communication, advanced signal processing and quantization techniques are required [3]. Numerous variables must be considered to dynamically adjust the behavior and parameters of the involved algorithms and devices in real time. One critical variable is temperature, which can be monitored using the thermal sensor located in the CSA. The main problem in this case is that the thermal sensor is unique and as mentioned before the Quad Slice is an aggregation of four different Data Slices and each of them might need to retain an updated value of the temperature. In the multi-core configuration, each core is dedicated to a specific Data Slice of the Ser-Des system being able to directly access its memory map [2].

A resource sharing issue arises in this context, as the four cores need to access the same resource, potentially interfering with each other's operations and hindering the correct completion of the task. Since a shared memory region at which all cores can access is also present in the memory map, this solution could be very simple; we could assign the task to read the thermal sensor to a specific core (for example *core0*), which will then save the value in the shared memory region, where all the other cores can access, read the value and make it available for the relative Data Slice.

This is undoubtedly the simplest solution, but it introduces an overhead for the specific core tasked with reading the thermal sensor. The process involves waiting for access to the LNK BUS and then waiting for the completion of the analog-to-digital (AD) conversion. This task is not immediate and must be repeated periodically, as the temperature is not constant, thereby introducing a non-negligible overhead for the relative core (in this example core0) [3].

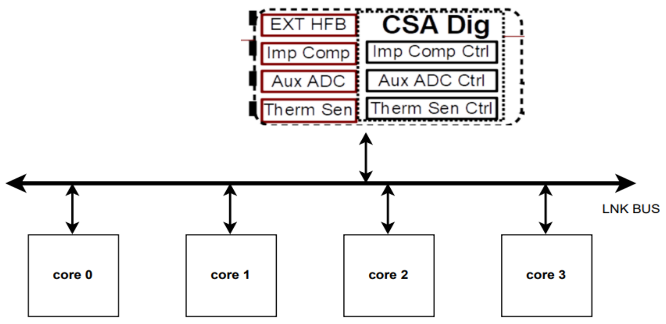


Figure 4: schematic of the relation between the clock slice auxiliary and the four cores.

Our goal is to find a general solution that allows the four cores to operate independently, balancing the workload and improving overall system efficiency. The main idea is for the least busy core to take charge of the task and complete it. In other words, the first core able to handle the pending request to read the temperature will perform the task for all the other cores. This decision will not be made by the user but will be an automatic result of the core's activity. This approach enables all cores to remain operational across all channels, without penalizing any single core, and allows for a more predictable and deterministic system.

To achieve the mentioned advantages, the cores will need to run the same code (which translates to executing the same instructions), ensuring that there are no branches where only specific cores execute certain tasks. This uniformity in code execution ensures balanced workload distribution and system efficiency. Below, we will begin by listing and briefly explaining some of the peripherals required for accessing and reading the thermal sensor, followed by the applied methodologies.

3.1 Link Bus

The link bus is a modified version of the IIC bus, which allows to access the memory map of others quad in a composition of more quads or to access the Clock Slice memory map from any quad.

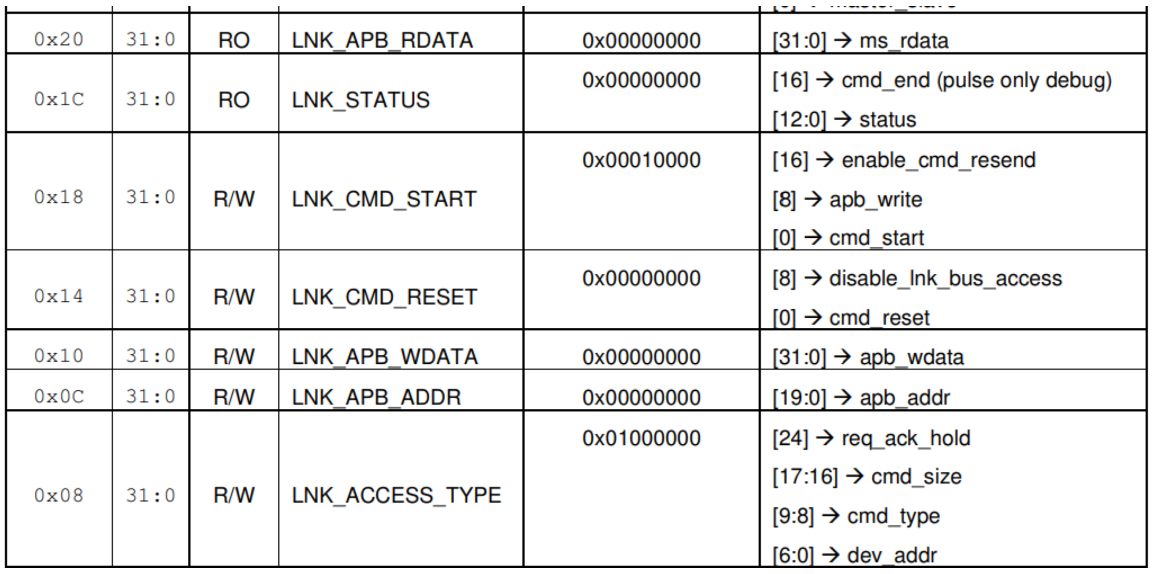
The IIC features supported also by the link bus are:

- Multi master mode: this means more than one master device can initiate communication and control the bus.
- 7 bits addressing.
- Arbitration: when two or more master device attempt to take control of the bus simultaneously, an arbitration process determines which master gets control.
- Repeated start: this allows device to maintain control of the bus and continue communication with either the same or a different slave device.

The link bus can be configured and controlled with the micro by accessing the related memory map and setting the required values on the registers. These register allow to select the master(the QUAD) and the slave (CS), by specifying the device address bits in the related register.

The most important registers for managing communication and initiating a transaction with the link bus, once the master and slave devices have been selected, are the STATUS registers. The primary register is the LNK_STATUS, which holds several critical flags, including:

1. LNK_STATUS[0] → 'status_cmd_done': it indicates the end of the actual access.
2. LNK_STATUS[1] → 'bus_busy': it indicates when the bus is busy and can't start new transactions [4].



0x20	31:0	RO	LNK_APB_RDATA	0x00000000	[31:0] → ms_rdata
0x1C	31:0	RO	LNK_STATUS	0x00000000	[16] → cmd_end (pulse only debug) [12:0] → status
0x18	31:0	R/W	LNK_CMD_START	0x00010000	[16] → enable_cmd_resend [8] → apb_write [0] → cmd_start
0x14	31:0	R/W	LNK_CMD_RESET	0x00000000	[8] → disable_lnk_bus_access [0] → cmd_reset
0x10	31:0	R/W	LNK_APB_WDATA	0x00000000	[31:0] → apb_wdata
0x0C	31:0	R/W	LNK_APB_ADDR	0x00000000	[19:0] → apb_addr
0x08	31:0	R/W	LNK_ACCESS_TYPE	0x01000000	[24] → req_ack_hold [17:16] → cmd_size [9:8] → cmd_type [6:0] → dev_addr

Figure 5: Link Bus register map [4].

3.2 ADC auxiliary

The auxiliary ADC, as previously mentioned, is located in the Clock Slice Auxiliary and is a fundamental block for reading the thermal sensor, as the sensor's original output is an analog value. The ADC converts this analog signal into a digital format and writes it to its dedicated registers, where it can be accessed by the microcontroller. Like the Link Bus, both the calibration and normal operation of the ADC are controlled by software.

Initially, the ADC needs to be powered on, and the thermal sensor must be selected as the current input for the ADC, since a preliminary multiplexer allows for different input sources.

A conversion can occur when the ADC's specific start signal is high. After a certain number of clock cycles from the start command, the ADC raises the end of conversion signal indicating that the output data in the output register is valid. When a new start conversion command is issued, the end of conversion signal resets until the new conversion is finished.[3]

3.3 Necessary steps for accessing ADC and Link Bus

To access the ADC, two types of LNK_BUS access functions are necessary: one for read mode and one for write mode. Below are reported the sequential steps used for accessing the LNK_BUS to start a transaction with the auxiliary ADC in the CSA:

1. Write in the dedicated LNK_BUS size register the size of the data to read/write (8/16/32 bits).
2. Define the address of the peripheral to be accessed (register address of the ADC)
3. Give start command for starting the transaction.
4. Wait until LNK_STATUS register shows end of transaction.
5. In case of read operation, return the read value which will be stored in a LNK_BUS register.

The core will continuously use these functions for accessing the registers of the ADC.

Now the steps for accessing and starting the ADC are showed (each step is implied making use of the LNK functions):

1. Turn-On the analog to digital converter.
2. Give start conversion command to ADC control register.
3. Wait for end of conversion by continuously reading ADC status register, until EOC bit is risen; now the data is valid for reading.
4. Turn-off ADC.

3.4 Strategies for Core Concurrency handling

Now that all the necessary devices involved in this task have been presented, we can begin analyzing the possible algorithmic approaches for implementing this task and handling core concurrency. The objective is not only to correctly manage the core concurrency involved in this task but also to design an efficient method for accessing a shared resource (in this case the thermal sensor), which value can be required by all the cores, starting from a practical case and try to generalize it for all similar scenarios. Different approaches will be considered analyzing the pros and the cons.

Different considerations in fact, will be made based on the purpose and final use scenarios of the temperature data, which may influence the choice of the best solution. Therefore, we will analyze different approaches and finally choose the one that better suits our application.

3.4.1 First approach: on-demand temperature request

One approach is to have each core start the thermal sensor and initialize the ADC conversion whenever they need the temperature: this is the so called on-demand request approach. It is important to consider that temperature is not a parameter that changes rapidly and does not need to be consistently updated. Depending on the running application, the temperature could be considered stable for 20 seconds, 30 seconds, one minute, or even longer.

This means that if a core needs the temperature a few seconds after another core has already started the ADC and obtained the temperature, it is unnecessary for this core to restart the process, as it will likely read the same value. It is worth noting that we are not considering applications that require continuous temperature tracking over a specific period or very high precision values, as this assumption may not hold in such scenarios.

For this purpose, each time a core reads the temperature, a timestamp is recorded. If a new core or even the same core requests the temperature within an interval during which the previous temperature value is still considered valid, there is no need to initiate a new conversion. Instead, the temperature can be directly read from the ADC output register or from the shared memory region.

Concurrency must be managed in this scenario, as multiple cores might simultaneously request to access the auxiliary ADC for reading the thermal sensor.

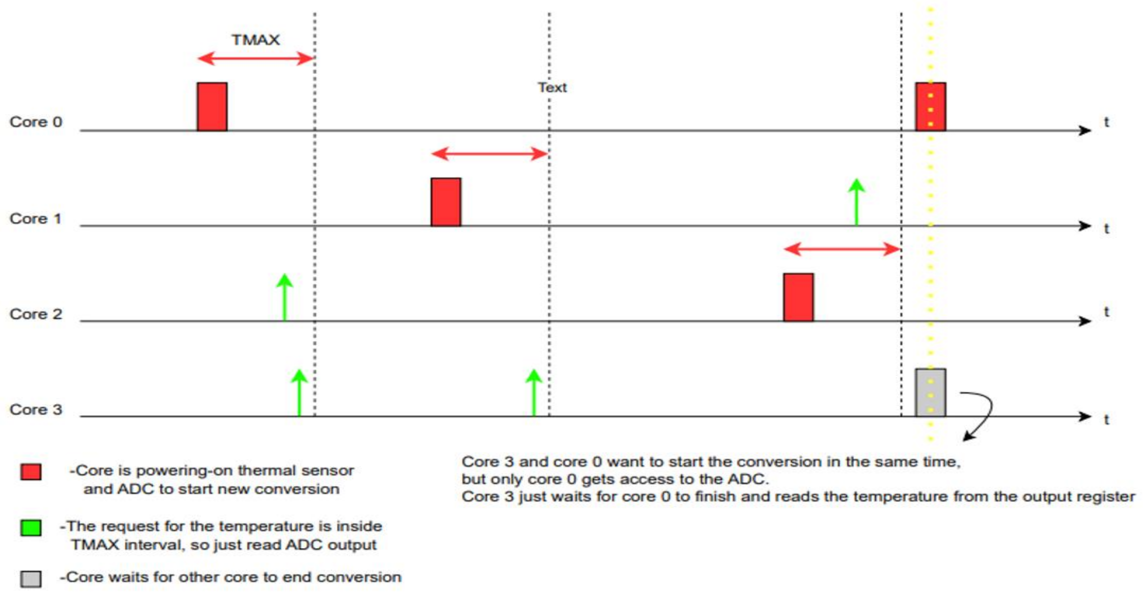

TMAX
Text
Core 0
Core 1
Core 2
Core 3
t
t
t
t
🟥 -Core is powering-on thermal sensor and ADC to start new conversion
🟩 -The request for the temperature is inside TMAX interval, so just read ADC output
⬜ -Core waits for other core to end conversion
Core 3 and core 0 want to start the conversion in the same time, but only core 0 gets access to the ADC.
Core 3 just waits for core 0 to finish and reads the temperature from the output register

Figure 6: graphic representation example of On-demand approach**.**

The critical issue with this approach is that if a specific core (for example, *core0*) has a higher demand for temperature readings compared to the other three cores over a certain period, *core0* will be solely responsible for reading the thermal sensor. This could lead to inefficient workload balancing, as we would not be utilizing other cores that might be idle during that period, potentially resulting in suboptimal performance.

On the positive side, this approach ensures that the thermal sensor is read only when necessary, avoiding unnecessary ADC conversions and conserving resources.

3.4.2 Second approach: Periodic temperature request

Another possible approach, which might be more advantageous in terms of performance for a different type of application, is to implement a periodic temperature read request. In this scenario, at regular intervals (for example, every 30 seconds), all cores receive a command to read the temperature value from the thermal sensor. A mechanism must be in place to ensure that only one core accesses the resource, with the fastest core to respond taking the request. This mechanism is illustrated in **Figure 7.**

In this setup, cores that do not secure the request will return to their execution. When a specific core subsequently needs the temperature, it can simply retrieve the value from the ADC output register or the shared memory region, as the value will still be valid.

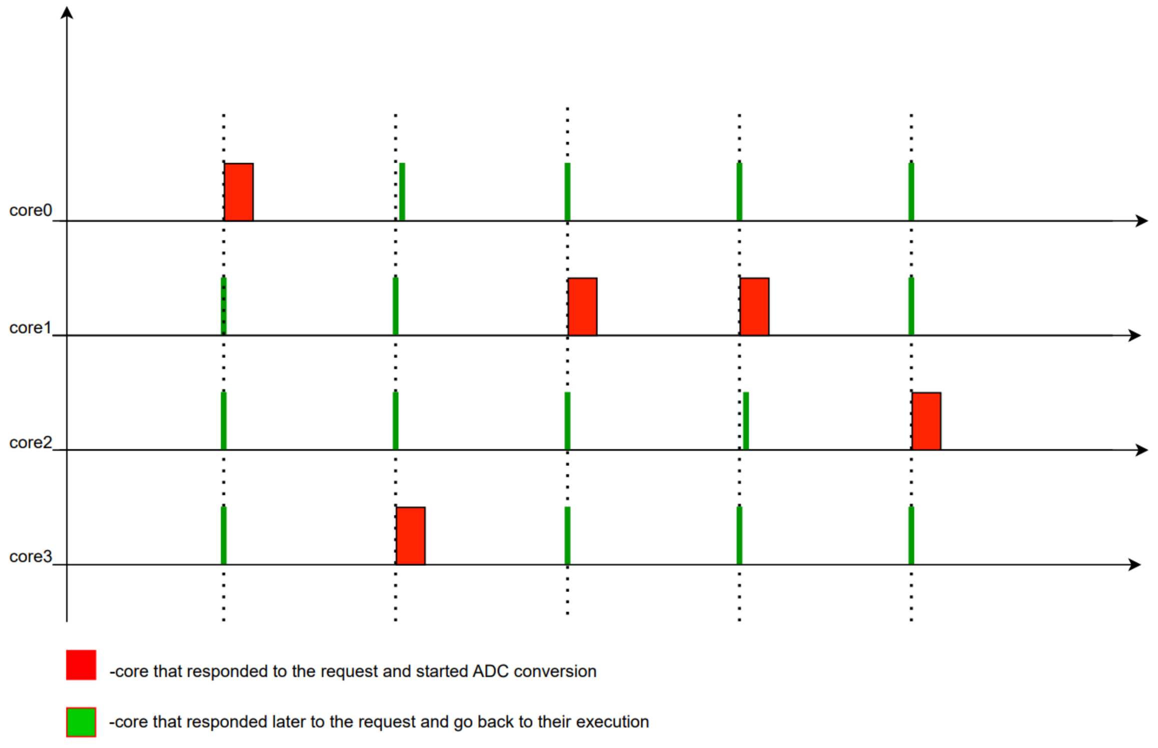

core0
core1
core2
core3
-core that responded to the request and started ADC conversion
-core that responded later to the request and go back to their execution

Figure 7: graphic representation periodic request approach.

The advantage of this solution is that it fully exploits workload balancing among the cores, as the process of reading the temperature will always be handled by the readiest core (i.e., the least busy one). However, this approach also has a downside: it requires a significant amount of energy to always ensure a valid temperature value. In environments where the temperature can be triggered by rapid changes, this needs a shorter intervals for periodic requests, leading to a higher frequency of thermal sensor accesses and ADC conversions. Consequently, the number of sensor readings and conversions could be much higher than the actual number of times the cores effectively requests the temperature.

3.4.3 Third Approach: Combination of periodic and On-demand request.

The third approach proposed in this chapter is more interesting and comprehensive. It combines the first two techniques discussed earlier, effectively balancing, and mitigating the critical aspects of the previous methods.

In this case, a periodic request is still sent to all cores, but only one core processes it. Unlike the previous methods, the time interval for these requests does not necessarily need to be less than or equal to the maximum duration in which the last temperature value can still be considered valid. This reduces the number of useless accesses.

When a core needs the temperature data, it checks if the last ADC (Analog-to-Digital Converter) conversion occurred within a valid timestamp. If the timestamp is valid, the core retrieves the temperature value from the registers. If not, the core starts a new ADC conversion and updates the temperature value. The next periodic request will use this event as the starting point of its period.

The graphic in the image below illustrates how this process works.

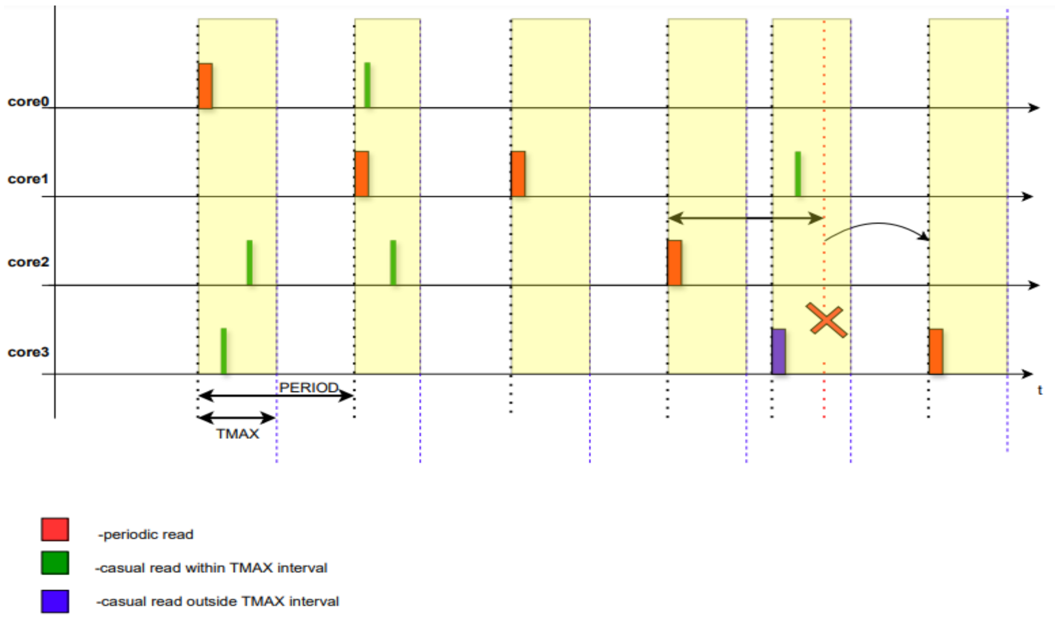

core0
core1
core2
core3
PERIOD
TMAX
t
-periodic read
-casual read within TMAX interval
-casual read outside TMAX interval

Figure 8: On-demand and periodic request implementation.

From this, we can see that every *PERIOD*, a new request to update the temperature value is sent to all the cores. As before, only one core handles the request while the others return to their executions.

The yellow area represents the valid timestamp within which the last update can still be considered valid. This means that if a core needs the temperature data (for example, *core2* and *core3* in the first yellow area), they will simply retrieve the last updated value.

The purple color indicates that a core (*core3* in this example) requested the temperature outside the *TMAX* interval. Since the temperature data cannot be considered valid in this case, the core starts a new ADC conversion. The next periodic request will occur a *PERIOD* after this event.

This approach leverages the workload balance among the different cores due to the periodic requests while simultaneously reducing the number of ADC accesses to closely match the actual number of times the temperature is needed by the cores.

This solution has ultimately been chosen for being implemented in the firmware.

3.5 Practical Implementation

As previously stated, the purpose is to maintain the same code for all the cores in order to balance the workload between all the cores, and the previously showed algorithm is in alignment with this, since no specific core is in charge of taking specific branches.

In this approach, we need to handle core concurrency for both periodic and on-demand requests:

1. **Periodic Requests**: We must ensure that only one core gains access to the auxiliary ADC. This prevents multiple cores from attempting to process the periodic request simultaneously.
2. **On-Demand Requests**: We need to consider that multiple cores might request the temperature at the same time. However, only one core can initiate the ADC conversion, while the others must wait for it to complete.

3.5.1 Synchronization Mechanisms

A secure synchronization mechanism in order to avoid race conditions between the cores is required when handling simultaneous requests.

As reported in the micro-subsystem manual, the micro itself features a dedicated synchronization mechanism using a specific register called RESOURCE_SYNC_REG when using multi-core configuration. This register is accessible by all cores and allows them to lock resources and release them at the end of an operation. Dedicated hardware ensures the proper behavior of this register.

This register is composed of nibbles, and each of them can be associated to specific resource(this a chosen by the user, there is not an hardware association between the resource and the nibble.

At reset, the generic nibbles show a value of 0xF, indicating that the resource is free and not locked by any core. Any write to the fourth bit of the nibble (bit[3]) represents a resource request by the core. If, at the moment of the request, the nibble is still free (value 0xF), it will latch the number of the requesting core (0x0, 0x1, 0x2, or 0x3). If the resource is already locked, the write is ignored. When a nibble is locked by a certain core X, any write to that nibble by other cores is ignored. Only the locking core can write back to the nibble, setting ‘bit[3]’ to '1', which frees the resource, and the nibble will then show '0xF'.

After a core attempts to lock the nibble, a read operation is required to verify if the request was successful. By reading the nibble, any core can determine which core is locking a particular resource [4].

3.5.2 Periodic request implementation

A primary challenge for this task is defining how to issue the command to start the conversion to all cores simultaneously at each defined period of time. A possible solution is to use a timer interrupt. As mentioned in previous chapters, the microcontroller has 5 timers available, each capable of triggering an interrupt [4].

The timers have different priority levels, meaning that if two interrupts are triggered simultaneously, the hardware will select the interrupt with the higher priority. These timers are 32-bit registers that count downwards, and when they reach a value below zero, an interrupt is triggered. All cores have access to these timers, allowing multiple cores to be triggered by the same interrupt [4].

The main idea is to select one of these timers and trigger an interrupt each period to command each core to start the conversion. It is important to note that only the command is given to the cores during the interrupt; the actual conversion is not performed. This approach avoids lengthy interrupt handling routines, which would make the system less responsive to other external events (like for example other interrupts coming during that operation). Based always on the user manual, higher priority interrupt cannot be interrupted by lower ones [4].

After returning from the interrupt handler, the first core to respond to the command will lock the resource and start updating the temperature value.

3.5.3 Algorithm

Now that all the necessary elements for accomplishing this task have been explained, we can examine the developed final algorithm.

Figure 9 shows the implemented algorithm that is followed by all the four cores to ensure all the advantages and purposes previously discussed. The explanation is as follows.

All cores must receive a command to start a specific task. If no command is received, the core stays in an idle state. We'll start by analyzing the left branch of the graph. When the cores receive the command for the periodic read, they attempt to lock the resource (in this case, the ADC) by writing in a specific nibble of the RESOURCE_SYNC register. They then check if they successfully acquired the resource or if another core did. The successful core powers on the ADC, starts the conversion, and waits for the end-of-conversion signal. After the output of the ADC is valid, the core reads the temperature and puts it in a shared memory region where all the other cores can recover it.

The right side of the graph shows what happens when the core effectively needs the temperature. First of all it checks if the last update of the temperature happened within a timestamp in which the value can still be considered valid. If it did, nothing further is done, and the temperature value is simply read from the memory.

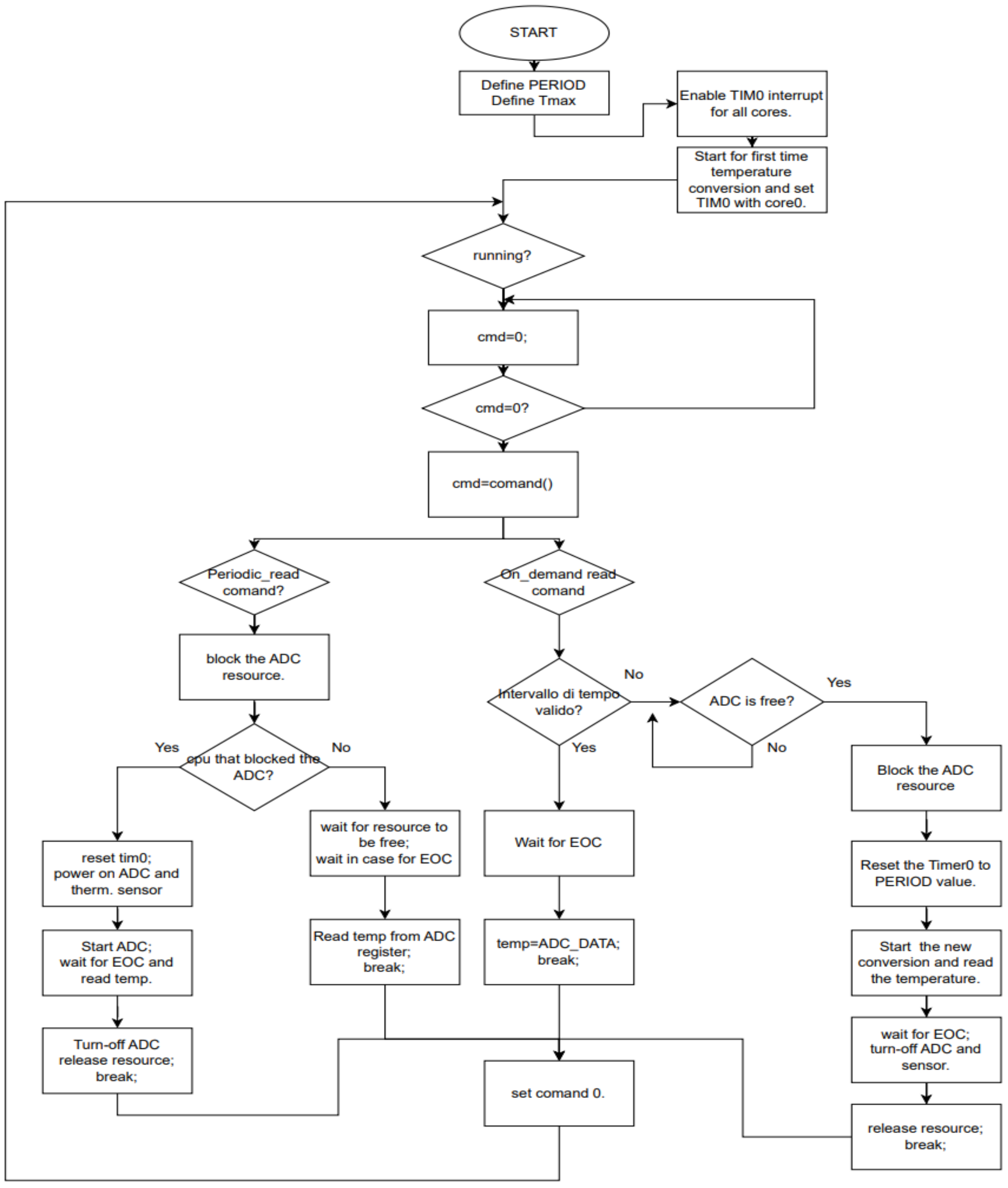

START
Define PERIOD
Define Tmax
Enable TIM0 interrupt for all cores.
Start for first time temperature conversion and set TIM0 with core0.
running?
cmd=0;
cmd=0?
cmd=comand()
Periodic_read comand?
On_demand read comand
block the ADC resource.
cpu that blocked the ADC?
Yes
No
reset tim0; power on ADC and therm. sensor
Start ADC; wait for EOC and read temp.
Turn-off ADC release resource; break;
wait for resource to be free; wait in case for EOC
Read temp from ADC register; break;
Intervallo di tempo valido?
No
Yes
Wait for EOC
temp=ADC_DATA; break;
ADC is free?
Yes
No
Block the ADC resource
Reset the Timer0 to PERIOD value.
Start the new conversion and read the temperature.
wait for EOC; turn-off ADC and sensor.
release resource; break;
set comand 0.

Figure 9: flow flow diagram of temperature reading algorithm**.**

On the other case, a new conversion is required. The core checks if the ADC resource is free (another core might be using the ADC for the same or a different purpose).. If the resource is free, the core locks it, verifies that it has successfully acquired the resource, and proceeds with the usual routine to start the conversion. Ultimately, the core releases the resource, resets the timer interrupt for the periodic request to restart counting from this moment (since the temperature value has been just updated) and returns to the idle state until a new command is received. If the ADC is not available, the core simply waits in a loop until the resource is released.

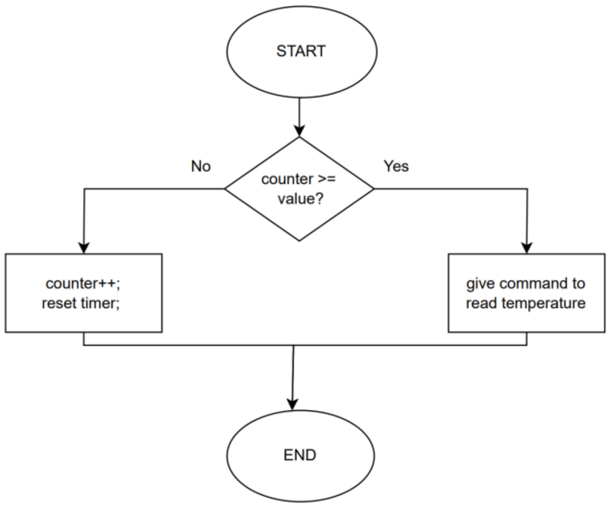


Figure 10: flow chart of timer interrupt handler.

Figure 10 shows the algorithm followed by the handler routine of the timer interrupt that gives the command for the periodic read. All the cores are sensitive to the same timer interrupt, so this means that the command arrives in the same instant to all the cores.

A consideration must be made here. With a 32-bit counter, the maximum number of counts is 0xFFFFFFFF. Depending on the microcontroller's running frequency, this represents the maximum countable time interval for a single timer cycle. On the considered board, the microcontroller operates at a frequency of 437.5 MHz, meaning that a timer cycle represents 9 seconds. To handle longer intervals, multiple timer cycles must occur. For example, to perform the periodic request every 30 seconds, three timer cycles are required. We need a method to calculate the number of these cycles and issue the related command only after the required number of cycles has occurred.

A solution can be the use of a counter variable that each new cycle is increased. Each interrupt handler checks if the necessary number of cycles has occurred and in case gives the command.

3.6 Encountered Problems

As previously mentioned, the approach where all cores follow the same instructions and autonomously handle the synchronization of resources presents many advantages. However, there are also some criticalities to consider.

For example, when the same interrupts arrive simultaneously at all different cores to give the same command, it might happen that a core is already handling a higher-priority interrupt. As a result, the new interrupt might be delayed, causing the command to arrive at that specific core later than at the other cores. During this delay, another core might have already locked the resource, completed the task, and released the resource. Consequently, the delayed core would find the resource free and start performing the same operation (in this case, starting the ADC conversion) even if it is not needed.

Fortunately, in our practical scenario of interest, this is not a significant problem. The worst-case scenario is that an additional reading of the thermal sensor is performed, which does not compromise the purpose of the task.

However, in more critical applications where timing and resource management are crucial, this delay could lead to inefficiencies and potential errors. For instance, in real-time systems where precise timing is essential, such delays could disrupt the synchronization of tasks and lead to inconsistent states. In these cases, this type of approach could not be the best or a more advanced interrupt handling system might be required. In our case, even though it is not necessary, a potential solution could be to use a very high-priority timer interrupt to issue the periodic request. However, this would mean removing this possibility for other tasks across all. This trade-off must be carefully considered to ensure that the system remains efficient and reliable while meeting the specific requirements of the application.

3.7 Final considerations

Handling resource sharing in a multi-core environment can present additional challenges and criticalities. As seen with this specific implementation, some issues may arise, especially if a more general algorithmic approach is followed to improve performance, work load balance, and efficiency.

In this specific task, synchronizing commands given to different cores can lead to unwanted behaviors. In the application of our interest, we saw this is not a big problem. However, this could be a more serious issue for tasks where very high and unpredictable behaviors cannot be accepted.

In such cases, a more suitable approach would be to assign a specific core the task of recovering a resource needed by all the cores and sharing it through the shared memory region. Although this approach may not provide all the listed advantages deriving from the first approach and introduces overhead for the specific core, it offers a more robust solution for different types of applications.

By adopting this method, the system can ensure that critical resources are managed more reliably, albeit at the cost of some performance benefits. This trade-off is often necessary in environments where predictability and stability are paramount.

4. Scheduler Development for STRED_L architecture

The second part of this thesis focuses on designing a scheduler for the STRED_L microprocessor in C language, enabling it to handle more tasks simultaneously. The main idea is to start with four tasks to replicate the multicore configuration, which is designed to work with four different channels.

In single-core mode, the processor can operate at a frequency that is four times higher than in multi-core mode. The micro system is designed so that, when multiple cores are used simultaneously, each core is responsible for sampling only one-fourth of the clock signal ramps. This means that each ramp of the clock signal is sequentially sampled by the next core, and so on. This approach helps to distribute the workload among the cores, thereby alleviating some of the limitations related to the total concurrency of the limited resources [4].

A program and a code memory RAM are present. In multicore configuration the data RAM is divided into 4 blocks and each of them is assigned to a specific CPU, making that area accessible only by the assigned CPU. Also, the memory map of the registers is reconfigured so that the different CPUs can use the same address, but to point at different physical locations in RAM. This allows for using the same code for all the cores, without having the need to calculate the offset for the different data slices.

In single core configuration the memory map becomes unique so that the CPU can access all regions of the RAM. The code memory RAM is shared among all the CPUs: the first 64 kB are read-only slots, while the last 64 kB can be accessed in read or write mode by all cores and can be used as a shared memory region [4].

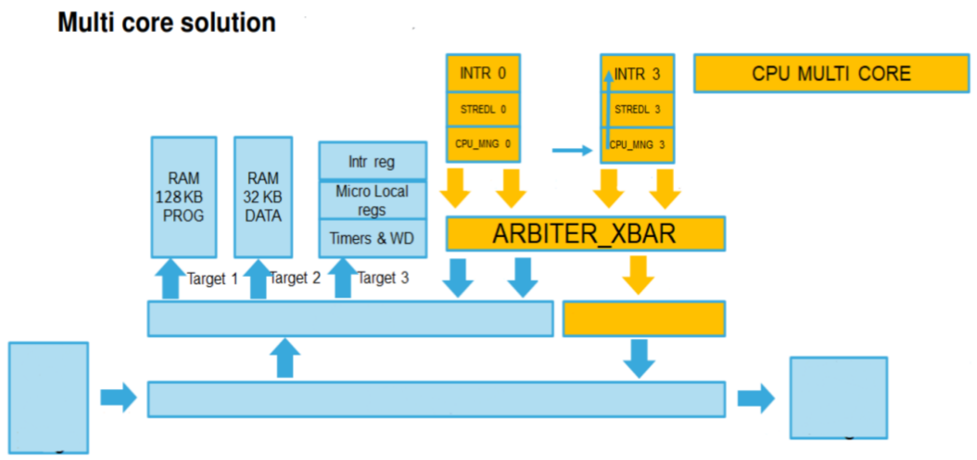


Figure 11:block diagram of the microcontroller system in quad core configuration [4].

From **Figure 11**, it can be seen that to enable the four cores to interface with all the microcontroller peripherals, additional control logic is required. This includes mechanisms to select between single-core and multi-core configurations, as well as control logic for managing access from different cores on the bus, including arbiters and selectors. Additionally, specific registers are reserved for multi-core mode [4].

The primary necessity for experimenting with a new scheduler implementation is based on the following two points:

- There is currently no scheduler available for the STRED-L architecture, unlike other architectures that already have dedicated schedulers.
- Due to the very limited resources in terms of RAM and computational power till now shown, a dedicated and optimized solution is necessary.

Our objective is to develop a scheduler for the specified core architecture that can efficiently manage context switches between various tasks. Beyond simply implementing a functional scheduler for this system, we aim to analyze its performance in a single-core setup and compare it with the quad-core configuration.

Theoretically, the multi-core setup should deliver superior performance. Despite operating at only a quarter of the maximum frequency, it can execute four distinct tasks concurrently, eliminating the need for context switching and its associated overhead. Context switching, in fact will incur a cost due to the necessary operations required to ensure the correct execution of tasks [8]. Our goal is to quantify this performance disparity and evaluate the true benefits of the multi-core configuration in terms of efficiency. Additionally, we aim to assess whether a single-core solution equipped with a scheduler could also be a viable alternative.

The presentation will begin with a brief introduction to real-time schedulers and an overview of the various algorithms they employ. Following this, there will be a general description of the STRED_L architecture, which is in particular essential for understanding the application of the context switch operation.

4.1 Brief introduction to real time schedulers

Real-time systems are those systems in which the correctness of the system depends not only on the correct execution of the task but also on its execution within a specific deadline [7]. A real-time system is dynamic and responsive to external or internal inputs. These systems must handle events in a timely manner to ensure predictable behavior and meet stringent timing constraints [11].

All these characteristics of a real-time system are enabled by the scheduler, the system component responsible for managing the execution of tasks or processes in a computing environment. It decides the order and timing of task execution, ensuring efficient utilization of system resources. More specifically, a scheduler assigns the CPU to specific tasks based on priority or other criteria, handles task switching while ensuring data integrity, and in real-time systems, manages task execution timing to ensure that tasks meet their deadlines [7].

4.1.1 Scheduling algorithms

A scheduler can implement various algorithms to manage task execution order, balancing factors such as responsiveness, efficiency, fairness, and real-time constraints. By selecting the appropriate scheduling algorithm, a system can optimize performance based on its specific needs [6].

For instance, some algorithms prioritize tasks to ensure quick response times, making them suitable for interactive systems, while others focus on maximizing overall system efficiency by minimizing idle time. Fairness is another critical aspect, where algorithms ensure that all tasks receive a fair share of CPU time, preventing any single task from monopolizing resources [6].

In this application, the primary requirements are to enhance system responsiveness and ensure fairness among tasks, replicating the behavior of the quad-core configuration where each core is associated with a specific data slice. To achieve this, a Round-Robin algorithm will be implemented. This simple algorithm allows each task to run for a specific time slice before switching to the next available task, ensuring that every task receives an equal amount of CPU time.

The implementation will progress from a basic solution to a more complex and comprehensive final version. This iterative approach allows for incremental development and testing, ensuring that each stage is functional before proceeding to the next.

4.2 Non preemptive scheduler

The first step for creating the scheduler is to start with a non-preemptive algorithm. This means that the tasks will be executed in sequence and no context switch will happen between them. Every task is interrupted by reaching the end of its execution, and by leaving control back to the scheduler which will check for the next available task in the ready queue and execute it [5]. This is a much simpler version of the scheduler and of course it has many limitations. Some of these are:

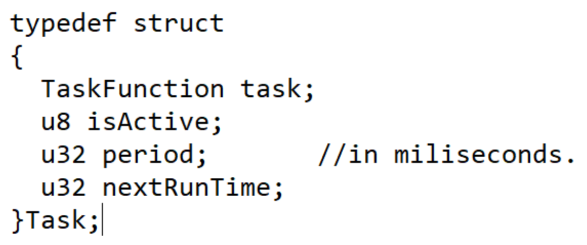
- **Poor responsiveness**: since tasks cannot be interrupted once they start running quick response time cannot be guaranteed.
- **Inefficient CPU utilization**: if a running task is blocked or waiting for an I/O operation to complete, the CPU remains idle until the task finishes, while it would be more efficient to start executing another task in the meanwhile. Idle states are not exploited efficiently.
- **Less capable of handling real time constraints**: the impossibility to preempt tasks makes it more difficult to ensure that all tasks meet their deadlines[5].

Due to these limitations, this type of scheduler is suitable only for specific environments where real-time constraints and high responsiveness are not critical requirements. In our case, it may be useful in industrial applications and serves as an initial starting point for designing of a more complex scheduler.

4.2.1 Task Definition

First thing is define the task structure: every task will needs an ID number to be identified unequivocally, an active flag, that tells if the task is active or not, a pointer to the task function that the task will execute and a period attribute for cyclic tasks: by assigning an integer N higher than zero to this field, it means the task will be executed every N milliseconds. A maximum number of tasks, which can be defined by the user, defines the size of the tasks array. The position of the tasks in the task array also represents the ID number of task. The task array is initialized in the main, where the user can specify all the previous attributes for each task.

Once all necessary tasks have been initialized the scheduler can start its job.



```
typedef struct
{
  TaskFunction task;
  u8 isActive;
  u32 period;       //in miliseconds.
  u32 nextRunTime;
}Task;
```

Figure 12: task structure definition for non-preemptive scheduler.

4.2.2 Scheduler Function

The scheduler itself is defined as a non-returning function since it implements an infinite while loop. Inside this loop, a counter starting from zero is incremented until it reaches the maximum number of tasks in the task array. For each cycle, the corresponding task is checked to see if it is active. If the task is not active, it will not be scheduled, and the loop will proceed to the next cycle.

If the task is active, the scheduler checks if it is a periodic task. If it is not periodic, the scheduler simply executes the task. Once the task execution finishes (when the task returns), it is set as inactive and will not be scheduled again until it is reactivated.

For periodic tasks (those with a non-zero periodic field), the scheduler checks if the necessary period since the previous execution has passed. This is achieved by using a timer interrupt that updates a counter variable every millisecond. The period of the task is expressed in milliseconds (this can be easily adjusted as needed). Every time the task is executed, its *nextRunTime* field is updated by adding the task's period to the timer interrupt's counter variable. The task will not be executed again until the counter variable reaches or exceeds the *nextRunTime* value.

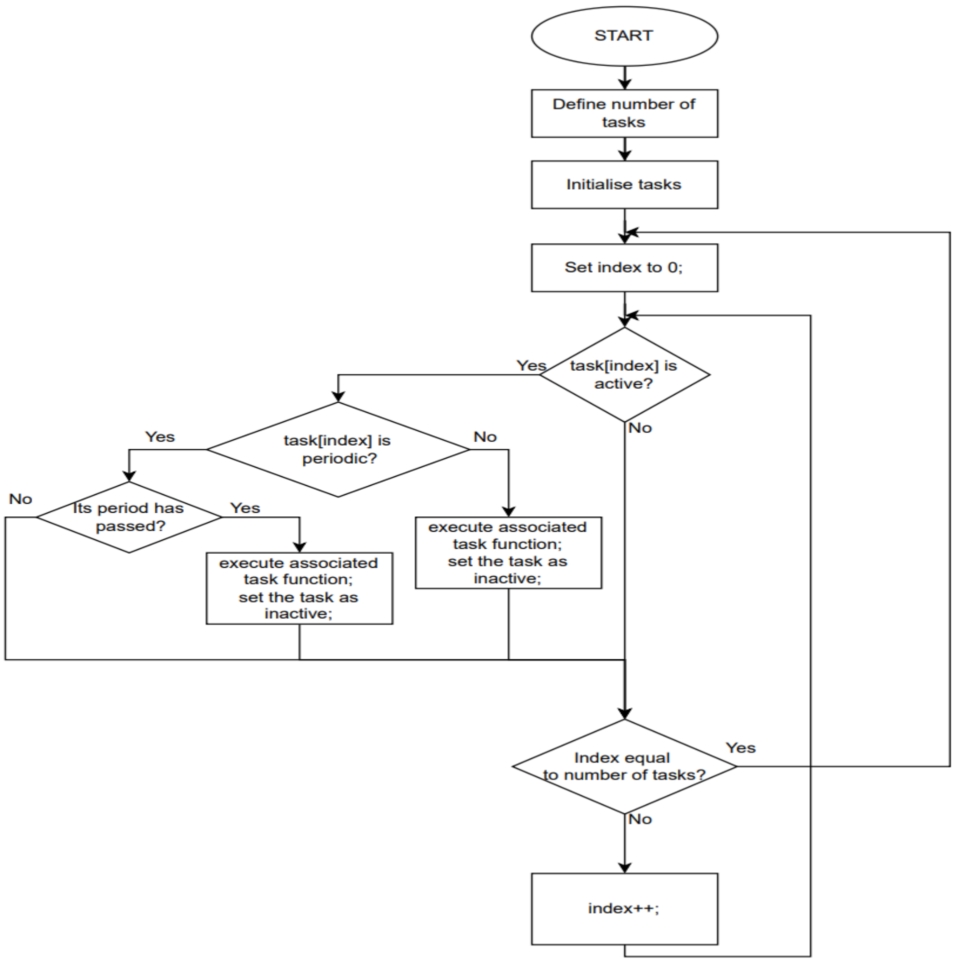


Figure 13: flow chart of non-preemptive scheduler algorithm.

While this mechanism allows for periodic tasks, it is important to note that it is not very precise. The execution of a running task cannot be preempted, so even if the period for a periodic task has passed, it must wait for the currently running task to finish execution before it can be scheduled again.

4.3. Round Robin Scheduler

Now we can start with the implementation of the round-robin context switch scheduler, that will have no more the limitations of the previous solution but will allow for higher responsiveness, a better real time constraints handling and a more efficient CPU utilization.

The key point in the development of this part, and probably the most challenging, is to correctly perform context switching between tasks. This operation refers to the switching of the CPU from one process (or task in this case) to another [8]. This involves saving the state of the currently running task and accurately restoring the state of the next task that will access the CPU.

Saving the state of a task means capturing its program counter, stack pointer, link register, and all related data used in its intermediate operations. This information must be stored in a known and specific region of the RAM memory [7].

After restoring the task, it should be able to resume execution from the exact instruction where it was interrupted, with the same data as its previous state. All these data are stored in the register files of the CPU architecture during task execution. To correctly access save and restore them when needed, a deep understanding of the core architecture is first required.

4.3.1 STRED_L architecture

For reservation rights its properties and characteristics cannot be fully disclosed; Only the essential aspects necessary to describe this work will be presented.

STRED-L is a sophisticated 32-bit RISC architecture, capable of handling up to 256 bits per operation. As reported in its manual It is intended to be integrated into ST's proprietary System-on-Chip (SoC), whether as a standalone core or within a multi-processor unit, to support a diverse range of computing-intensive application [2].

It includes sixteen 32-bit standard registers, a set of special registers to handle interrupts and exceptions, a program counter register and instruction result flags. Its Harvard topology allows for concurrent data memory accesses and instruction fetches [2].

An important part for being able to perform context switch, as said before, is to analyze the Register File of the micro architecture. This one is composed of 16 registers going from R[15] down to R[0], which are accessed by all the standard instructions and can maintain an 8/16/20 or 32 bit data type[2].

Three of these registers serve special purposes: the link register, the stack pointer, and the program status register. These registers are crucial for the effective implementation of context switching.

- **Link Register**: During a function or subroutine call, the current value of the program counter is saved in the link register. This value is then used by the return instruction at the end of the subroutine to jump back to the original routine.
- **Stack Pointer Register**: it points to the top of the stack memory.
- **Program status register**: this register holds the program status word and the saved program counter during interrupt routines[2].

In this architecture the program counter is not directly readable, but it can be accessed and modified through standard jump instructions and the RFE (return from exception) instruction.[2]

Exceptions handling

An exception is any event that disrupts the normal flow of program execution [2]. In the STRED-L core, two types of exceptions are defined: interrupts and traps. For our purposes, interrupts are the most relevant, as they will be used to regulate the timing of context switching. This is crucial for implementing the Round Robin algorithm discussed earlier.

The exception point is the specific moment during program execution when an exception occurs. All instructions initiated before this point will complete and update the architectural state. However, no subsequent instructions will update the architectural state until the exception is handled.

During an exception, the program execution branches to the required handler, similar to a function subroutine, but some additional operations need to be considered. Most importantly, the Program Counter (PC) is stored in the Save State register, and then the PC is updated to point to the exception handler routine.

The RFE instruction is used to recommence the execution at the exception point. It takes the content of the saved state register and writes it back on the Program Status (PS) and PC registers respectively. It is important to notice that the values stored in PS register can be accessed and changed during the interrupt handler execution [2] and this will be very important for the implementation of context switch.

#index	Priority
1:7	Low
8:15	Middle low
16:23	Middle high
24:31	High

Figure 14: interrupt priority levels [2].

The interrupt controller handles from to 31 interrupt sources. Each interrupt has its own priority level, and it depends on the index number of the interrupt.

List of some instructions

Some of the necessary assembly instructions (listed in the manual of the core) for implementing a context switch are described here. As mentioned, we need to save the state of the execution in memory and be able to restore it at a different time. Therefore, instructions that allow access to memory, such as load and store instructions, will definitely be required.

- The store value to memory operation is given by the assembly instruction:

  ```
  stw    [imm], %ry
  stw    [%rx], %ry
  ```

  The instruction can be used in two different semantics: the first one allows for storing the content (in this case a 32-bit data) of the register **ry** at the address targeted by the immediate value **imm**, while in the second case the memory address is the one targeted by the content of the register **rx** [2].

- The load from memory operation is given by the assembly instruction:

  ```
  ldw    %ry, [imm]
  ldw    %ry, [%rx]
  ```

  This allows to load the content stored in the targeted address memory into the **ry** register. Also in this case the memory address can be targeted with an immediate value or using the content of a register.

Other important instructions to consider for our purpose are the jump instructions. These allow the program to jump to a different branch of execution and store the return address in the link register or the PS register [2].

4.3.2 Context Switch

We now possess all the necessary elements to effectively implement context switching between different tasks. To ensure data integrity, it is essential to save all sixteen registers in the register file, as it is unpredictable which registers the compiler will utilize for executing the task's instructions. The most straightforward approach is to store their values sequentially in the task's stack memory and then, in the reverse order, load them from the stack back into the register file by directly using the *stack pointer* register.

The save and restore context operations must be atomic, meaning they must be executed as a single, indivisible action to ensure no external instructions can interfere and compromise the integrity of the restore process. This precaution is necessary because any extraneous instructions could alter critical values in the register file, thereby compromising the data integrity required for the current task's execution. Therefore, a method to ensure atomicity is required.

Once the save context operation is completed, new external instructions can then be executed. At this point, the scheduler can assume control of the CPU. Depending on its functionalities, it can select the next task to run, remove inactive tasks from the schedule, manage task priorities, and perform other necessary scheduling operations, without the risk of compromising the status of the just preempted task.

Once the next task has been selected, the scheduler must restore its context from the designated memory region (task's stack memory) and load it back into the register file. As mentioned, atomicity must be assured during this operation as well. After the context has been successfully restored, it is necessary to update the Program Counter (PC) to point to the next instruction of the restored task to restart its execution.

In summary, the final scheduling algorithm implementing context switch will be composed of three main ordered sections:

1. Store context of current running task.
2. Task scheduling algorithm.
3. Restore context of next scheduled task.

In a Round Robin scheduling algorithm, context switches must occur at regular, predefined intervals. In the STRED_L architecture the only effective way to achieve this is by using a timer interrupt to mark each time slice during which a task takes control of the CPU. For example, if each time slice is 1ms, a timer interrupt will occur every 1ms. When the timer interrupt occurs, the execution jumps to the interrupt service routine, which initiates the context switch process and gives the command to perform the three sections listed above.

There are several ways to handle this mechanism. The first possibility, for example, is to perform the entire activity within the Interrupt Service Routine (ISR). This solution would have the structure shown in **Figure 15**.

```
void scheduler_ISR()
{
        store context of current task;
        task scheduling algorithm;
        restore next task;
}
```

Figure 15: scheduler ISR pseudocode for context switch.

The advantage of this mechanism is that by choosing the highest priority timer interrupt as the ISR, all these operations can occur without external interruptions. No lower priority interrupt could preempt this operation, thereby ensuring atomicity as required. However, there are also some minor drawbacks to consider. Although this is a solid operation, this method might result in a relatively lengthy Interrupt Service Routine (ISR). The *task management* section could vary in complexity based on the functionalities we wish to add to the scheduler. Keeping this section within an interrupt handler routine might not be the best choice, as it could introduce significant overhead and reduce system responsiveness to other interrupts. Additionally, this section does not require atomicity, making its inclusion in the ISR unnecessary.

A second option is to use the ISR solely to jump to the scheduler routine, where the three necessary sections will be performed. This means that in the IRQ, the saved status register (let's hypothesize it is *r13*) must be filled with the address of the entry point of the scheduler function. Then, the *RFE* (Return from Exception) instruction is executed. This instruction will pop the content of *r13* and load it into the Program Counter register. Once in the scheduler routine, all three necessary sections for the context switch can be performed.

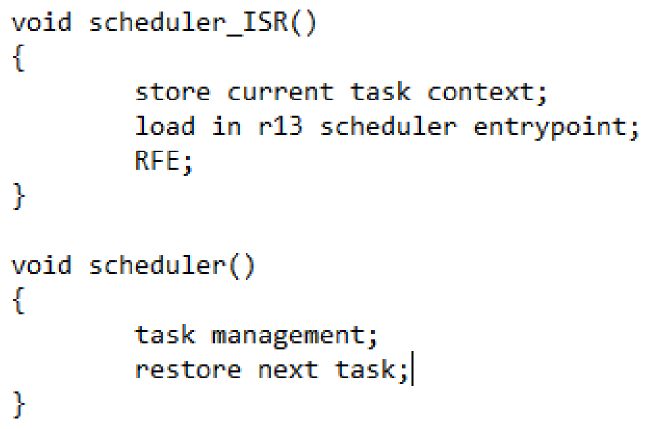
```
void scheduler_ISR()
{
        load scheduler entrypoint in r13;
        RFE;
}

void scheduler()
{
        store context of current task;
        task scheduling algorithm;
        restore next task;
}
```

Figure 16: pseudocode for scheduler and ISR routine.

This solution allows for a much shorter and faster ISR. The problem in this case is that atomicity of the context switch activity is not guaranteed since other interrupts can occur during the scheduler routine potentially compromising the data integrity of the context switch. To avoid this, one possibility is to globally disable interrupts during these two procedures.

The chosen solution is a modified version of the last one presented. **Figure 17** shows the final implemented algorithm. The context of the currently running task is stored directly in the timer interrupt handler, as this already guarantees atomicity for the operation.



```
void scheduler_ISR()
{
        store current task context;
        load in r13 scheduler entrypoint;
        RFE;
}

void scheduler()
{
        task management;
        restore next task;
}
```

Figure 17: the ISR only handles the store context than jumps to scheduler routine.

After that, the ISR jumps to the scheduler routine, where all the necessary task management activities are handled and the context of the next selected task is restored. During this last operation interrupts must be disabled.

4.3.3 Stack Memory Handling

Stack memory is a specific region of a device's memory that operates in a last-in, first-out (LIFO) manner, meaning the most recently added item is the first to be removed. It is primarily used for managing function calls, local variables, and control flow within a program.

Memory allocation and deallocation on the stack are generally handled automatically by the compiler, as also in the STRED architecture. When a function is called, its local variables are pushed onto the stack, and when the function returns, its local variables are popped off the stack. A fundamental component of this mechanism is the Stack Pointer, a variable that points to the top of the stack and is automatically updated as items are pushed onto or popped off the stack. In our architecture, stack memory grows downwards, which means that the first element pushed onto the stack will be at the highest address.

For the scheduler implementation, to each task is assigned a dedicated memory region to be used as stack memory. This is achieved by initializing the stack pointer register during the first execution of each task to point to the top of its assigned stack memory. Since the flow of execution will be interrupted and replaced with other tasks, it is also necessary to have a specific address to store the content of the stack pointer register itself, so that it can be recovered during the restoring phase.

If the stack pointer register were saved on the stack, it would be impossible to know which address to access in memory during the restoring context, as the top of the stack address will be lost during the execution of other tasks. Therefore, at the end of the store context for each task, the stack pointer register is saved in a known, dedicated memory address. This ensures that during restoration, the stack pointer can be easily retrieved and used to access the stack of the newly scheduled task.

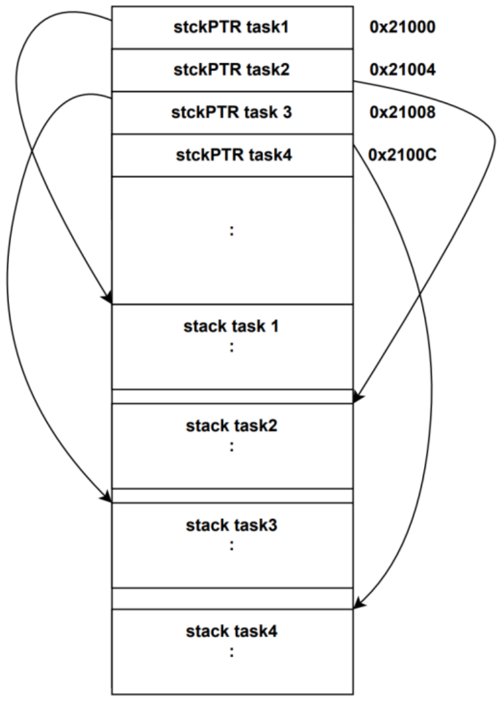


Figure 18: Stack handling representation.

Let's suppose *task1* is currently running and a context switch occurs. The context of task1 needs to be saved, so all the registers are pushed onto the stack using the stack pointer register. Once all the registers have been stored, the stack pointer register itself must be saved to a known address for future restoration. In this case, the stack pointer register will be saved to the memory address designated as *stackPTR_task1*.

When *task1* is rescheduled, we can access the address *0x21000* to retrieve and load its content into the stack pointer register. This will ensure that the stack pointer points to the top of *stack_task1*, allowing the scheduler to load back the register values and resume execution.

Figure 19 summarizes both the store and restore context operations of a task.

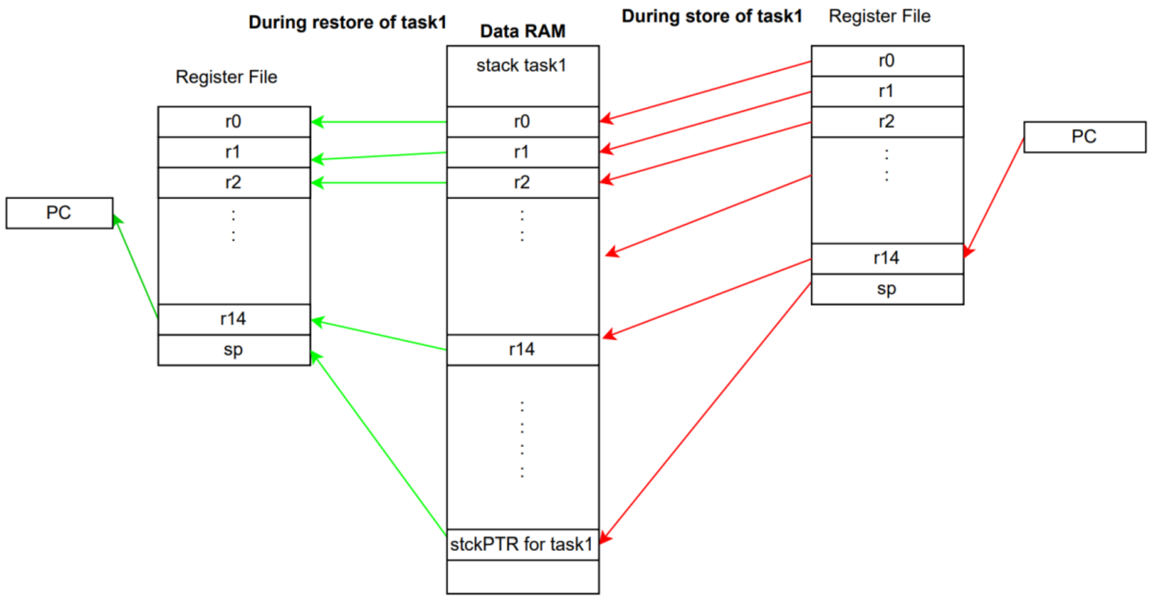


Figure 19: During the context store phase, registers are saved onto the task's stack. In the context restore phase, they are retrieved from the stack in the reverse order, as the stack operates in a Last-In, First-Out (LIFO) manner. A hardware mechanism allows to save the PC in one of the registers, so that it can be saved and restored.

Now the three sections composing the context switch operation will be showed in detail.

4.3.4 Store Context

The storing of the running context doesn't require external data. By just using the stack pointer register the content of all the registers can be pushed into the stack. The following image shows how the algorithm would look.

```
store_context()
{
        stw[%sp], %r0;
        subw %sp, #4;
        stw[%sp], %r1;
        subw %sp, #4;
        stw[%sp], %r2;
        subw %sp, #4;
                :
                :
        stw[%sp], %r14;
        subw %sp, #4;

        stw [stckPtr_addrss], %sp;
}
```

Figure 20: store context algorithm.

From the reported image (**Figure 20**), we can see that the stack pointer register (*sp*) is used to point to the stack memory of the currently running task for storing all the standard registers, starting from register *r0* until *r14*. After a **stw** instruction, the stack pointer register (*sp*) is updated by subtracting the immediate value 4. This is because the stack grows downwards, so to point to the next free address after a push, it has to be decreased. The value 4 stands for the number of bytes in each register. After each push operation, the *sp* value is reduced by 4 bytes to point to the next location.

The final operation to be done is the storing of the *sp* register itself. Based on the current running task, this is stored in a specific memory address to be easily retrieved during the restoring session.

After this last operation, all the necessary data to restart the execution of this task the next time it will be scheduled are stored in memory. At this point, external instructions cannot affect the integrity of this operation.

4.3.5 Task scheduling

The scheduling of the next task can be managed based on the functionalities of the scheduler. The algorithm for selecting the next task to run can vary in complexity depending on these functionalities. Initially, we aim to keep it as simple as possible for two main reasons:

1. **Testing purposes:** A simple scheduling algorithm will make it easier to test the functioning of the context switch operation.
2. **Overhead analysis:** the simplest task management will allow to calculate the lowest overhead value introduced by the scheduler activity.

Considering always to have four tasks, the simplest implementation for Round-Robin algorithm is to schedule them sequentially: starting with *task1*, followed by *task2*, *task3*, and *task4*, and then repeating the cycle.

(In the last chapters more complex algorithms will be introduced able to manage task's priority).

4.3.6 Restore Context

After selecting the next task to run, the scheduler must access the top of its stack memory and restore the elements by loading them into the register file. This operation must be performed in the reverse order of the store context process. For instance, if the store context operation began by saving register *r0*, followed by *r1*, *r2*, and so on until *r15*, then the restore context operation should start by loading register *r15*, followed by *r14*, *r13*, and so forth, ending with r0. This ensures that all the registers are restored with the correct values and that the stack pointer register will point to the same exact position as it did before the context switch.

```
restore_context()
{
        ldw %sp, [stckPtr_address];

        ldw %r14,[%sp];
        addw %sp, #4;
        ldw %r13,[%sp];
        addw %sp, #4;
        ldw %r12,[%sp];
        addw %sp, #4;
                :
                :
        ldw %r0,[%sp];
        addw %sp, #4;
}
```

Figure 21: pseudocode for restore context algorithm.

The first step is to update the stack pointer register with the address of the top of the stack for the selected task, which is stored in a known location. This address is loaded into the stack pointer register to access the stack memory of the selected task. As each register is restored, the stack pointer register is incremented to point to the next element in the stack, ensuring proper sequential access as elements are popped off. After this no other instructions that could modifie the values of the register should be effectuated since it could affet the data integrity for the new running task.

4.3.7 Final solution

By putting together these three sections the final Round-Robin scheduler for the STRED-L architecture can be completed. As in the first non-preemptive scheduler a task structure is defined, composed of a task identifier (task ID), a stack memory array and a task function.

```
struct
{
        int taskID;
        void* TaskFunction;
        void *stckPTR;
        u32 stack[STACK_SIZE];
}Task;
```

Figure 22: Task structure definition.

The new defined *stack* field represents the assigned stack memory for each task. Its dimension can be directly set by the user. During task's execution the stack pointer register will point to this region in memory.

The task initialization is performed in the main program. As illustrated in the flow graph in **Figure 23**, the *initTask* function sets the *taskID* for all tasks and, based on the *taskID*, assigns each task a specific memory address to act as the stack pointer. At first the stack pointer will point to the first address of the stack and is immediately used to initialize the stack locations that will be loaded into the register file for the first execution of the task.

All stack locations that will be loaded into the register file for the first execution of the tasks are set to 0 (just for safety purposes), except for the one corresponding to *r13* (status register), which is overwritten with the entry point of the associated task function. This value will, in fact, be loaded into the Program Counter (PC) register [2], when the task will be scheduled for the first time.

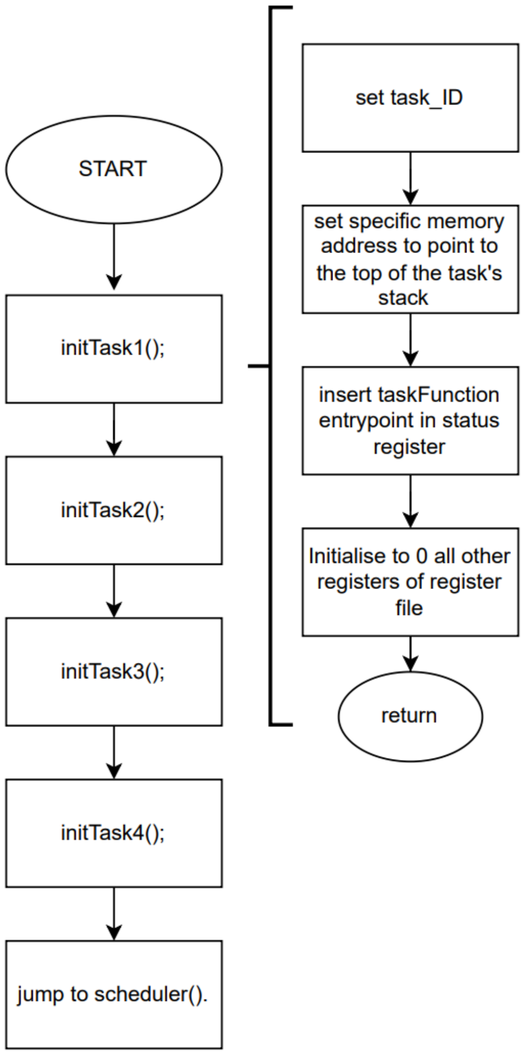

set task_ID
START
set specific memory address to point to the top of the task's stack
initTask1();
insert taskFunction entrypoint in status register
initTask2();
Initialise to 0 all other registers of register file
initTask3();
return
initTask4();
jump to scheduler().

Figure 23: flow chart of main and task initialization.

Before task execution, these elements will be popped off the stack and loaded into the register file. Once initialization is complete for each task, the *current_task* global variable is set to 1, ensuring that *task1* will be the first to be scheduled. The final step is to jump to the scheduler routine, which will schedule the first task. From this point onward, execution will never return to the main function again.

The first step here is to save all the registers onto the stack and store the content of the stack pointer register in the current task's memory location, as shown in previous chapters. The timer is then set to 0 to halt until the complete context switch is applied.

Now that all the data of *task1* have been saved, the *r13* register is overwritten with the address of the scheduler entry point since the old value has already been stored in memory. The interrupt bit is cleared, as required at the end of a handler routine. With the *RFE* (Return from Exception) instruction, the *r13* value is popped into the Program Counter (PC) register [2], causing the execution to jump to the scheduler function.

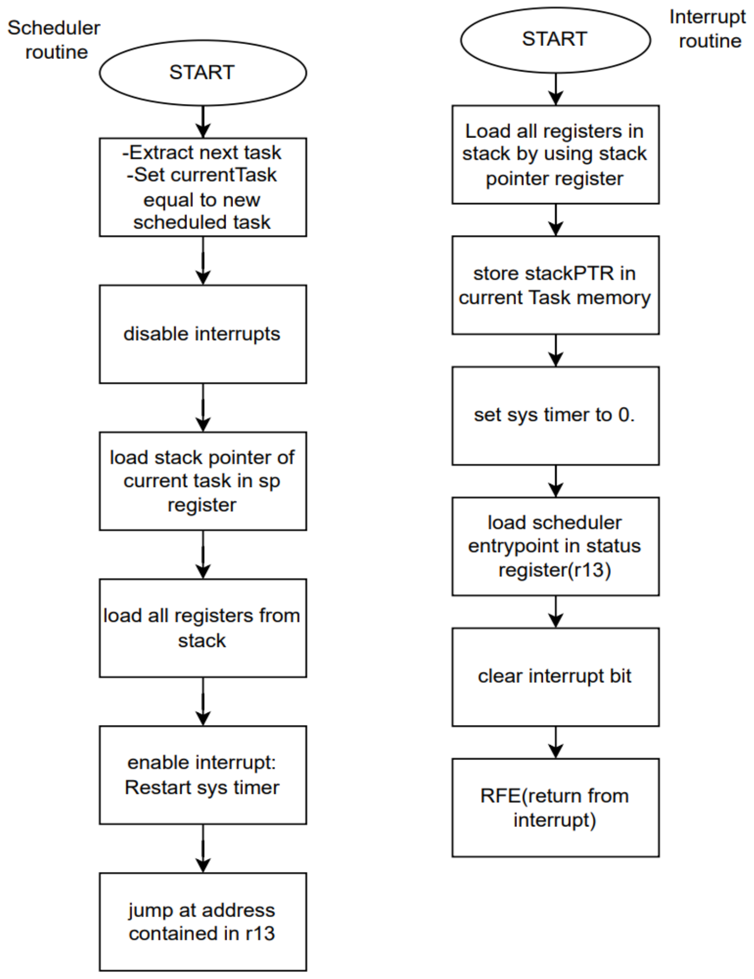

Scheduler routine
START
-Extract next task
-Set currentTask equal to new scheduled task
disable interrupts
load stack pointer of current task in sp register
load all registers from stack
enable interrupt: Restart sys timer
jump at address contained in r13
Interrupt routine
START
Load all registers in stack by using stack pointer register
store stackPTR in current Task memory
set sys timer to 0.
load scheduler entrypoint in status register(r13)
clear interrupt bit
RFE(return from interrupt)

Figure 24: flow chart of scheduler and sys_timer interrupt routine**.**

Once in the scheduler routine, the task to be run for the first time is extracted from the task array. The global variable *currentTask*(it indicates which task is currently being executed) is updated with the new *taskID*. Now, before starting the restore context section, interrupts are globally disabled making this operation atomic. Restore context is then applied as explained in the previous paragraph, and before jumping to the entry point of the first scheduled task, interrupts are enabled. Disabling and enabling the interrupts requires just to write in the dedicated interrupt register a '0' or a'1'.

Also, before taking the jump, *timer4* (the highest priority timer interrupt in the system [2]) is set with the necessary value that defines the established time slice. The new scheduled task will execute until the timer expires again and the interrupt occurs.

From here, the new task is selected (in this case, *task2*). The system timer is then restarted to begin counting the new time slice.

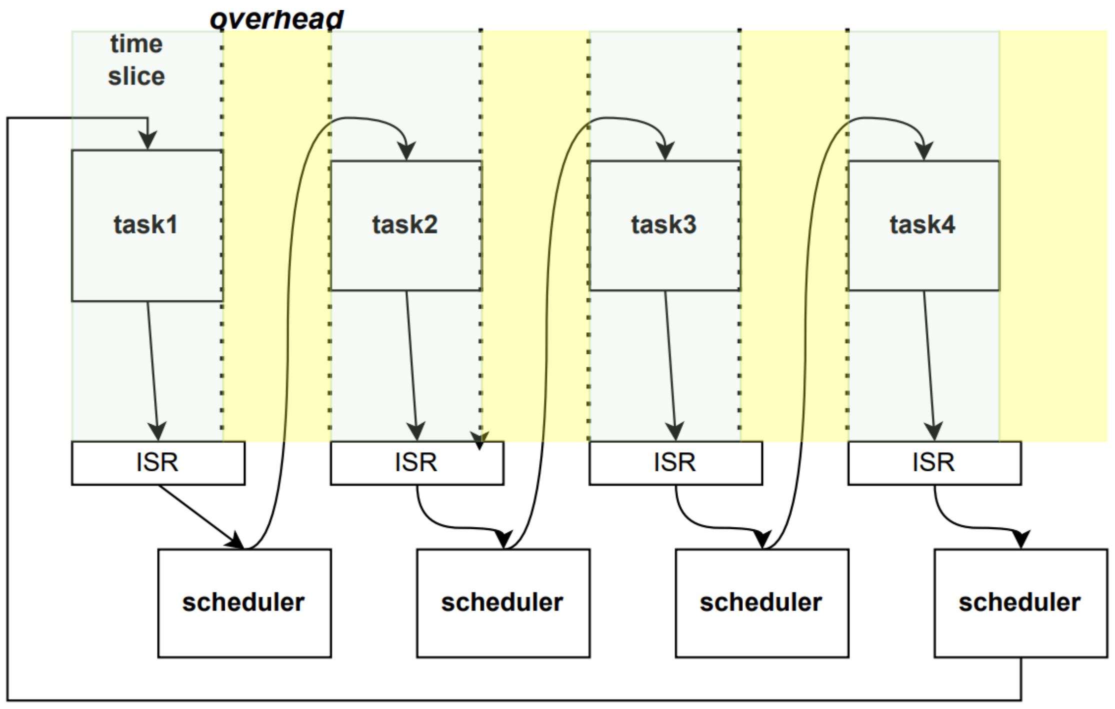


Figure 25: scheduling execution flow with four tasks.

4.3.8 Testing

The scheduler has been tested to ensure its robustness and reliability. Initially, loop test programs were employed to verify the basic functionality and performance of the scheduler under continuous and repetitive operations. These tests helped identify and resolve any issues related to task switching, timing, and data integrity.

Subsequently, the scheduler was subjected to more rigorous testing using industry applications. These applications were chosen to simulate real-world scenarios and workloads, ensuring that the scheduler can handle the complexities and demands of the industrial environments for which it is intended. The main test is the firmware of the receiver calibration. Each task has runned the calibration algorithm for a specific data slice context switching.

By combining both loop test programs and industrial-grade applications, we have ensured that the scheduler is capable of efficiently managing tasks in a wide range of scenarios, from simple repetitive tasks to complex and high-demand operations.

Timing performance comparation

A timing performance analysis is performed to quantify the scheduling overhead and then compare it with a quad-core configuration. This is done by running a timer and taking samples t_1 at the start of the process and t_2 at the end of it. Since the timer counts down, the final duration is obtained by:

$$T = t1 - t2$$

The first step is to calculate the overhead of the context switch. This is done by taking the first timestamp at the arrival of the interrupt and the last timestamp at the beginning of the next scheduled task. The test measurements have been portrayed in a system running at a frequency of 437.5 Mhz.

The context switch operation itself takes 142 cycles operations, which at the corresponding frequency means 324 ns of overhead.

The second timing performance test was conducted by running a series of four loops, each consisting of a defined number of cycles (1 million in this case), without interruptions (i.e., without context switching) to obtain a precise measurement of the program's duration. Subsequently, each task was executed by running a single loop. This means that the tasks now run a program that is four times shorter in duration, but with context switching between four tasks at each defined time slice. The duration of the loop in each task is measured and compared with the first case.

We expect the task to require an amount of time for loop execution that equals the time taken in the case without context switching to complete the four sequential loops, plus the introduced overhead.

As shown in Table 1, these measurements were conducted for different values of time slices to quantify the impact of context switching on performance.

Time slice [ms]	Overhead
0.01	3.385%
0.02	1.682%
0.04	0.830%
0.08	0.405%
0.1	0.328%
0.2	0.117%
0.3	0.074%
0.4	0.058%
0.5	0.047%
0.6	0.036%
0.8	0.029%
1	0.023%
2	0.012%
5	0.005%

Table 1: overhead percentage impact over different time slices values**.**

The measurements were started considering a minimum reasonable time slice of 0.01ms. In this case, we can see that the scheduler affects performance by an increase of 3.39%, which can be considered an optimal percentage for this time slice. By increasing the time slice, the overhead, as expected, decreases until it almost reaches 0%. Conversely, further reducing the time slice would increase the overhead, eventually reaching a point where the CPU spends more time context switching than actually executing the tasks.

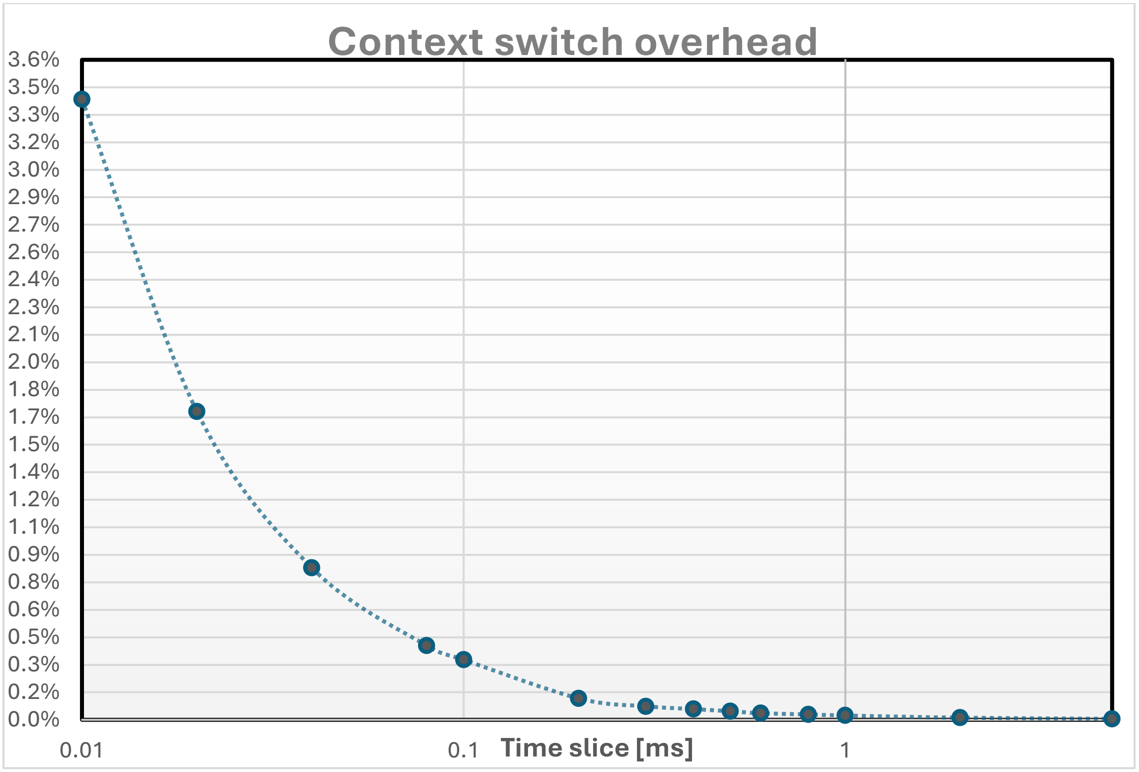


Figure 26: Analysis of overhead curve with varying time slices.

4.3.9 Final Considerations

From the analysis, we can see that a single-core scheduler might be a valid solution with respect to the quad-core configuration, considering that at this frequency the introduced overhead is very small and almost negligible, even for very small time slices.

However, we must also consider that the results obtained here represent the minimum introduced overhead. The scheduler is currently very simple and limited. It does not allow for activating or deactivating tasks, does not consider priority levels for tasks (which would make the system more dynamic and adaptive to a greater number of application scenarios), does not allow for pausing tasks or other useful functionalities.

All these additional functionalities would introduce complexity into the scheduling algorithm. Even though the store and restore sections would remain the same, the increased number of instructions required for the scheduling activity would lead to a higher overhead.

From the point of view of power consumption, an analysis has not been conducted. This is because, for the type of application towards which this system is designed, a difference in power consumption would not be appreciable. The entire SerDes system is not intended to be a low-power application. Therefore, any potential variations in power usage due the multi-core and single core environment are considered negligible and not critical to the overall system performance.

In the next chapters new features will be added to the scheduler making it more complex and dynamic.

4.4 Adding new functionalities.

The scheduler algorithm developed till now theoretically allows for the highest performance since the introduced overhead is minimal considering that task management is very simple. This solution might be enough for certain types of applications, but it’s still not very efficient for real time environments since it does not account for the varying importance or urgency of different tasks.

To address this limitation and make the scheduler more complete for future possible applications, priority levels are introduced within the Round Robin algorithm. This means that to each task can be assigned a priority attribute (within a certain range) that will allow it to receive higher attention from the scheduler. This enhancement aims to improve the overall efficiency and responsiveness of the system, particularly in environments where certain tasks are critical and require prompt execution.

Also, an important feature that is very important for making the system more complete is adding locking mechanisms. As presented in the previous chapters of this document, the multi-core configuration of the micro allows to synchronize the different cores during access to shared resources by accessing a specific register (SOURCE_SYNC_REG), properly designed in hardware to latch-in with the writing core. This mechanism cannot be exploited anymore in the single core configuration, so a new mechanism is necessary for allowing the different tasks to securely access shared resources.

In the next paragraph, these new added features will be presented starting from the new priority base round-robin algorithm.

4.4.1 Task's priority handling.

The implemented algorithms for taking into account also priorities is based on the fact that each defined time slice the scheduler finds the task with the highest priority and runs it. If more tasks have the same priority (the highest in task's active list), then context switch will happen only between these tasks. The tasks with a lower priority will not be scheduled, until its priority will be equal or higher than the other's tasks. **Figure 27** shows this concept with an example.

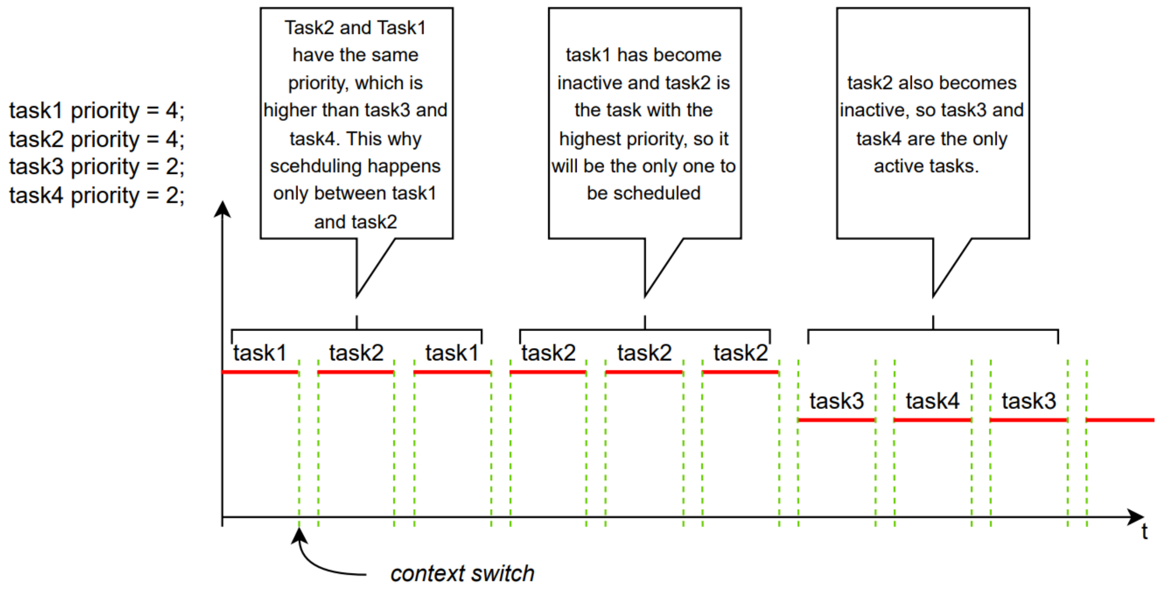

task1 priority = 4;
task2 priority = 4;
task3 priority = 2;
task4 priority = 2;
Task2 and Task1 have the same priority, which is higher than task3 and task4. This why scehduling happens only between task1 and task2
task1 has become inactive and task2 is the task with the highest priority, so it will be the only one to be scheduled
task2 also becomes inactive, so task3 and task4 are the only active tasks.
task1
task2
task1
task2
task2
task2
task3
task4
task3
t
context switch

Figure 27: scheduling example with priority.

To implement this mechanism, a priority queue is created, consisting of a head pointer and a tail pointer. Each task is assigned to a queue representing its level of priority, and these queues are part of a queue array. The position of the queue in the array represents the priority level of the queue. During initialization, all tasks are assigned to their respective queues.

Task insertion in the priority queue

A queue is a data structure that follows the First-In-First-Out (FIFO) principle, which means that the first element added to the queue will be the first one to be removed. Since we want to preserve the order in which tasks are inserted the queue structure results in the most suitable one. It is composed of:

- **Head pointer:** Points to the first element inserted in the queue, which is the next element to be extracted from the queue.
- **Tail pointer:** Points to the last element inserted in the queue.

The task structure becomes more complex and with more fields. Now besides the one described in the previous paragraphs, each task structure will have a *priority* and a pointer to the next task in the queue. This one is needed for pointing to the next task in the same priority queue and maintain the order of insertion in the queue. See **Figure 28**.

At initialization, the priority queue is empty, and the head and tail fields are NULL pointers. The first task inserted into the queue will be pointed to by both the head and the tail. Being the first task inserted, its *nextTask* field will be set to NULL. The next task inserted into the queue will be pointed to by the current task tail and become then the new tail of the queue.

```
struct
{
        int taskID;
        bool isActive;
        int priority;
        void* nextTask;
        void* stckPTR;
        u32 stack[STACK_SIZE];
}Task;
```

Figure 28: Task structure for handling priorities.

Task Extraction Algorithm

To schedule the next task at the end of the time slice, the scheduler, after saving the context (as described in the previous chapters), moves the current task to the end of its priority queue. The next task pointed to by the current task becomes the new head of the priority queue.

After this operation, the array of priority queues is scanned from the highest index to the lowest until the first non-empty priority queue is found. The task pointed to by its head will then be scheduled. Every scheduled task is re-inserted at the end of its priority queue. See **Figure 29**.

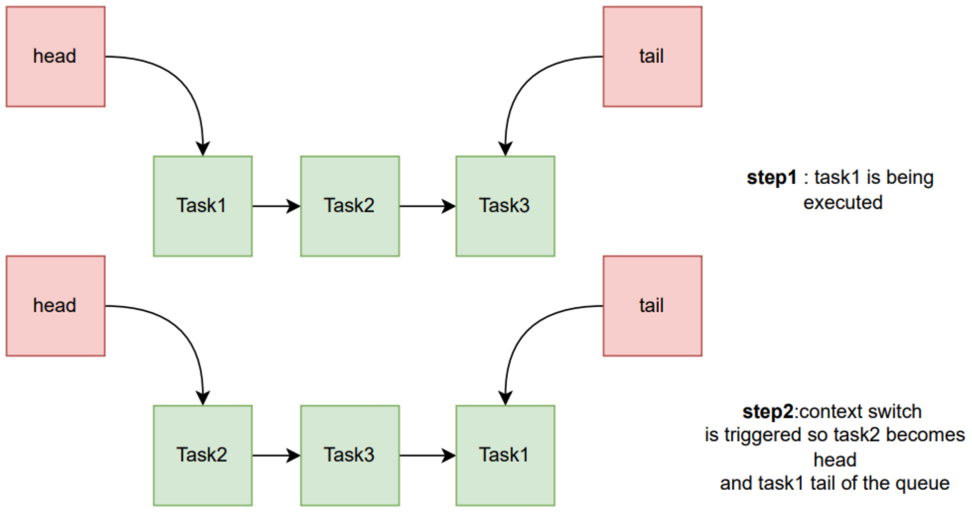

head
tail
Task1
Task2
Task3
step1 : task1 is being executed
head
tail
Task2
Task3
Task1
step2:context switch is triggered so task2 becomes head and task1 tail of the queue

Figure 29: task management during scheduling.

Considering for example a 5-level priority system it means that 5 priority queues will be instantiated and the head of the first non-empty highest priority queue will be scheduled.

The flow chart in **Figure 30** shows in detail the extraction of the next task to be scheduled algorithm. In the first branch the scheduler checks if the current running task is the tail of its queue, because this might mean only two things:

1. The task is the only one in the priority queue, so it does not need to be moved to the tail.
2. The task has changed its priority during execution and has already been inserted at the tail of the new priority queue. Therefore, no further moving operations are required.

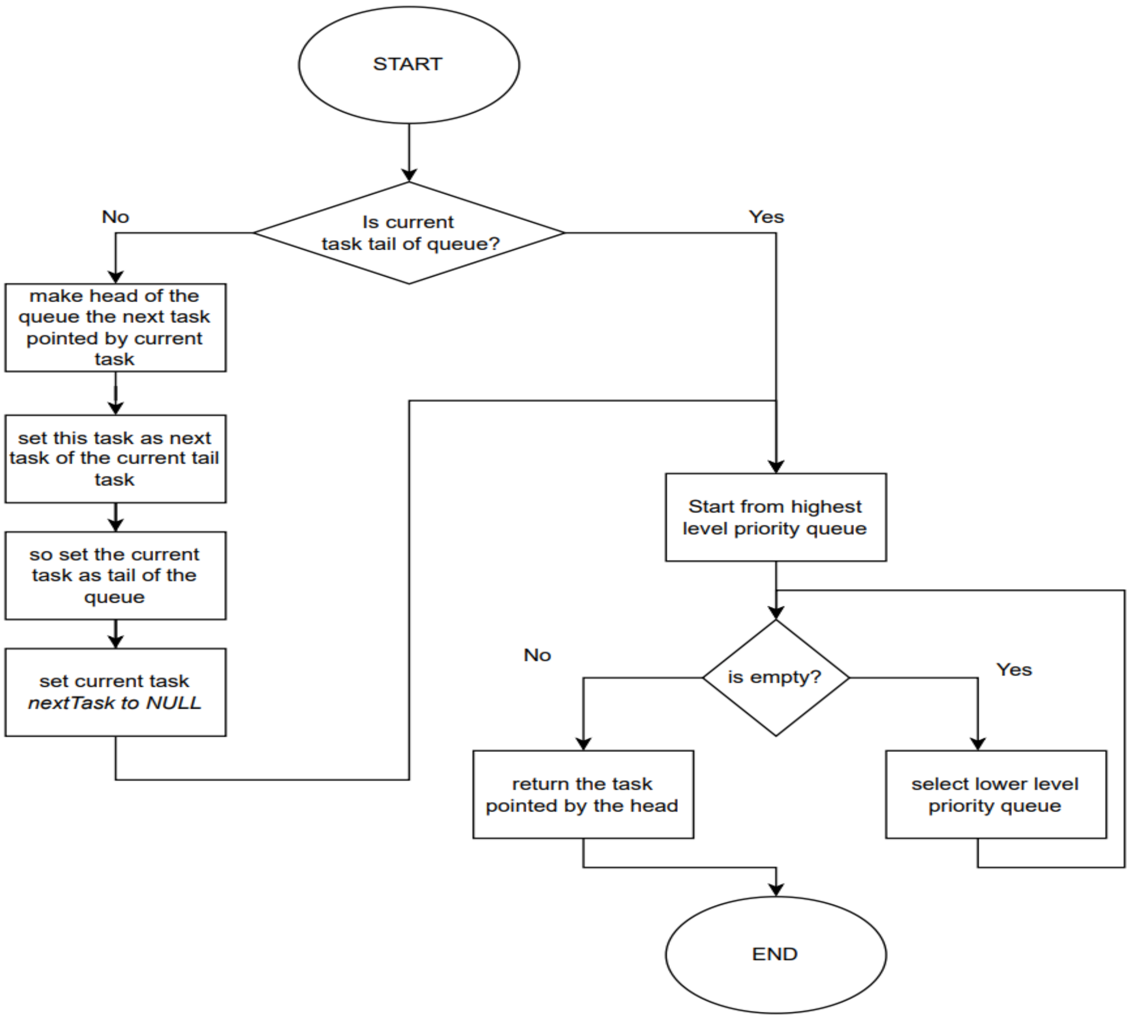

START
Is current task tail of queue?
No
Yes
make head of the queue the next task pointed by current task
set this task as next task of the current tail task
so set the current task as tail of the queue
set current task *nextTask to NULL*
Start from highest level priority queue
is empty?
No
Yes
return the task pointed by the head
select lower level priority queue
END

Figure 30: flow chart of task extraction algorithm.

Remove task from priority queue.

To make the whole system more dynamic and complete tasks should be able to change priority during execution. This means that a task might change its own or the priority of another task. To change the priority of a specific task all is needed is to remove the task from the current priority queue and insert it in the new selected one.

Pause task

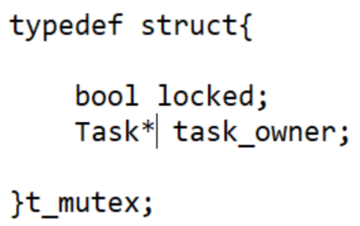
The *PauseTask* function allows a task to be put into an inactive mode. When this occurs, the task is removed from its priority queue and will not be scheduled until it is reactivated. If a task pauses itself, the function does not return to that task because it immediately triggers a context switch by writing to the timer interrupt.

The task can be reactivated by another task using the *ActivateTask* function, which simply re-inserts the task into the priority queue corresponding to its current priority level.

4.4.2 Synchronization mechanism

As mentioned before, the synchronization mechanism available in the quad-core configuration cannot be used here, so a software method is necessary to handle cases in which more than one task wants to access the same resource.

The designed method is based on a simple mutual exclusion (mutex) mechanism. The first task that accesses the resource locks it, and any subsequent tasks that want to access the resource must wait until the locking task unlocks it. To facilitate this, a locking structure has been defined with two members: the first one is a flag indicating whether the mutex is locked or unlocked, and the second member is a pointer to the task that has locked the mutex.



```
typedef struct{

    bool locked;
    Task* task_owner;

}t_mutex;
```

Figure 31: mutex structure definition.

To use a mutex, it must first be initialized. During initialization, the mutex is set to a free state, and the task owner is set to a NULL pointer. After initializing the mutex, two additional functions are necessary to implement this mechanism: one function to lock the mutex and another function to unlock it.

Lock mutex function

The most crucial aspect of implementing the lock mechanism is to guarantee the atomicity of the operation. Two scenarios might occur while a task is trying to lock an available mutex:

1. The operation is interrupted by a context switch, allowing another task to attempt to acquire the same mutex, leading to race conditions.
2. The operation is interrupted by an occurring interrupt handler that might lead to unpredictable behaviors.

These situations must be avoided to ensure that a task can securely lock a resource. Since both worst-case scenarios stem from interrupt sources, the simplest solution is to globally disable interrupts during this operation. The reported flow chart below illustrates the steps involved in this process.

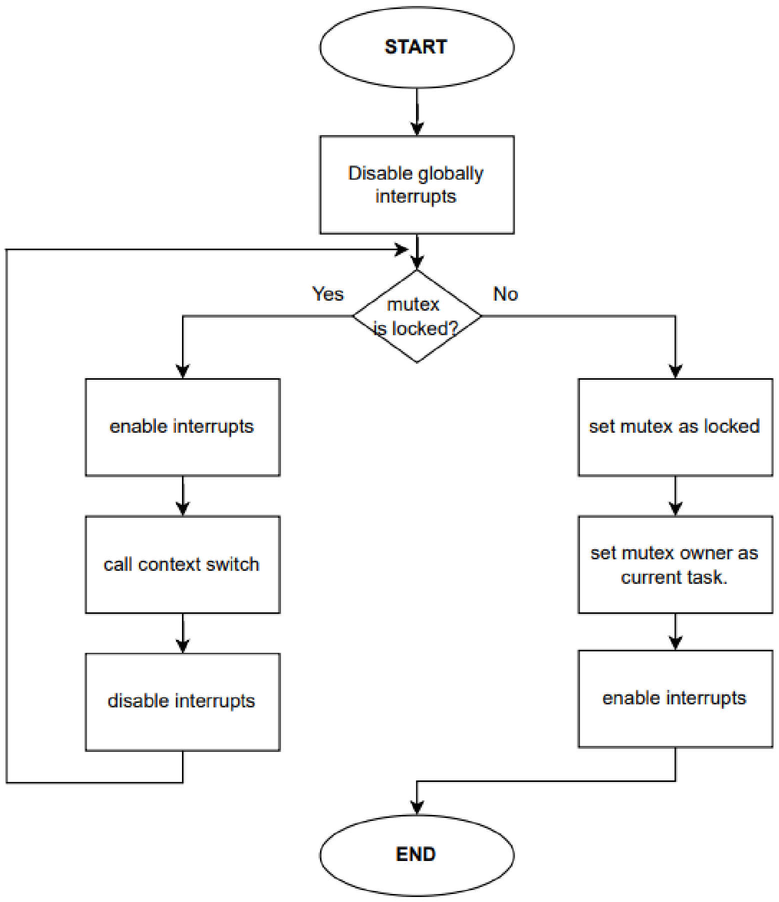


Figure 32: lock mutex function algorithm.

The first step to ensure atomicity, as mentioned, is to globally disable interrupts. Then, check if the mutex is free. If the mutex is free, it is set to a locked state, the mutex owner is updated to the task currently acquiring it, and interrupts are re-enabled.

If the mutex is already locked, there is no reason for the task to wait until the next context switch, as the mutex will not be unlocked during that time slice by another task. Therefore, a context switch is triggered. Before triggering the context switch, interrupts are, of course, re-enabled. When the task is scheduled again, it will disable interrupts and check once more if the mutex is free. This cycle will be repeated until the mutex is freed, allowing the task to lock it.

The unlock mutex function follows the same logic for making the operation atomic.

4.4.4 Overhead analysis

The same overhead analysis conducted in the first case has also been applied to the new scheduling algorithm. Measuring the impact of overhead in this scenario is more challenging, as the algorithm is more dynamic, and the effective introduced delay will also depend on the distribution of tasks in the priority queue.

To simplify the analysis, the worst-case scenario has been considered. All tasks are assumed to be inserted into the lowest priority queue; in this way the scheduling algorithm will follow the longest path for extracting the next task. The overhead will also depend on the number of priority levels introduced since the scheduling algorithm is a scanning loop starting from the highest level to the lowest. In this case, a reasonable value of five priority levels is being considered.

Using the same measurement method, the context switch overhead in this scenario is 335 cycles. At the corresponding frequency of 437.5 MHz, this translates to 789 ns, which is more than twice the overhead of the previous case. Despite this increase, the result can still be considered acceptable.

4.5 Conclusions

By adding priority levels to the scheduler, the single-core solution can now be considered more suitable for real-time environments, where certain tasks must complete their jobs within specific deadlines. Although the solution developed thus far still presents some limitations, it marks a significant first step towards the development of a more complex system.

From an industrial perspective, the results of the context switch impact on operating system performance are very promising, making the single-core solution a viable alternative to the quad-core configuration for future developments. Cutting the need for four cores can increase the available area by removing all the control logic dedicated to managing the quad-core configuration. This freed-up space could be utilized, for example, to increase the RAM memory.

However, determining whether the single-core scheduling solution is superior to the quad-core configuration is not straightforward. The scheduler solution still presents some limitations compared to the quad-core setup. Firstly, the performance of the quad-core configuration remains still higher, which might be crucial for certain applications. Additionally, synchronization between different tasks in the quad-core configuration is provided by hardware, which is significantly faster than the software-based mutex synchronization in the single-core solution.

5. Future Developments

Future development of the work done so far could involve stressing the new functionalities of the scheduler for the STRED_L architecture to implement more robust testing. This would help identify potential issues and ensure the system's reliability under various conditions.

Priority levels enhance task management by ensuring critical tasks receive necessary resources promptly, leading to more efficient and predictable system behavior. This is particularly beneficial in real-time applications. Mutexes ensure controlled access to shared resources, preventing race conditions and data corruption.

However, some new challenges and criticalities can be present with this implementation. Regarding priority levels, lower-priority tasks may suffer from starvation. A solution to this might be introducing mitigating techniques like aging, where the task's priority changes dynamically based on the waiting time without being scheduled. On the mutex side, there might be situations where a lower-priority task holds the mutex required by a higher-priority task, leading to priority inversion. Implementing priority inheritance protocols might be a good solution to this problem.

Another challenge that could enhance system responsiveness is the ability to preempt lower-priority tasks even during their time slice when a higher-priority task becomes ready to execute. This means that if a task with a higher priority than the currently executing task becomes ready, an immediate context switch should occur to allow the higher-priority task to run. In the current implementation, a higher-priority task must still wait for the completion of the defined time slice before it can be executed.

In conclusion, while the implementation of priority levels in scheduling and mutex mechanisms offers significant benefits, it also introduces challenges that must be addressed to enhance performance. By considering potential issues and implementing proper strategies, a robust and efficient scheduling system can be achieved.

Bibliography

[1] D. R. Stauffer, J. Mechler, Michael A. Sorna, K. Dramstad, C. Rosser Ogilvie, A. Mohammad, J. D. Rockrohr, "Serdes Concepts" in *High Speed Serdes Devices and Applications.* Springer. 2008

[2] STRED-L "Architecture Reference Manual Part I: STRED L Instruction Set Architecture". Revision 0.7, 2018.

[3] STMicroelectronics. "S112N3E_LR Architectural Spec 1.1.6 ", 2023.

[4] STMicroelectronics. "S112 LR Micro Subsystem".

[5] Jian-jia Chen, Gregor von der Bruggen, "Non-Preemptive and Limited Preemptive Scheduling", TU Dortmund, 2017.

[6] L. Kishor, D. Goyal. "Comparative Analysis of Various Scheduling Algorithms." International Journal of Advanced Research in Computer Engineering & Technology (IJARCET), vol. 2, no. 4, April 2013.

[7] F. Cottet, J. Delacroix, C. Kaiser, Z. Mammeri. "Basic Concepts" in *Scheduling in Real-Time Systems*. Wiley. pp.282, 2002.

[8] C. Li, C. Deng, K. Shen, "Quantifying The Cost of Context Switch", https://www.usenix.org/legacy/events/expcs07/papers/2-li.pdf

[9] "SerDes." Wikipedia, 2024, https://en.wikipedia.org/wiki/SerDes

[10] K Ravi Kiran, A. Kumar, "Design and Implementation of High Speed Serializer/Deserializer for High Speed Data Transfer Applications." Vol. 63 No. 6 (2020).

[11] R. Mall, "Introduction" in *Real-Time Systems: Theory and Practice,* Pearson Education India, 2009

List of figures

List of tables

Acknowledgements

I would like to thank Professor Maurizio Martina for his guidance throughout this final path of my academic journey.
I also extend my gratitude to my industrial tutors at STMicroelectronics, Luca Longhi and Stefano Antoniazzi, for their valuable feedback and insights, which have enhanced my knowledge and enabled me to complete this thesis.

A final special thanks goes to my family for their unwavering support and patience during these long years of academic studies.