Developer focused Real-Time micro-Kernel for Arduino UNO

Davide Castro
FEUP
Porto, Portugal
up201806512@edu.fe.up.pt

Henrique Ribeiro
FEUP
Porto, Portugal
up201806529@edu.fe.up.pt

João Costa
FEUP
Porto, Portugal
up201806560@edu.fe.up.pt

***Abstract*—In the universe of embedded systems, there are multiple Real-Time micro-Kernels available. This work documents the development of a new micro-Kernel for the Arduino UNO. The main focus of this Kernel is on developer usability, and keeping a low memory footprint, so more code fits in the device.**

***Index Terms*—Kernel, micro-Kernel, EDF, ICTOH, Scheduling, Real-Time, Mutex, PIP, Arduino UNO**

I. Introduction

The Kernel is the basis of any Operating System. In the world of Real-Time systems, the Kernel is responsible for scheduling and dispatching the tasks, according to Real-Time constraints. In this project, the group designed a Kernel to run on an Arduino UNO [1] (R3) board (ATmega328P), based on work done for FreeRTOS [2].

There are multiple protocols that can be used to schedule tasks. The developed Kernel uses the Earliest Deadline First (EDF) algorithm, because it is optimal (if there is a feasible scheduling for a given task set, EDF finds it), and doesn't require extra information from the developer. Following the idea of developer usability, the group implemented extra features like mutexes, assertions, and stack guards.

The code, documentation, build system, and benchmark data can be found on the project's repository, and a demonstration video on YouTube.

II. Architecture

This section contains the description of both the micro-Kernel, and its components.

A. Tasks

The Kernel schedules periodic Real-Time tasks. These tasks need an execution period, a deadline, the code to execute (see section II-B), and the size of its stack. With this, the period of the task can be different from its deadline. Additionally, tasks can have an initial delay (to offset tasks).

A task can be in any of four states: ready, blocked, not ready and waiting. These states symbolize, respectively, that the task is ready to be executed, is blocked on a shared resource, is not ready to execute, and is sleeping (artificial extension of execution time).

B. Task's code

Each task contains a pointer to the function it executes, and an optional argument to pass to this function. The design of these functions is similar to the ones from POSIX threads: the functions are only called once, and take a C void pointer (`void*`) as argument.

Since each function is only called once, it should be prepared to repeat execution. For purpose, task functions start with a setup section, followed by a loop containing the code that should be executed in each activation of the task. It is imperative that each tasks yields (calls the `Sched_Yield` function) when it finishes all work for the current activation.

C. Stack

The Kernel assigns a stack for each task (multi-stack Kernel), meaning that each task allocates a section on memory for their own stack, keeping its address. The task's developer is responsible for determining the size of the stack needed the code of each task. This size should be kept as small is possible, specially in memory constrained environments.

In order to make both determining this size and debugging easier, developers can activate stack canaries during compilation using the `-DDOCANARIES` GCC compiler flag. When active, the Kernel will emit code that reserves some bytes at the beginning and at the end of each task's stack, and fills them with pre-determined values. Then, it evaluates the integrity of these values every time a task *yields* (finishes work for an activation). Upon detecting a problem, the Kernel goes into the **failure state** (see section II-D).

D. Failure state and asserts

When the assertion functionality is active, the Kernel performs runtime checks on multiple critical operation conditions. These include: failure of memory allocations, out-of-bound vector accesses, and alignment of stack addresses. Developers can activate this functionality by the `-DRUNTIMEASSERT` GCC compiler flag. To add an assertion to the code, the developer needs to write the following line of code: `assertCond(condition, message)`, when `condition` is the condition that should be true, and `message` is the message to emit in case of failure.

When an assertion fails, the Kernel enter the **failure state**. In this state, the Kernel emits a message about the error, as well as the function, the file, and the line where it was found. Afterwards, the Kernel gets stuck in a cycle of blinking the devices built-in LED, spelling *SOS* in Morse code.

E. Scheduler

For the scheduling of tasks, the kernel uses the Earliest Deadline First (EDF) algorithm. In this algorithm, the Scheduler assigns priorities dynamically to each task. The priorities are inversely proportional to their deadlines, i.e., the earliest the deadline, the higher the priority.

The scheduler keeps a list containing every task. Upon every tick of the Kernel, the scheduler updates the tasks on the list: decrements the delay until the next activation and the sleep time, sets tasks as *ready* (if needed), and sorts the tasks.

1) Timer implementation: The main problem with EDF is the implementation of absolute deadline. As the time passes and tasks complete activation, deadlines get pushed further in time. This Kernel, represents deadlines using 16-bit unsigned integers. “Notice that fixed priority schedulers do not have this problem, since periods and relative deadlines can be mapped into a set of priority levels and do not have to be explicitly represented” [4]. As such, the system as described has a maximum lifetime, P, corresponding to the overflow of the deadline representation: in 16-bit representation, $P = 2^{16} = 65536$.

To circumvent this problem, the group implemented the approach described by Giorgio Buttazzo et al. in ’Efficient EDF Implementation for Small Embedded Systems’. With this, the Kernel represents the deadline’s time cyclically: the Implicit Circular Timer’s Overflow Handler (ICTOH) algorithm.

Although the ICTOH algorithm solves the deadline representation overflow problem, it introduces a second constraint: two deadlines can’t be further than $P/2$ time units apart. This happens, because if the system doesn’t make that assumption, it is impossible to be sure which deadline comes first when comparing any two dead deadlines. See ’Efficient EDF Implementation for Small Embedded Systems’ [4] page 5 for the full explanation.

With this algorithm in place, two deadlines are compared according to Tab.I, where $\ominus$ is a subtraction modulo P.

TABLE I
COMPARING DEADLINES WITH THE ICTOH ALGORITHM

Comparison	Expression	Meaning
$t(e_i) > t(e_j)$	$(e_i \ominus e_j) < \frac{P}{2}$	e_i is later than e_j
$t(e_i) < t(e_j)$	$(e_i \ominus e_j) > \frac{P}{2}$	e_i is earlier than e_j
$t(e_i) = t(e_j)$	$(e_i \ominus e_j) = 0$	e_i and e_j are at the same time

Since the modulo operation is expensive and the Kernel uses 16-bit integers to store the deadlines, it is possible to compare the deadlines as signed integers and simplify the comparisons. Tab.II shows the new comparisons (note that the $<$ and $>$ signs in the expression column change). See ’Efficient EDF Implementation for Small Embedded Systems’ [4] section 3.3 for the full explanation.

TABLE II
COMPARING DEADLINES WITH THE ICTOH ALGORITHM (OPTIMIZATION)

Comparison	Expression	Meaning
$t(e_i) > t(e_j)$	$(e_i - e_j) > 0$	e_i is later than e_j
$t(e_i) < t(e_j)$	$(e_i - e_j) < 0$	e_i is earlier than e_j
$t(e_i) = t(e_j)$	$(e_i - e_j) = 0$	e_i and e_j are at the same time

F. Dispatcher

The Dispatcher gets the first element of the task list (sorted by the Scheduler (see section II-E)). If this task is ready and has higher priority than the currently running task, the current task suffers preemption in favor of the new one. To do this, the Dispatcher saves the execution context of the current task in its stack, and load the new task’s execution context from the new task’s stack. FreeRTOS’s documentation [3] describes and was the reference for the implementation of this process.

Additionally, when a task *yields*, the dispatcher immediately passes the execution to the highest priority ready task, thus ensuring **maximum uptime**. Furthermore, this also happens when a task *sleeps* (calls `Sched_Sleep(ticks)` to extend its execution time).

1) Idle task: In some circumstances it is possible for the Kernel to have no tasks ready to run at a given point in time. When this happens, the Kernel passes execution to the *idle task*. This task is an empty loop that never yields. As such, it is always ready to execute.

When the Scheduler is sorting the tasks, it always places the *idle task* as the lowest priority of the ready tasks. With this, the Dispatcher doesn’t need any special considerations when deciding which ready task to execute, as the *idle task* is always ready.

G. Resource Sharing and Mutexes

The Kernel supports using mutexes for the purpose of resource sharing between tasks. With these, developers can protect critical regions in their tasks’ code. To use a *mutex*, developers just need to instantiate one (e.g, `Mutex* mut = new Mutex();`, and make sure that that their tasks have access to the object. These mutexes use the Priority Inheritance Protocol (PIP) to bound the duration of the periods of priority inversion. The group chose to implement this protocol, because it is transparent to the developer (focus on usability).

Every time a mutex locking call leads to blocking, the mutex calls the *Dispatcher*, and it passes the context to the highest priority ready task. When this happens, the mutex saves the calling task in its list of blocked tasks for the purpose of notifying it when the mutex is unlocked. Whenever a task successfully acquires/locks a mutex, it inherits the deadline of the task with the highest priority that is blocked on the same mutex. Since tasks can acquire multiple mutexes, each task as a *map* data structure that maps all acquired mutexes to the priority of its highest priority blocked task.

Locking and unlocking a *mutex* involves a critical region. For this, the Kernel needs to use a global synchronization mechanism. In this case, during *mutex* locking/unlocking,

preemption is disabled at Kernel level. This ensures only one task can enter this critical section at a time.

III. Development Environment

The development environment had to be adapted to work for both Windows in Linux. As such, the group decided to use and modify the files from the *Arduino-Makefile* project [5]. The purpose of the modifications made to the project was mainly to allow for a more organized project structure and add extra *Make* targets for debugging purposes.

SimAVR [6] was a vital part of the development process. SimAVR is an Atmel AVR simulator that allowed the group to run the code without a physical Arduino UNO board. This in turn, enabled the use of the GNU Project Debugger (GDB) to analyze the system state and debug the code.

IV. Evaluation of Results

This section discusses results relating to memory footprint and performance of the Kernel and its components.

A. Memory footprint

When developing for hardware like the Arduino UNO board, the size of the code is of major importance. For this reason, except for debug builds, the Kernel and code is compiled optimizing for size: using the *-Os* GCC compiler flag.

1) Task Control Block size: Each TCB has three parts: the static memory relating to the task, the task's stack (dynamically allocated array), and memory related to each mutex that the task uses (dynamically allocated map). The Kernel uses C++'s static asserts to ensure that the TCB size is the expected one (only counts static memory). Each task uses **28 bytes of static memory**, **4 bytes for each mutex** currently locked by the task, and the task's stack size (configured by the developer).

2) Kernel size: The idle task's (see section II-F1) TCB occupies most of the size of the Kernel. The Kernel uses **13 bytes of static memory**, **2 bytes for each task on the task set**, and one extra TCB for the idle task (see section IV-A1).

3) Code size: When compiled optimizing for size, the Kernel occupies 21.7% of the Arduino's available program data space, and 13.1% of the Arduino's available data space. This allowed the group to register up to 10 tasks without optimizing their stacks size.

B. Performance

This section discusses the time taken to perform various processes of the Kernel. It should be noted that in order to measure these times, the Arduino reported timestamps through serial communication. This introduces some delays in the execution.

1) ISR execution time: The Interrupt Service Routine (ISR) of the Kernel is responsible for starting the *Scheduler* (see section II-E), and preempting the currently running task (if needed). Since this is a tick-based Kernel, the ISR is executed periodically. By default, the Kernel executes 250 ticks per second. Fig.1 shows the evolution of the time taken by the ISR for different numbers of registered tasks.

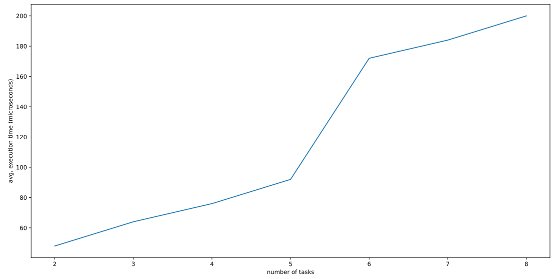

Fig. 1. ISR average execution time

2) Task activation to execution time: The time taken since the activation of a task until it starts executing is affected by various factors, e.g., the existence of higher priority tasks, tasks *yielding*, and the time until the next ISR tick.

In a system with a single task that only *yields*, it takes on **average 4015 microseconds** from the moment it is active to the moment it starts executing. As mentioned in the previous section, ISR runs 250 times per second. This means that it runs every *4000 microseconds*. This fact is coherent with the results found.

3) Context switch time: The context switching time is the time it takes for saving the context of a task and loading the context of another. This happens during preemption. To measure this time, the group used a system with 2 tasks, where one task frequently yields the execution to another. This makes it easier to discard the effect of the scheduler (run on the ISR). On this system, one of the tasks is long-running, and the other is short and fast. On average, this context switching process took **165 microseconds**.

Conclusions

This document describes the implementation of a Real-Time micro-Kernel for the Arduino UNO microcontroller, including experimental results and analysis of memory footprint. The code is open-source, customizable, and made with a focus on ease of development. In the project's repository, developers can find all information needed to compile, test, and upload the code for both Windows and Linux.

With this Kernel, developers can implement Real-Time periodic tasks, with multiple aspects inspired by POSIX (e.g., the prototype of tasks' code). Multiple aspects of the kernel are dynamic, e.g., the maximum number of tasks in the task set. These contribute to the usability of the Kernel, since developers do not need to change know how it works/change parameters in it.

References

[1] Arduino, Arduino UNO R3 Documentation, https://docs.arduino.cc/hardware/uno-rev3

[2] FreeRTOS, freertos.org landing page, https://freertos.org

[3] freertos.org, 'RTOS context switching saving the CPU registers', 2018, https://www.freertos.org/implementation/a00015.html

[4] Giorgio Buttazzo and Paolo Gai, 'Efficient EDF Implementation for Small Embedded Systems', 2006

[5] Sudar Muthu, Arduino-Makefile, https://github.com/sudar/Arduino-Makefile

[6] Michel Pollet, simavr, https://github.com/buserror/simavr