

MST4FreeRTOS - A standard model library for FreeRTOS

E. Alberti

September 2025

Contents

1	Project data	1
2	Project description	2
2.1	Design and Implementation	3
2.1.1	The MST API	3
2.1.2	The MST Configuration File	3
2.1.3	Task TCB	3
2.1.4	Periodic Task Implementation	3
2.1.5	Sporadic Task Implementation	4
2.1.6	Rate Monotonic Scheduling (RMS)	4
2.1.7	Earliest Deadline First (EDF)	4
2.1.8	Sporadic server implementation	4
3	Project outcomes	5
3.1	Concrete outcomes	5
3.2	Learning outcomes	5
3.3	Existing knowledge	5
3.4	Problems encountered	5
4	Honor Pledge	6

1 Project data

- Project supervisor: Tomas Antonio Lopez
- Project delivered by:

Last and first name	Person code	Email address
Alberti Enrico	10712676	enrico1.alberti@mail.polimi.it

- It is possible to find the project repository at: <https://github.com/chiccoalbe/MST4FreeRTOS>

2 Project description

The presented project lies in the domain of **real-time software**, designed to control **real-time systems**.

Real-time systems typically have **safety-critical requirements**, which can be summarized as:

- **Logical correctness**: the produced output is functionally correct.
- **Timing correctness**: the output is produced within the required time constraints.

To guarantee these requirements, it is essential to **model real-time systems**. In such models, we capture key properties and parameters within what we call a **standard model**.

The standard model can be summarized as:

A set of tasks with known worst-case execution times (WCET), release times (periods), and deadlines, scheduled according to an algorithm under resource-sharing rules, with the fundamental property of timing predictability.

In a real-time operating system, the central concept is the **task**:

- A **task** is an abstract entity representing a sequential piece of code.
- A **job** is a single execution instance of a task.

Tasks are characterized by a set of parameters:

- **Requirements**
 - Periodic or sporadic nature
 - T_i : period, D_i : deadline
 - ϕ_i : phase (optional)
- **Metrics**
 - C_i : Worst-Case Execution Time (WCET)

The foundation of this project is **FreeRTOS**, a lightweight real-time operating system designed for microcontrollers. It provides multitasking with priority-based preemptive scheduling, inter-task communication mechanisms (queues, semaphores, mutexes), software timers, and memory management.

The goal of this project is to build a **standard model** on top of FreeRTOS. The result, **MST4FreeRTOS**, is a scheduling library implemented entirely in user space, without modifying the kernel. This approach allows for flexible customization by the user while creating a theoretical framework on top of the existing RTOS, without altering its core behavior.

MST4FreeRTOS provides the following features:

- **Periodic tasks interface**
- **Sporadic tasks interface**
- **Earliest Deadline First (EDF) scheduling**
- **Rate-Monotonic Scheduling (RMS)**
- **Sporadic server to run sporadic jobs**
- **Acceptance tests for sporadic jobs (for both EDF and RMS)**

2.1 Design and Implementation

The library has been developed following the style and conventions of FreeRTOS. It provides an API to create and manage periodic and sporadic tasks, a configuration file to select scheduling policies, and the option to enable or disable the use of a sporadic server.

2.1.1 The MST API

The library exposes a simple API for task creation and management. Examples include:

1: Creating a periodic task

```
1 vMSTPeriodicTaskCreate(MSTTask2, "2",  
2 configMINIMAL_STACK_SIZE,  
3 NULL, 2, &Task1Handle, 1000, 300, 10000, 300);
```

2: Creating a sporadic task

```
1 vMSTSporadicTaskCreate(MSTTask3, "3",  
2 configMINIMAL_STACK_SIZE,  
3 NULL, 1, &Task2Handle, 1,  
4 400, 150);
```

3: Requesting execution of the previously created sporadic task.

```
1 vMSTSporadicTaskRun(&Task2Handle);
```

2.1.2 The MST Configuration File

The library requires a configuration file (*MSTScheduler.h*) where the user specifies system parameters:

- `mst_test_PERIODIC_METHOD`: technique for periodic task management (1 = delays, 2 = software timers).
- `mst_schedSCHEDULING_POLICY`: scheduling policy (default = RMS).
- `mst_USE_SPORADIC_SERVER`: whether to enable the sporadic server.

2.1.3 Task TCB

Each task in FreeRTOS is represented by a Task Control Block (TCB), which stores the information required by the kernel to manage the task. The standard FreeRTOS TCB does not include the parameters needed for MST scheduling. To address this, the MST library extends the TCB using thread-local storage to store requirements and metrics, including *period*, *phase*, *deadline*, *WCET*, and *interarrival time*.

2.1.4 Periodic Task Implementation

Periodic tasks are created with `vMSTPeriodicTaskCreate`, which augments standard FreeRTOS tasks with parameters T , D , ϕ , and C . Releases are handled through delays or software timers, and deadline misses are monitored at runtime.

2.1.5 Sporadic Task Implementation

Sporadic tasks are created with `vmSTSporadicTaskCreate` and are characterized by interarrival time T_{ia} , deadline D , and WCET C . Jobs are released explicitly by the user. If the sporadic server is enabled, each release request is subject to an acceptance test before execution.

2.1.6 Rate Monotonic Scheduling (RMS)

Under RMS, tasks are assigned static priorities based on their deadlines at scheduler start-up. In case of sporadic server usage, an acceptance test for sporadic jobs is ran. This is done by calculating if the slack $\sigma(t) \geq 0$

$$\sigma(t) = \left\lfloor \frac{d_{s1} - t}{p_s} \right\rfloor e_s - e_{s1} \geq 0$$

where d_{s1} is the job deadline, t the release time, p_s the server period, e_s the server budget, and e_{s1} the job execution time.

2.1.7 Earliest Deadline First (EDF)

EDF scheduling dynamically orders tasks by their deadlines. An acceptance test is performed for each sporadic job to ensure schedulability:

$$\frac{e}{d - t} + \Delta_{s,k} \leq 1 - \Delta$$

The deadlines partition the time from t to \inf into $n + 1$ discrete intervals: I_1, I_2, \dots, I_{n+1} .

e is the execution time, d the absolute deadline, t the release time, $\Delta_{s,k}$ the density of interval I_k , and Δ the total system density.

2.1.8 Sporadic server implementation

A sporadic server is a type of bandwidth-preserving server used to handle sporadic and aperiodic tasks. Its purpose is to guarantee responsiveness for irregular jobs while ensuring that periodic tasks remain schedulable. The key property is that it never demands more processor time than a periodic task with the same period and budget would.

It is implemented as a special periodic task (T_{ss}, C_{ss}) , with:

$$T_{ss} = \frac{C_{ss}}{U_{ss}}, \quad U_{ss} = m \left(2^{1/m} - 1 \right) - \sum_i \frac{C_i}{T_i}$$

where m is the number of periodic tasks, and $\frac{C_i}{T_i}$ the utilization of task i .

The sporadic server runs with period T_{ss} and WCET U_{ss} .

It keeps an EDF-ordered list, the sporadic tasks queue. Before running it runs the before-mentioned RMS acceptance test.

A notification system between the server, a budget depletion watchdog timer and the running task makes the server work dynamically.

3 Project outcomes

3.1 Concrete outcomes

The result of the work is a library which can be used with any version of FreeRTOS.

A sample project for a STM32F4 microcontroller is also available. The files and more detailed description of the project can be found at the repository: <https://github.com/chiccoalbe/MST4FreeRTOS>.

The current work is to be seen as a basis for more detailed future work. In this current version there are many possible points of optimization and many critical sections need to be more thoroughly tested.

An alpha version of this library has been used to port a bare-metal version of the real-time software for a Battery Management System (BMS) into a OS-based version. This has been tested on actual hardware.

When the first stable version of this library will be ready, this software will be installed permanently to control an actual battery.

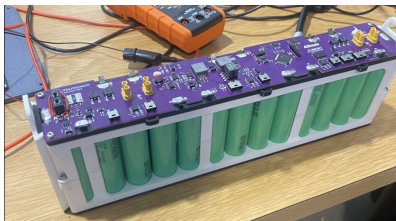


Figure 1: 13s2p Li-ion battery with BMS running an early version of MST4FreeRTOS

3.2 Learning outcomes

- Learned to identify and analyze the requirements for modeling real-time software systems. This knowledge is particularly valuable for my work with real-time systems, as understanding how an operating system ensures deterministic timing behavior greatly improves my ability to design and evaluate such systems.
- Learned the internal workings of FreeRTOS. Which inspired a better way to handle compile-time customization. While working I found a possible flow in the kernel, in particular on initialization of thread-local storage pointers. The issue and proposed solution can be found here: <https://forums.freertos.org/t/possible-non-restricting-feature-22073>

3.3 Existing knowledge

Courses useful for the project:

- AOS
- Embedded systems

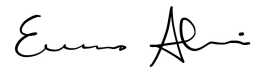
3.4 Problems encountered

- The main problem was to be able to exactly monitor timings. Without touching the kernel it has been necessary to force the user to setup internal timers for run-time stats.
- Many sections were not thread-safe at initial implementation, leading to race conditions, usually when sporadic jobs were inserted concurrently.

4 Honor Pledge

I pledge that this work was fully and wholly completed within the criteria established for academic integrity by Politecnico di Milano (Code of Ethics and Conduct) and represents my/our original production, unless otherwise cited.

I also understand that this project, if successfully graded, will fulfill part B requirement of the Advanced Operating System course and that it will be considered valid up until the AOS exam of Sept. 2025.

A handwritten signature in black ink, appearing to read 'Enrico Alberti', with a stylized flourish at the end.

Enrico Alberti