# Embedded Scheduler with Task Reset Capabilities

2023 IEEE 29th International Symposium for Design and Technology in Electronic Packaging (SIITME) | 979-8-3503-4411-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/SIITME59799.2023.10431084

Raul Chechisan
Applied Electronics Department
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
chechisan.va.raul@student.utcluj.ro

Mihai Daraban
Applied Electronics Department
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
mihai.daraban@ael.utcluj.ro

Cosmina Corches
Automation Department
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
cosmina.corches@aut.utcluj.ro

Gabriel Chindris
Applied Electronics Department
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
gabriel.chindris@ael.utcluj.ro

***Abstract*—Embedded systems play an essential role in our daily lives as they function behind the scenes to improve our experiences, streamline processes, and provide convenience. For example, in residential homes, embedded systems can provide security alarms through a network of sensors that detect intruders and detect fires and noxious gases. In smart houses, such systems have become indispensable as they automatically regulate the temperature based on the activity inside the house and external weather changes. These examples prove how embedded systems have been integrated perfectly in our everyday lives, offering comfort and optimizing processes. The primary component behind an embedded system is the microcontroller and the application written in its memory. The tasks of embedded systems are typically controlled via an operating system that manages its actions. However, at times, because of malfunctions or unexpected behavior, the system may freeze. To recover from such a situation, embedded systems may require end user intervention or a system reset using a watchdog interrupt handler. The paper presents an embedded scheduler with task reset capabilities. Through this approach, the system control and end user experience were considerably improved.**

***Keywords—embedded system, microcontroller, operating system, task, scheduler.***

## I. Introduction

The operating system (OS) in an embedded system is among the most important components as it manages the functionality of the system [1]. Compared to a computer OS, an embedded OS offers limited features and functionalities through which it attempts to ensure that the actions performed are consistent and performed at determined moments in time, Fig. 1.

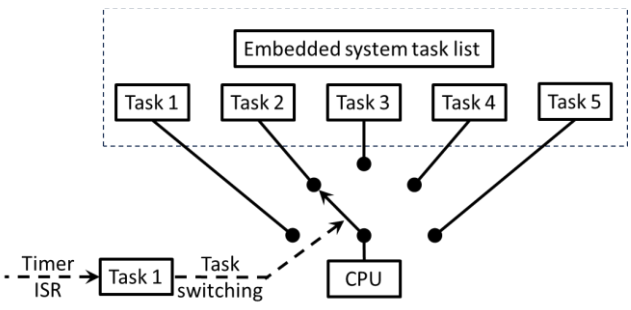


Fig. 1. Embedded system resource management.

In case of an embedded system, the system costs must be as low as possible; hence, microcontroller resources are limited (e.g. low frequency, small memory). This is also reflected in an embedded OS and scheduler, with both not having the capabilities of a desktop OS to overcome a situation wherein a task is no longer responding or relinquishing the resources. A watchdog peripheral is commonly used to perform a system reset in an attempt to recover the system operation [2, 3]. However, in case of a sensor malfunction or a communication interface issue, the system may revert to a frozen state.

Reinitializing a task in an embedded system is challenging. As such systems have dedicated functionalities, each action performed by the embedded OS aids in achieving the purpose for which it was designed. Consequently, in majority of the designed embedded systems, the approach involves reinitializing the microcontroller and the application through a software reset. However, the proposed mechanism of reinitializing a frozen task can be used on computational tasks, on those responsible for controlling peripheral communication, or those that read data from the sensors. For such tasks, there exists no risk of damaging the embedded system, and the user experience is improved by managing the downtime in the background.

In this paper a scheduler with improved task management for embedded systems will be presented. The system will still rely on the watchdog to reset the embedded system if one of the tasks is freezing. To enhance the system performance and provide a better user experience, through the proposed approach, additional checks on the tasks during runtime are done instead of going straight to resetting the system. The system will be reset only after a few attempts to restart the problematic task.

## II. Embedded System OS and Scheduler

An embedded system is a combination of hardware and software, which performs a designated function. In certain scenarios, the embedded system works within a larger system, wherein is responsible for a single functionality in a multifunction device. The embedded systems can be considered as tiny computers, with small memory and processing power compared to the current personal computers (PC). Designed to focus exclusively on its intended purpose, a built-in system lacks the versatility of a PC.

### *A. Operating Systems for Embedded devices*

The OS used for an embedded system are cooperative or noncooperative. In case of cooperative OS, predicting the task execution and performing an analysis over the minor and major cycles is easy. However, for advanced applications organizing the tasks and managing the timings can become complex; thus, noncooperative OS are preferred for such scenarios. Compared to a cooperative OS, a noncooperative OS is more complex because the OS scheduler is responsible for saving the CPU registers' state on the task's dedicated stack, Fig. 2.

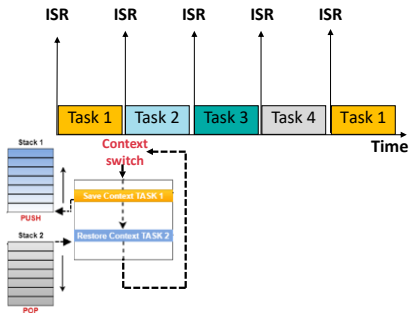


Fig. 1. Context switching mechanism in a non-cooperative embeded OS

In a cooperative OS the context switching mechanism is handled automatically by the CPU; however, this imposes a constraint on the tasks. In this case, the tasks must terminate their activity and relinquish the resources until the next timer interrupt (i.e., scheduler calls). Consequently, in case of a frozen task, the system operation will be affected. Thus, appropriate monitoring is required to prevent such situations.

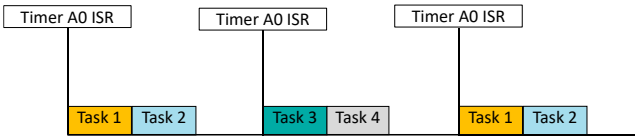


Fig. 2. Task execution in a cooperative operating system

The purpose is to prevent the calling of the piece of code that caused the task to freeze. Another approach involves marking to ensure deletion of the task from the scheduler list. These approaches allow the system to continue to operate; however, certain functionalities are turned off or the performance is decreased.

### *B. Task revive*

Task reviving mechanism can be observed on a PC through the Windows OS. This is possible through the system's process management capabilities, error handling, and recovery mechanism:

- Task monitoring – tasks and processes are constantly monitored for an unresponsive state.
- Timeouts – each task is assigned a timeout. When no response is obtained within the specified timeout period, the OS considers it unresponsive.
- Error handling – when an unresponsive task is detected, different actions are taken: displaying a message to the user, attempting to recover the task, and forcefully terminating the task.
- Restarting – attempt to restart a task or a process through restarting a service or restarting an application if it is configured to do so.

A general-purpose OS, such as Windows, will always have differences in capabilities and design goals compared to OS designed for microcontrollers [4]. Real-time OS (RTOS) are resource-constrained embedded systems with specific real-time requirements. A mechanism such as that of Windows, would contravene with the following limitations of an embedded system:

- Resource constraints - microcontroller systems have limited resources (e.g., CPU, memory etc.), and adding sophisticated error-handling mechanisms can be impractical due to these constraints.
- Determinism - restarting a task unpredictably may violate real-time requirements, as the restart time may not be predictable.

The automotive industry has proposed certain approaches to mitigate such situations wherein a task freeze affects the system operation. Automotive Open System Architecture (AUTOSAR) proposes the implementation of function inhibition manager (FIM). FIM was designed to manage the inhibition of certain functions or services within an embedded automotive system. It plays a crucial role in ensuring the safety and reliability of the system by controlling when specific functions can be activated or deactivated based on certain conditions.

FIM implementation is reliant on four components: monitoring the defined conditions, condition evaluation, function control, feedback and recovery. As implied by the first component, the FIM continuously monitors the sensor data, vehicle speed, and system health to gather the data for condition evaluation. The second component evaluates the conditions against predefined rules or thresholds defined in AUTOSAR to inhibit the associated functions. The inhibition process is realized through function control component, which involves sending commands to deactivate specific components or services. Other parts of the system, including the human-machine interface and diagnostic system, are informed of the inhibited functions. Depending on the implementation, the functions can be reactivated when the conditions become favorable again.

The FIM implementation complexity is dependent on the number of monitored conditions. In case a component is also related to safety, the implementation requires addition of redundancy and a fault tolerance mechanism. Imposed by AUTOSAR, FIM module must be integrated with other modules, such as Diagnostic Event Manager and the Communication Stack, which can increase the system complexity.

To use FMI in a cooperative embedded OS the following criteria should be fulfilled:

- Task coordination – FIM must be implemented as a task that cooperates with other tasks to inhibit or enable functions as needed.

- Resource management – access to shared resources is required, such as sensor data or communication channels.
- Event-driven model – monitoring for events (e.g., sensor data changes). The cooperative OS should support event-driven programming or a polling mechanism to supply data for FMI monitoring module.

For an embedded system that does not raise safety concerns toward the end user, the FIM implementation could prove to be excessive, resulting in an increase in the product price and development.

As evident from the analyzed solutions, task revitalization can be undertaken on embedded systems [5]; however, it raises the concern of what the impact of such an implementation would be over the system. In a PC OS, this is possible as the scheduler is considerably more advanced in managing the resources and the memory, and in automotive industry the final product performance and safety justify the resource and price.

## III. Proposed Solution

In embedded systems, a technique to verify a task operation or if the system is unresponsive is through the use of the watchdog peripheral. However, the watchdog approach is not capable of task reviving; rather, it performs a software reset in case the watchdog counter is not reset at predefined moments in the application to prevent it from overflowing.

To enhance the system performance and provide a better user experience, an additional check on the tasks activity is done during scheduler runtime to decide task interruption and reinitialization. The proposed approach can be used on those tasks that allow for such a mechanism (e.g., a communication task, the task responsible for sensor reading etc.) and do not create a risk for the embedded system hardware.

### *A. Timeout handler*

A cooperative operating system relies on a timer interrupt request to execute the scheduler. Based on the properties specified in the task control block (TCB) and the scheduling algorithm, the scheduler determines the tasks that should be launched. By adding an additional timer (i.e., timeout timer), it can be examined whether the scheduler could run the designated tasks and whether their status was switched to finished or waiting to be launched when it is time for the corresponding task execution. If the tasks do finish in time, the scheduler can turn off the second timer, thereby preventing the timeout mechanism from being triggered (i.e., launching the timeout handler).

For the timeout mechanism to work, the second timer interruption must be set to a higher priority than the priority of the scheduler timer such that the microcontroller automatically switches the context. The timeout handler can be considered as an additional task that will be executed in case the planned tasks are unresponsive. The execution timing of the additional task must be considered when designing the minor cycles for the cooperative OS. If the minor cycles are not designed correctly, there exists a risk of interfering with the system determinism, Fig. 3.

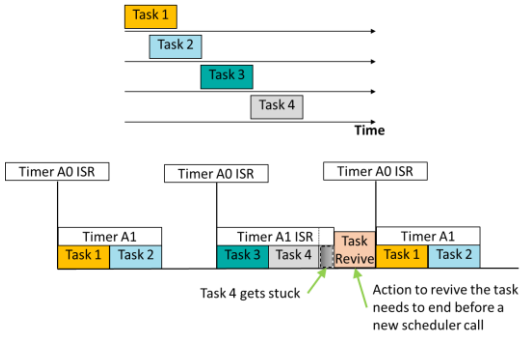


Fig. 3. The revive action needs to be considered durring operating system minor cycle design

The timeout handler identifies which task is unresponsive and inhibits the scheduler's execution as expected. Based on the task proprieties and system design, adequate measures can be taken to reinitialize the associated peripheral or reattempt a task launch at the next scheduler run.

After timeout handler execution, the task should relinquish the resources and return to OS scheduler execution for resetting the watchdog and end the associated timer handler execution before the next timer ISR.

### *B. Task design*

When designing the OS's tasks, the potential scenarios where a task might become unresponsive must be identified. In such cases, a "freeze flag" to abort the task can be defined. The freeze flag can be defined as a propriety in TCB, which at initialization can be set to false to not interfere with the task execution, Fig. 4.

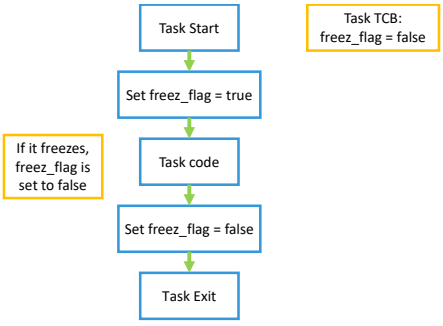


Fig. 4. Task control block freez flag

In addition to the freeze flag, each task should have associated filed counter in the TCB for the maximum reviving attempts. When the associated threshold is reached, the timeout timer should no longer revive the task. Thus, the watchdog timer can reset the system. This is necessary when the task's operation is dependent on the output of other tasks, and in such cases, a system reset is the only way to restore proper system operation.

Moreover, when a task freezes, the execution of the tasks in the minor cycle is affected. Hence, the proposed implementation can be used in systems that are not safety critical.

The block diagram of the mechanism of the proposed system is presented in Fig. 5. The diagram is divided in four

sections: the initialization block (blue color), the block responsible for running the tasks (orange color), the block responsible for task reinitialization (green color), and the section that performs reset owing to watchdog peripheral (red color).

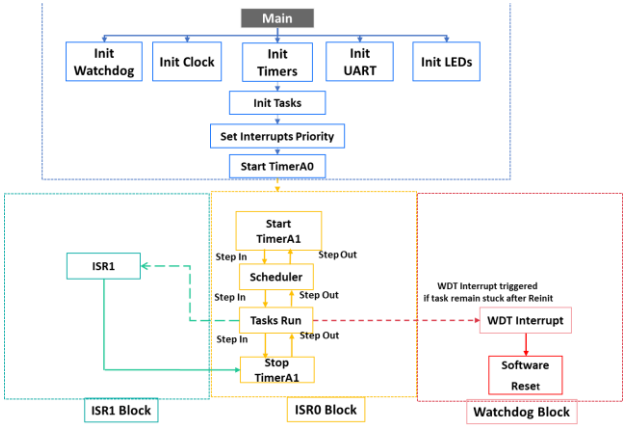


Fig. 5. Proposed system diagram block

## IV. Conclusions

This paper proposed a technique to revive the tasks inside a cooperative operating system. Rather than removing the task execution from the scheduler list, the proposed approach signals to the system the task that is unresponsive in the attempt to unlock the operation. The implementation was intended for non-safety-critical embedded systems, wherein certain tasks not being executed at designated timings was tolerated. In future development we plan to improve the control over the task timeout, without adding excessive overhead. The advantage of the proposed approach is not introducing excessive overhead to the TCB specific to a OS implementation. Moreover, it is easy and flexible in terms of implementation.

## References

[1] M. J. Pont, "Creating an embedded operating system" in Embedded C, Pearson Education Limited, 2002, pp. 143-187.

[2] R. K. Unni, P. Vijayanand and Y. Dilip, "FPGA Implementation of an Improved Watchdog Timer for Safety-Critical Applications", 2018 31st International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID), Pune, India, pp. 55-60.

[3] S. F. Taj and M. C. Annamala, "Modified VLSI Architecture of Watch Dog by Windowing Technique", International Journal of Scientific Engineering and Technology Research, Vol. 8, pp. 607-612, Jan-Dec, 2019.

[4] B. L. Belasco, "High stability Windows programming for real time control," 44th Annual 2010 IEEE International Carnahan Conference on Security Technology, San Jose, CA, USA, pp. 127-133, 2010

[5] I. Christoforakis, O. Tomoutzoglou, D. Bakoyiannis and G. Kornaros, "Runtime Adaptation of Embedded Tasks with A-Priori Known Timing Behavior Utilizing On-Line Partner-Core Monitoring and Recovery," 2014 12th IEEE International Conference on Embedded and Ubiquitous Computing, Milan, Italy, pp. 1-8, 2014,