ES6153 Project: Phase 1

Gao Yiyi and Michele Xiloyannis

Abstract— **In the present manuscript we describe our ideas and procedure to implement an Application Programming Interface (API) for time-triggered tasks in μC/OS–III. The API uses the system's Tick Task as a timer to dispatch the periodic periodic tasks to the ready-list when their period expires. Using a Fibonacci Heap for recursion and an AVL Tree for an RM scheduler allows for a more efficient implementation.**

I. INTRODUCTION

Hard real-time systems are characterized by the need for both functional and temporal correctness. Such systems not only have to produce appropriate outputs (functional correctness), but also within the specified time constraints (temporal correctness). The design process of hard real-time applications usually involves splitting of the required functionality into separate tasks. Each task is responsible for only a portion of the functionality to be handled. For example, consider a typical automotive engine control system, in which some of the hard real-time functionality such as control of air/fuel ratio, idle speed of the engine and valve timing are implemented as separate tasks. From the OS perspective, a task can be considered as a function with a priority value, requiring a set of registers, its own stack area as well as heap to store some data. From the scheduler perspective, a task is a series of consecutive jobs with regular ready time. The applications such as an engine control software executes different kinds of computational activities like control or monitoring a sensor. Control tasks such as valve opening and closing are triggered at predefined rotation angles of the crankshaft (event-triggered tasks) whereas other tasks such as temperature monitoring are activated by a timer at fixed time intervals (time-triggered tasks). Hence, it is important for a Real Time Operating System (RTOS) such as FreeRTOS, Micrium/OS-III to support both time-triggered and event-triggered tasks.
For this purpose, we wrote an API to support periodic tasks in micrium. The API comprises a function to create recursive tasks, one to delete them and one for releasing the task synchronously. The following sections detail how the recursion was achieved, which data structures used to store the periodic task's TCBs and dispatch them in a more efficient way and the implementation of a Rate Monotonic scheduler (RM).

II. PERIODIC TASKS

The API for periodic tasks comprises a function *OSRecCreateTask* to create tasks that is structured as follows:
it creates a task using the existing API; it initialises a node in the Data structure for recursion containing the following information: a pointer to the TCB, a *PreviousDispatch* field and a *NextDispatch* field. The *PreviousDispatch* is initialised to 0 and the *NextDispatch* is initialised to OSTimeGet()+ the user's predefined period. The node is inserted in the data structure for recursion.
Periodicity is achieved by dispatching each task to the ready-list every time OS_TickCtr reaches the *NextDispatch* field in a node of the recursive structure. This check is done in the OS_TickListUpdate() function, which is recursively called by the TickTask(). The OS_TickTask() is a periodic task that waits for signals from the tick ISR, generated by a hardware timer, running at at a rate of OS_CFG_TICK_RATE_HZ, which we left to 1000Hz.
Deleting a recursive task similarly calls the standard OS_TaskDelete() function from the OS's API and removes it from the data structure for recursion.
Synchronization is achieved by dispatching all tasks to the ready list, resetting the OS_TickCtr to 0 and setting the *NextDispatch* field of each node to the user's specified value for the period.

```
OS_TickListUpdate(){
        •
        •
        •
        Search the Heap(
        if OS_TickCtr == node➔nextDispatch
              Dispatch Task
              Update nextDispatch = OS_TickCtr + period
        )
        •
        •
        •
}
```

Fig. 1: Pseudocode on how we achieved periodicity for the tasks using the Tick Task Timer and the given data structure.

III. DATA STRUCTURE FOR RECURSION

To handle the tasks created by the user, check when their period expires and dispatch them to the ready-list we used a Fibonacci heap. Fibonacci heaps are a collection of trees satisfying the minimum-heap property, that is, the key of a

child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root of one of the trees. Thanks to this property, the Fibonacci heap is ideal to find the task with minimum remaining time. The find minimum function takes, as a matter of fact, a constant time.

Each node in our Fibonacci heap contains a pointer to a Task's Control Block (TCB), and two fields, one storing the previous dispatch time *PreviousDispatch* and one storing the next dispatch time *NextDispatch*. These are used and updated when checking if the task needs to be despatched: if the OS Tick Counter is equal to the next dispatch time the task is sent to the ready list, the *PreviousDispatch* time is set to the current tick time and the *NextDispatch* time is set to the previous dispatch + the user's specified period.

We thus implemented a Fibonacci head and a set of functions to perform: insert, delete minimum, decrease key, unite trees and delete node.

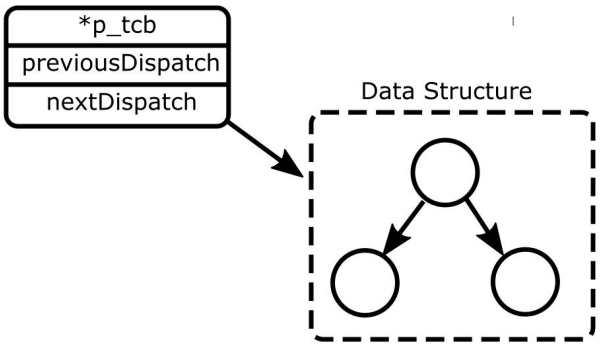

Fig. 2: Nodes fields of the data structure for recursion. Each node contains a pointer to the task's TCB and two values tracking the previous and next instance of the task in time.

IV. SCHEDULER

We implemented a Rate Monotonic scheduler by simply setting the priority of a task depending on the user-specified time period. The smaller the period the higher the priority, i.e. the smaller the specified OS_PRIO field of the task. We make sure that priority OS_CFG_PRIO_MAX-1 = 19, reserved for the idle task, is never assigned and that the maximum priority is left to the tick task, which we set at 3. The range of priorities could thus be assigned between 4 and 18.

V. DATA STRUCTURE FOR THE SCHEDULER

AVL Trees are self-balanced binary trees structured so that the heights of the two child subtrees of any node differ by at most one; if such property is not met, the tree is rebalanced. Lookup, insertion, and deletion all take $O(log(n))$, with n being the number of nodes in the tree. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations. AVL Trees are, thus, very convenient for lookup operations.

The AVL tree was used as ready-list.

VI. HANDLING THE TICK COUNTER OVERFLOW

μC/OS-III source's code has no way of handling overflows of counting variables. We have included a very simple function to reset the value of a counting variable when it overflows. This is done for both the OS_TICKCTR an the field in the recursive data structure that keep track of the previous and next release of each task.

For the OS_TICKCTR, we have included a check in the OS_TickListUpdate(): if the counter reaches its max value, i.e. $256^{sizeof(OS\_TICK)} - 1$, we reset it and keep track of the number of overflows via a separate variable.

When this occurs we also update the *nextDispatch* field in the nodes of the recursive structure to be: $nextTick = nextTick - OS\_TICKCTR$. This allows the tasks to continue being released recursively even when the OS_TICKCTR is reset.

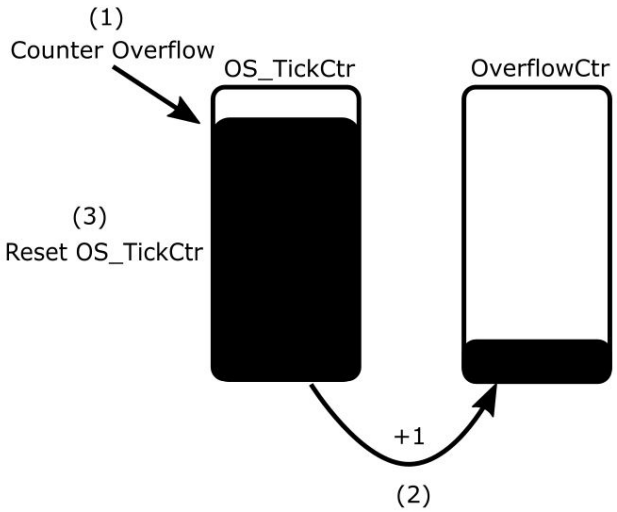

Fig. 3: Handling the overflow of the tick counter and the time-keeping variables in the recursive data structure. (1) When and overflow is detected a new OS_Tick variable is increased (2) and the Tick counter is reset (3).

VII. BENCHMARKING