Hartstone Hard Real Time Benchmark
Test Series Implementation

Daniel Casini, Emiliano Palermiti, Matteo Pampana

Abstract This document presents the implementations of the PH series under the two different operating systems: FreeRTOS and ERIKA. In the first part, the implementation of the Whetstone code is discussed since it is shared among the to different implementation and can be reused. In the last part the two different implementations are explained, considering the unsolved problem of handling the deadline management.

1 Whetstone

The whetstone module is implemented following the Whetstone-Benchmark code being compliant to the benchmark specification. In order to avoid that additional function calls not provided by the benchmark code can invalidate the results, the Whetstone code is implemented as a macro that takes as argument the workload in Kilo-Whetstone-Per-Period (KWPP). Calling this macro periodically the final result is that the task itself generates the proper periodic task load.

2 FreeRTOS Implementation

As it has already discussed, FreeRTOS does not directly support the implementation of period tasks, but it provides the basic functions to do that.
Recalling the problem of the deadline management it is necessary to provide the support for this issue. In order to solve this problem, a set of macro has been developed in order to check, record and eventually skipped the task deadlines. To avoid that benchmark is corrupted due to the interference of the benchmark management, all the task variables used to store the test results are defined as global variables and no additional function calls are introduced.
As far as the operating system is concerned, it is possible to dynamically create and delete task at run time, without any side effect. This feature is very useful for the implementation, because it is possible to dynamically create the task set used within the specific test, during a certain experiment. Under this assumption, the implementation is structured with a main task which purpose, is only to manage the evolution of the benchmark, according to the specifications. This task is responsible to generate, delete and send the results of a certain task set, belonging to a test within each experiment. The management task works at highest priority, in such a way that, every time a test needs to run, after the task set is completely created and configured with the right parameters, the management task simply go to the sleep state for the test duration. As soon as it wakes up, the tasks composing the benchmark, will automatically stop their execution due to fact that they run at a lower priority. Therefore, since they are stopped, they can be deleted by the management task, that at this point, it can send the results and then creates the following task set and so on, until the series is completed.

2.1 Periodic Task

In this section the basic concept on how the periodic tasks are implemented, is described. FreeRTOS does not support directly periodic tasks but it offers a set of functions that allows programmers to generate a periodic behaviour. These functions cannot be used as it is because this benchmark requires a deadline miss check-

ing while FreeRTOS does not provide directly any mechanism of this type. For these reasons an additional work needs to done. The deadline miss checking and the other mechanisms for which the task suspend itself if a deadline miss is checked are implemented by the following macro called: *INIT PERIODIC*, *START PERIODIC* and *WAIT FOR NEXT PERIOD*. Using this set of macros the task body needs to be something similar to the following example.

```
void vGenericTask( void * pvParameters )
{
        INIT_PERIODIC()

        START_PERIODIC()

        WHETSTONE_CODE(load)

        WAIT_FOR_NEXT_PERIOD()
}
```

At a first glance could be intuitive to declare inside the task body a set variable to store the task state, such as the number of deadline miss, the response time for the current job and so on, but as it has already been discussed, in order to prevent that these variables interfere with the benchmark itself, they are held by a set of arrays declared as global. In each array, in the position i there will be the specific variable belonging to the i-th task, therefore there will be an array for each and every state variable needed. The last step is to make the task aware of its index, in order to access the right position inside each array. This requirement can be reached by means of the mechanism provided by FreeRTOS that allows to pass arguments to tasks, hence it is only necessary to pass to each task an integer that represents its index.

Lets analyse in detail each and every macro previously presented. The *INIT PERIODIC* is responsible to initialize all the state variables. The *START PERIODIC* macro is simply the start statement of an infinite loop that will be closed by the *WAIT FOR NEXT PERIOD* macro that is the one that starting from the response time evaluation, checks if the task job is completed within the deadline. Finally this last macro calculates the amount of time for which the task needs to sleep, in order to behaves as a periodic benchmark task.

3 Erika

Erika Enterprise natively supports the implementation of periodic tasks by means of *Alarms*. An Alarm is an entity connected to a counter that can be set in order to activate a task when the time limit associates with expires.

As far as the deadline management problem is concerned the solution is a little bit tricky, Erika does not provide a direct handling of deadline misses. Remembering the fact that an expiration of an alarm corresponds to the ending of the current period and the starting of the new one, we can assume that if a task - that can be activated only once - is already running when the alarm expires, then is experiencing a deadline miss. Thinking in general, this way of catching deadline misses is limiting when the deadline is not equal to the period of the task, but for our benchmark is what is needed.
In order to catch the alarm expiration Erika provides a list of ErrorHook that fits perfectly the problem.

In contrast to FreeRTOS, Erika does not provide dynamic creation of tasks at run time. Erika is an OSEK-compliant RTOS so the task set must be specified into a descriptor file, called OIL file, statically.

The task set is composed by a SuperTask that has highest priority which manages tests and experiments: it stops the previous test, check if the experiment ended, sends results, computes new test's characteristics, starts the new test. Another tricky part of this implementation is how to stop the execution of the current running tasks in between two different tests. In fact if a task needs to run more when the SuperTask preempts it, we can collect some unwanted "out of time" misses. In order to avoid this behaviour it is important to kill those tasks. In Erika tasks cannot kill each other. So, we decided to collect misses only if a certain global flag is equal to 0 then when the SuperTasks activates, by means of its *PreTaskHook*, this variable is set to 1 and the alarms linked to the tasks are cancelled, in order to prevent further "bad" activations of the tasks.