SOTR Final Project Report
Task Management Framework for FreeRTOS

Pedro Gonçalves - 88859
Pedro Silva - 89228

1 Implemented functionalities

Our framework, TMAN, is a task management service for FreeRTOS, which simplifies the creation and implementation of tasks in a user-friendly way. The first thing the user has to do is initializing the framework with a fixed number of ticks, meaning that if we choose 10, each TMAN tick corresponds to 10 FreeRTOS ticks (10 * 1 ms).
Afterwards the user can start creating tasks and adding attributes to each task, such as period, phase, deadline and precedences. When all the intended tasks are created, the user can then initialize them and watch the execution times and the order by which the tasks are executed. It is also possible to check some statistics every n tick, where n is defined by the user, and also close the framework when it is no longer necessary.

2 Used data structures

We created 2 structures in this project: TASK and TMAN.
The TASK structure has all the variables that describe a task created by the user, such as: period, deadline, phase, name and prec (indicates the precedence). TASK also has statistical variables like: activations and deadlines_missed that track each occurrence. It also has a few control variables: prec_executed, executed_once, id, prec_signal, state. These variables are used to monitor things like the number of occurrences and to store important information about the task that will be controlled on the scheduler.
The TMAN structure has 3 variables: taskId, maxTasks and a TASK array. Their main goal is to store the data inserted by the user when he is creating the tasks.
We also have a few global variables to help the synchronization, such as: tick (store the number of PIC32 ticks that form a TMAN_tick), tman_tick (counter for TMAN_ticks), started_tasks (counter for the tasks initialized), close_tick (store the number of ticks to call the TMAN_Close()), stats_tick (store the number of ticks to call TMAN_Stats()), TASK t1 (stores all information of the active tasks), TaskHandle_t xHandle (array with a TaskHandle_t to suspend/resume each task) and QueueHandle_t xQueue1 (to control the print flow).
When a user creates a task, the task is stored in TMAN and when he starts the task, its information will be stored in an index at the global variable TASK t1.

3 Auxiliary tasks

To control the framework, we had to write a few auxiliary tasks. These tasks are initialized along with the TMAN. Those tasks are the following:

- **scheduler**

The scheduler never stops running and has always the highest priority. It counts the number of TMAN_ticks using the vTaskDelayUntil function. This task also schedules all the other started tasks, using the global tasks array t1 to do so, suspending and resuming them when needed. When the tasks are created, they are suspended right away. Then, depending on their periods, phases and precedences, they are resumed. This is done with a loop that iterates through all the started tasks and verifies their state.

- **printing_queue**

When the tasks had the same phase and same priority, sometimes the prints were illegible because different tasks were printing characters at the same time. To avoid this, when a task is running, instead of printing the Task name and TMAN_ticks, we allocate those characters in a buffer with sprintf() and insert the buffer in the global queue xQueue1. This way, there wouldn't be conflicts because when some task is writing in the queue, the others have to wait.
After that, we needed the printing_queue task to pop and print the values from the queue. This task has the lowest priority possible because it can be executed at any time, because the values on the queue have the correct TMAN_ticks.

4 API

Let's now dive into the usage of each method of our API.

- **TMAN_Init(int ticks_tman)**

The first method the user has to initialize is the **TMAN_Init**. This method creates the framework as well as two tasks, the **scheduler** and the **printing_queue**. It takes the TMAN ticks as argument, which corresponds to a multiple of FreeRTOS ticks, since the FreeRTOS tick is nearly 1 ms, 10 TMAN ticks would correspond to 10ms. If the user wants to observe the tasks execution times in real time, he can set the ticks to 1000 so that each task will execute every second, which makes it easier to analyze if everything is correct.

- **TMAN_TaskAdd(TMAN t, char * name)**

The second method is the **TMAN_TaskAdd**, this method is responsible for adding the task to the Tasks array, for attributing it an ID, which is the next ID available and incrementing the ID's number. It receives the TMAN framework and the name of the task as arguments.
It's worth mentioning this method adds a task with the default configuration (period, phase and deadline equal to -1), if the user wants to configure a task, that's described on the next method.

- **TMAN_TaskRegisterAttributes(TMAN t, int period, int phase, int deadline, char *namec, char *prec_task)**

In this third method is where the tasks configuration is made. It receives the TMAN framework, the period, phase, deadline, name and precedences of a task as arguments. The period, phase and deadline parameters are all defined in TMAN ticks, the namec and prec_task are both an array of characters, the first corresponds to the name of the task we are configuring and the latter the name of the task that has precedence.
This means that if we implement this method in the following way **TMAN_TaskRegisterAttributes(tm, 4, 3, 5, "A", "B")**, we are attributing a period of 4 ticks, phase of 3 and deadline of 5 to the task “A”. We are also saying that the task “B” has precedence, which means the task “A” has to wait for the task “B” to finish executing for it to execute. All of this in the framework tm.
Moving on to the implementation of this method, we start by comparing the period of the task with its deadline and making sure the period isn’t bigger than the deadline, otherwise the system is not feasible and the program will close.
Then, if the task has precedences, we check which task precedes it and set the prec_signal to 1 and the prec to the name of the task that precedes it. If the task that precedes it has a higher period, the first task inherits the period and phase of the second.

- **TMAN_TaskStart(TMAN t, char *name)**

This method receives the TMAN framework and task name as arguments, and then increments the started_tasks variable. It also adds the started task to the global tasks array t1 to be scheduled.

- **TMAN_TaskWaitPeriod(int tar)**

After doing some computacional work and inserting the buffer in the queue, a task calls this function to suspend itself. Receives the task id as an argument to change the task state in the global tasks array. It increments the statistical counter activations and changes some tasks variables like prec_executed and executed_once.

- **TMAN_Close(int tick_c) and TMAN_CloseInternal()**

These two functions are responsible for the closing of the TMAN framework. The first function, **TMAN_Close,** is used by the user and receives the number of ticks the user wants to close the program on. The second function **TMAN_CloseInternal** is called internally and is responsible for the deletion of the tasks and the closing of the framework.

- **TMAN_TaskStats(int tick_s) and TMAN_TaskStatsInternal(int task_id)**

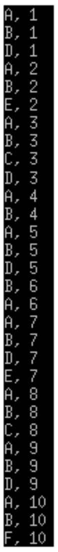
Finally, the last two tasks are responsible for printing out some statistics of the tasks, like the total activations and deadlines missed. The user only has to call the TMAN_TaskStats() and introduce the interval, in TMAN ticks, that he wants those statistics to be printed. This method is optional so the user can opt for not using it if he has no interest.

5 Results

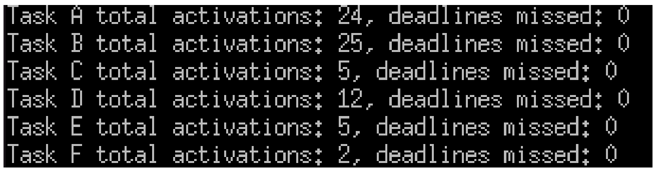
The results for the following configuration:

```
tm = TMAN_TaskRegisterAttributes(tm, 1, 0, 400, "A", "");
tm = TMAN_TaskRegisterAttributes(tm, 1, 0, 400, "B", "");
tm = TMAN_TaskRegisterAttributes(tm, 2, 1, 400, "D", "");
tm = TMAN_TaskRegisterAttributes(tm, 5, 2, 400, "E", "");
tm = TMAN_TaskRegisterAttributes(tm, 2, 0, 400, "C", "E");
tm = TMAN_TaskRegisterAttributes(tm, 10, 0, 400, "F", "");
```

Were as follow:



```
A, 1
B, 1
D, 1
A, 2
B, 2
E, 2
A, 3
B, 3
C, 3
D, 3
A, 4
B, 4
A, 5
B, 5
D, 5
B, 6
A, 6
A, 7
B, 7
D, 7
E, 7
A, 8
B, 8
C, 8
A, 9
B, 9
D, 9
A, 10
B, 10
F, 10
```



```
Task A total activations: 24, deadlines missed: 0
Task B total activations: 25, deadlines missed: 0
Task C total activations: 5, deadlines missed: 0
Task D total activations: 12, deadlines missed: 0
Task E total activations: 5, deadlines missed: 0
Task F total activations: 2, deadlines missed: 0
```

Fig 1 & 2. Results for the configuration above

Our results were what we expected, however, sometimes the priorities are not exactly working as they should, which makes some tasks that have lower priority execute before a task that has a higher one. We couldn’t understand the reason why the priorities are not working the intended way.

6 Notes

It’s worth mentioning that we changed the order of the creation of the tasks, since we could only attribute the precedence to the task “C” after the task “E” was created. This explains why the results are now more accurate than the previous ones.

```
tm = TMAN_TaskRegisterAttributes(tm, 5, 2, 400, "E", "");
tm = TMAN_TaskRegisterAttributes(tm, 2, 0, 400, "C", "E");
```

7 Extras

We also implemented a way to detect deadline misses, by checking if a given task had executed before its deadline, also taking into consideration its phase.