Advanced µkernel on Arduino

André Campanhã
FEUP
Porto, Portugal
up201806518@fe.up.pt

Miguel Almeida
FEUP
Porto, Portugal
up201806205@fe.up.pt

Rúben Almeida
FEUP
Porto, Portugal
up201704618@fe.up.pt

***Abstract*—In this project it was implemented a real-time tick-based, fixed priority, full-preemptive µkernel for the *Arduino Uno* where each task has its own stack. To share resources, mutexes were used and to solve priority inversion problems the Immediate Priority Ceiling Protocol was employed. The context switching functions required were implemented using Assembly and the remaining kernel functions were implemented in C. A simple task set with seven tasks was developed in order to test various functions of the kernel. In addition, measurements were taken in order to conclude that the overhead is low and therefore make this µkernel suitable for real-time applications.**

***Index Terms*—AVR, Arduino Uno, µkernel, microkernel, Context Switching, ISR, IPCP, Mutexes**

I. Introduction

A real-time µkernel allows for the ability to handle multiple tasks simultaneously, while trying keep resource use to a minimum.

The objective of this project is to develop a tick-based full-preemptive µkernel for *Arduino Uno*, that supports multi-threading and where each task, running on an individual thread, has an independent stack. The µkernel uses fixed priority scheduling to schedule tasks. Besides that, in order to be able to have tasks that share resources, the team implemented mutexes, using the *Immediate Priority Ceiling Protocol* to solve priority inversion problems.

II. Approach

The code for the µkernel was implemented using *Platform IO* in *Visual Studio Code*, with the target being the implementation in an *Arduino Uno*, which uses the AVR ISA-based *Microchip ATmega328P* microcontroller. To aid the development, a special debugger, *SimAVR*, was used. *SimAVR* is a simulator that works on any platform that supports `avr-gcc` and allows us to simulate an Arduino implementation while debugging, being able to access every signal and the memory registers of the targeted platform. This proved to be a massive aid in the development process, allowing the team to see precisely what was happening in the registers and stack. It helped in detecting errors and provided some clues into possible solutions for them.

The implemented µkernel is a tick-based, fixed priority, fully-preemptive kernel, where, unlike the kernel implemented in the lab classes, where all tasks shared the same stack, each individual task has its own stack, leading to the existence of multiple stacks. The size of each tasks stack was defined manually after analysing simple programs using *SimAVR*, where the value of 100 Bytes was determined to be enough to save the context and use in normal operation. The total memory used for the task stacks was predicted to be the number of tasks times the allocated individual task size, in this case $8 \times 100 = 800$ Bytes. This memory was allocated and declared as a global variable. Each task will have access to an equal portion of this array, acting as its own separate individual stack.

Mutexes were implemented to share resources, and the *Immediate Priority Ceiling Protocol* was used in order to solve the priority inversion problem.

III. Kernel Architecture

This section covers the architecture defined to support our solution for a full preemptive multi-thread kernel for *Arduino Uno*.

A. Task Execution Flow

Each of the seven tasks defined to demonstrate the kernel execution follow a POSIX style definition [1].

This definition, in opposition to non-real-time programming, is widely used in the Real Time (RT) industry, being the approach selected by FreeRTOS [2], a proved RTOS, to define their tasks with time requirements.

In this standard, tasks execute an infinite while loop and their preemption are made in two scenarios:

- Voluntary Resource Release — Each Task can notify the end of its periodic execution by invoking `schedule_yield` when it has finished its intended execution;
- Coercive Preemption — If a task depletes its execution time by reaching its deadline, it will be forced to release its resources by the `schedule_schedule` function.

In the solution proposed, due to the low resource demand of the tasks proposed when compared with the hardware capabilities at our disposal, the task preemption is always expected to be voluntary, turning `schedule_yield` into a relevant function in our architecture.

B. The Context Switching Process

The requirement for a runtime full preemptive kernel forced the introduction of context-switching mechanisms.

Context switching is the name for techniques that correctly change the CPU execution between two different threads/tasks.

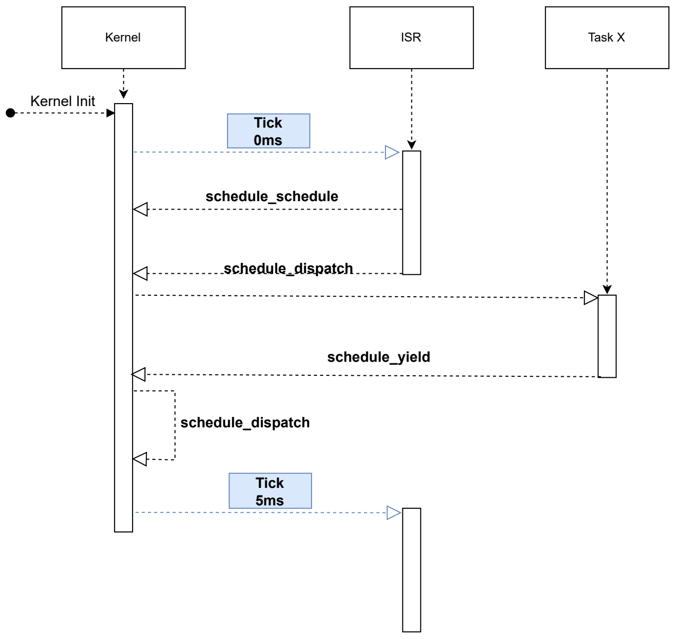


Figure 1: UML Sequence Diagram illustrating the task execution flow.

Every context-switching process requires at least two actions, a saving and a restore context action.

A saving context mechanism like the one implemented in our solution only requires saving the value of the CPU registers to memory, more precisely to the stack assigned to each task. Due to the heterogeneity of CPU architectures, this task requires assembly code hardware-specific to gain privileged access to low-level elements like the CPU registers and stack manipulating capabilities.

In the case of the *Arduino Uno*, the *ATmega328P* CPU has 32 general-purpose 16-bits registers, one 16-bit status register (SR), one 32-bit program counter (PC) and two 16-bit stack pointers (SP). All these registers must be saved in the saving context script.

The restore context is an inverse operation from previously described saving context operation. Restoring the context in the *ATmega328P* requires loading from the memory segment previously filled, the 32 general purpose registers, the SR, the PC and the two SP.

In order to reduce the specific cases and obtain a more generic code, resilient to bugs, we decided to initially load the task stack with a initialization context.

A scenario where the 32 general purpose registers have the value 0x00, the status registers is loaded with 0x80, that according to [3] is the SR value that globally enables interrupts.

C. Naked Functions

While performing a low-level operation like registry saving/loading from a stack manipulation, programmers want to avoid the compiler generating extra code or manipulating the value of registers without explicit instruction from the programmer to do so. This behaviour is obtained by introducing the GCC attribute 'naked'.

A naked function also "prevents the compiler from generating any function entry or exit cod" [4]. This forced us to specify the exit code explicitly when we used naked functions in our solution.

D. The Timer

Our proposal is based on the periodic execution of an Interrupt Service Routine (ISR). In order to obtain a periodic interrupt, we configured timer 1 of the Arduino Uno to raise an event every $5\,\mathrm{ms}$.

The ISR execution is expected to periodically invoke the scheduler to update the task deadlines and verify what task should run each time based on the current priorities. After selecting the candidate task, the ISR invokes the dispatcher to resume the execution.

The ISR process is, for this reason, a preemptive action by nature. This fact forces the ISR to be a naked function and the first action of ISR to be a context-saving action.

Due to the naked nature of the ISR, we added a *reti* assembly instruction to perform a return from interrupt instruction. The usage of *reti* forces the hardware to update the PC to the value pointed by the SP. If the SP is correctly configured, the execution will execute in the scheduled task.

E. The Idle Task: A Special Type of Execution

In the context of RTOS, it is common for a system to reach a state where all deadlines are accomplished, and the system has no task to run until the next period. The system is said to be idle.

However, the concept of idleness is strange to the nature of a CPU. This component must always have a task executed.

There are several solutions to solve this contradiction, our proposal, inspired by the way FreeRTOS address this problem, was the introduction of an idle task.

The idle task is similar in construction to all other tasks that produce relevant work, yet it is only composed of an empty while loop. It is the lowest priority task with a period of a single tick, meaning that it is available to run every time.

This task only runs when no other valuable task is available to execute.

F. The Kernel API: Functions Provided by the µkernel

Our µkernel implementation is founded in three functions that manage how it behaves.

The `scheduler_schedule` is the function that encapsulates the scheduling task itself. It is the first logic to run after each timer interrupt.

Due to the fixed priority behaviour of our scheduler, `scheduler_schedule` ensures that every time a high-priority task is available it will run, even if this forces a context switching from a lower priority running task.

In addition to selecting the best candidate according to the scheduling policy to run each time, the `scheduler_schedule` also updates the controlling structure that keeps track of the next task's activation and execution time left.

After the `scheduler_schedule` selects the candidate that must run in the next CPU cycle, the task must be pushed into execution, that is, it must be dispatched.

This process of dispatching a task into CPU execution is done by the `scheduler_dispatch` function. This function restores the context of the to-be-run task (composed of the CPU registers, the SP, and the SR) and restores the PC to where it was when preempted, or the beginning when it has not been ran before, with the *reti* instruction, that sets the value of the PC to the last two bytes in the stack. From this point onward, the execution continues in the task's code.

The `scheduler_yield` is the function every task invokes to signal the kernel that its execution is done and it is ready to release its resources. This function is similar to the ISR since it performs a save context followed by a `scheduler_dispatch`. It is also a naked function to ensure that GCC does not modify the registers during the yield invocation.

G. Mutexes: Immediate Priority Ceiling Protocol

A mutex structure was implemented that allows a task to lock a resource (`mutex_lock`), therefore preventing any other tasking from using it until the first unlocks it (`mutex_unlock`). However, a naive implementation of such primitive can lead to deadlocks and unbounded priority inversion. Therefore, the Immediate Priority Ceiling Protocol (ICPC) [5] was implemented. This protocol raises the priority of a task when the the resource is locked. According to [5], it is defined as:

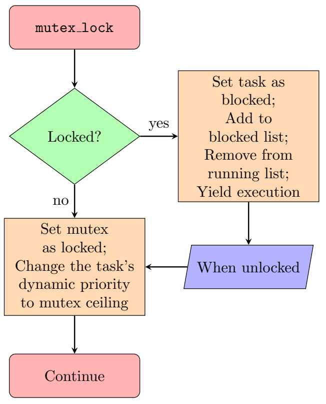
- Each task has a static default priority.
- Each resource has a static ceiling value defined as the maximum priority of the tasks that use it.
- A task has a dynamic priority that is the maximum of its own static priority and the ceiling values of any resources it has locked.

For this, a sorted (by the dynamic priority) list of ready to run tasks was implemented in the kernel, and of blocked tasks in each mutex. The algorithm for locking and unlocking the mutexes can be seen in Figure 2.

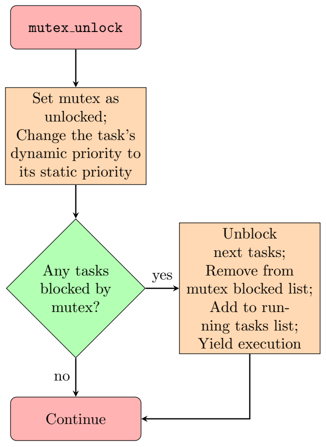
IV. Benchmarking

In order to quantify the performance of our µkernel, some measurements were made of critical sections of the code. An analysis of the context switching overhead (the time that it takes from the preemption of a task to next one starting running) was made by varying the number of tasks in the kernel and taking 50 samples for each one. The results in Figure 3 show that there when the task set size grows, the time it takes to switch context grows somewhat linearly.

Measurements of the mutex lock and unlock execution times where also taken by sampling each function 50 times as can be seen in Table I. It should be noted that the scenario where the mutex was locked was not taken into account, as the time it would take to acquire the lock would be dependent on the other task that had it locked.


mutex_lock
Locked?
yes
Set task as blocked; Add to blocked list; Remove from running list; Yield execution
no
Set mutex as locked; Change the task's dynamic priority to mutex ceiling
When unlocked
Continue

(a) Mutex lock


mutex_unlock
Set mutex as unlocked; Change the task's dynamic priority to its static priority
Any tasks blocked by mutex?
yes
Unblock next tasks; Remove from mutex blocked list; Add to running tasks list; Yield execution
no
Continue

(b) Mutex unlock

Figure 2: Diagrams of the algorithms for locking (a) and unlocking (b) the mutexes.

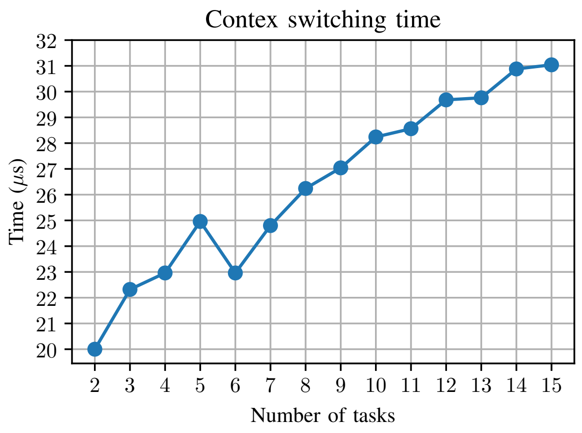


Figure 3: Analysis of the context switch overhead when the number of tasks grows.

Table I: Average execution time of `mutex_lock` and `mutex_unlock` functions.

Function	Average Time (µs)
`mutex_lock`	4.08
`mutex_unlock`	4.24

V. Conclusion and Future Work

With the implementation of the kernel described, we could fulfil this project's requirements. Our multi-task full-preemptive solution introduces multi-threading capabilities to Arduino Uno.

Our testing suite collected timing measurements of the kernel function. The results show that it operates as expected with low overheads, and the schedulability for the seven tasks design is guaranteed.

We extended the project's scope by introducing the kernel IPCP resource policies. With this, we were able to fulfil our goal of introducing mutexes. Our final solution introduces critical sections to Arduino.

Future improvements to this project include implementing different scheduling policies, like the EDF scheduling policy, and creating an abstraction that allows the kernel to operate using different scheduling policies. This allows for more flexibility in kernel operation, allowing the execution of different task sets operating in EDF and Fixed Priority, for example.

Acknowledgement

We want to thank to FEUP computing engineering students João Costa and Ricardo Fontão for advising us to use SimAvr integrated with Platform IO on Visual Studio Code as well as some precious tips on how to approach this project. They were SETR students in the previous academic year. We want to thank them for the time they saved us with their guidance.

References

[1] M. d. Sousa, "Embedded software architectures," Dec 2022.

[2] "Writing rtos tasks in freertos - implementing tasks as forever loops," Jan 2022. [Online]. Available: https://www.freertos.org/implementing-a-FreeRTOS-task.html

[3] "The status register," 2018. [Online]. Available: http://www.rjhcoding.com/avr-asm-sreg.php

[4] R. Barry, "Multitasking on an avr," *Recuperado de https://es.scribd.com/document/49299134/Multitasking-on-an-AVR*, 2004.

[5] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*, 4th ed. Pearson Education Canada.