

Dealing with Task Overruns

Standard scheduling works well when a processor is moderately loaded but may fail if the processor becomes overloaded. When a task is required to perform extra processing and takes longer than normal to execute, it may be scheduled to execute before a previous instance of the same task has completed. The result is a *task overrun*.

The most recent versions of the Embedded Target for Motorola® MPC555 and the Embedded Target for Infineon C166® Microcontrollers include a task scheduler that handles temporary task overruns. You can configure this scheduler to meet application-specific requirements. This article explains the basic principles of task scheduling and shows how these apply to the enhanced task scheduler in the Embedded Targets products. The enhanced scheduler can also be applied to any Real-Time Workshop® target, including custom targets developed by third parties.

Standard Task Scheduling

Real-Time Workshop includes three types of task scheduling: single-rate, multirate single tasking, and multirate multitasking.

Single-rate

In single-rate scheduling, every component in the model runs at a constant sample rate in a single task (Figure 1). As long as all the elements of the model can be executed before the next time step, no scheduling problems occur.

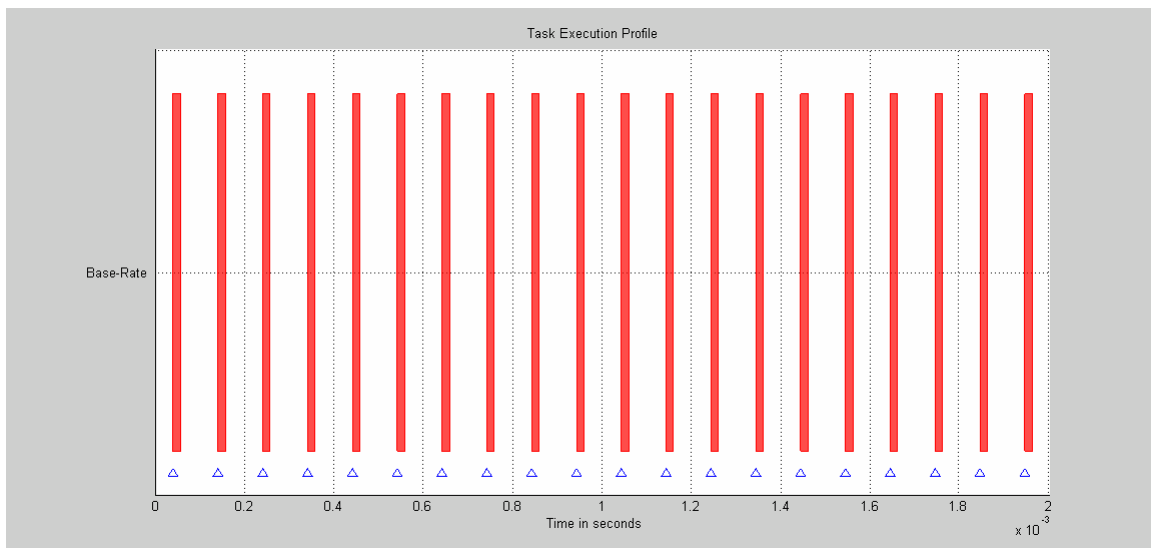


Figure 1 -- A single-rate execution scheme.

Multirate single tasking

When multiple rates are present, different portions of the code must be run at different sample rates. The simplest, but not the most efficient, approach for executing a multirate

Dealing with Task Overruns

system is to use a single-tasking scheduler. Figure 2, for example, shows a model with three separate sample rates executing in a single task. The process execution associated with the base task rate is executed at every timer interrupt. However, operations scheduled for sub-rates execute every fourth and sixteenth timer interrupt, respectively. For example, the task execution that begins at approximately 0.0075 seconds executes the code associated with all three sample rates, while every other fourth task execution processes the fastest two sample rate calculations.

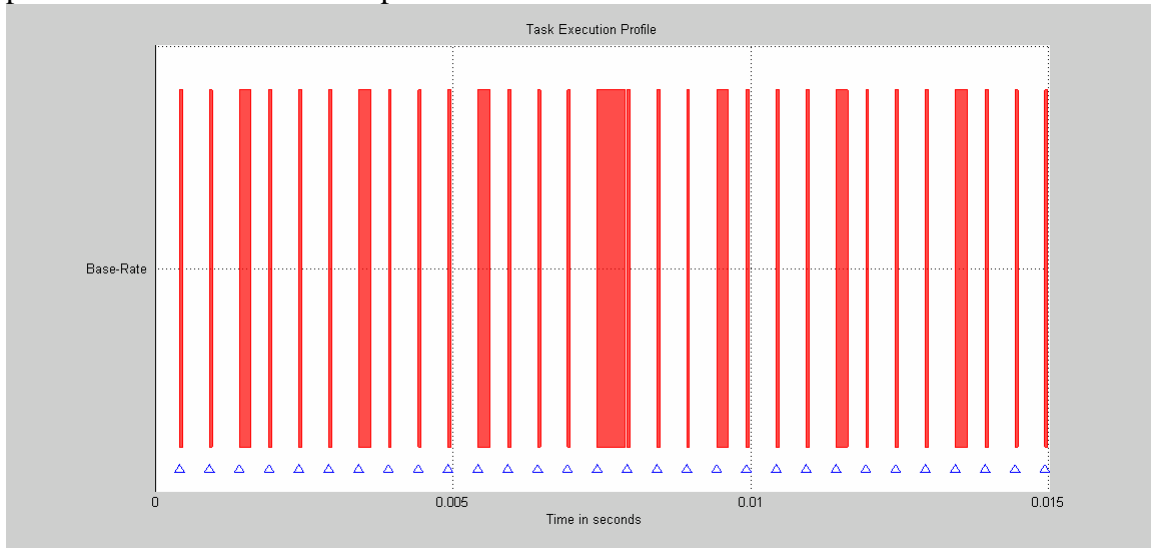


Figure 2 – A multirate, single-tasking execution scheme.

In this situation, the multirate single-tasking scheduler executes in the manner illustrated in Figure 1 but enables multiple rates by selective execution, as shown in the following pseudo code example.

```
main_task() {  
    execute fastest blocks  
    if (time == slow rate sample hit)  
    {  
        execute slower blocks  
    }  
}
```

Multirate multitasking

A priority-based multitasking scheduler is the most efficient method for executing a multirate system. A multitasking scheduler may be based on a real-time operating system or implemented directly as a timer interrupt service routine for your target hardware. For optimal execution, the separate tasks are prioritized in rate-monotonic order, from the fastest tasks to the slowest (Figure 3). A multitasking scheduler allows a lower priority (longer sample-period) task to be preempted by a higher priority task.

Dealing with Task Overruns

The grey areas in Figure 3 correspond to task preemption, meaning that task execution is suspended by the interrupt request of a higher priority task.

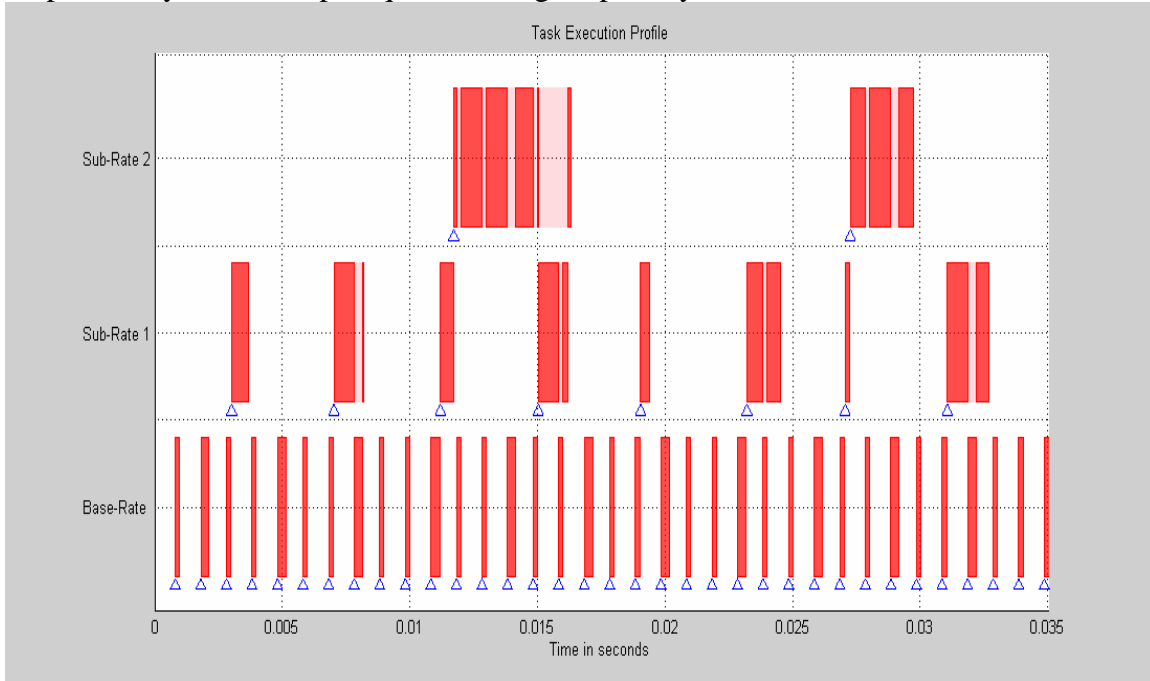


Figure 3 -- A priority-based multirate, multi-tasking execution scheme.

This process occurs on multiple levels. Figure 3 shows an instance in which the lowest priority task waits on the mid-priority task, which in turn is waiting on the highest priority task.

For more information, consult the Models with Multiple Sample Rates section of the Real-Time Workshop User Guide.

The Enhanced Scheduler in Action

The standard scheduler just described handles even a single task overrun as an error and terminates the application. The most recent version of the Embedded Target for Motorola® MPC555 and the Embedded Target for Infineon C166® Microcontrollers include an improved scheduler designed to allow temporary task overruns. This scheduler allows tasks to occasionally overrun their allotted execution time by delaying the start of the next task. It does not, however, allow the gross overruns that occur when the sample rate is too fast for the average task execution time. In this situation, you must use a longer sample period.

The operation of the temporary-overruns scheduler is best described through an example. Figure 4 illustrates a single task that takes longer than expected on a heavily loaded processor. An overload occurs, perhaps owing to an infrequent asynchronous event. The disruption in regularly timed interrupts, denoted by triangles, indicate that the execution during this overload did not complete until after the next scheduled start time of the task. In this case, the task executes immediately after completion in an attempt to catch up with

Dealing with Task Overruns

the scheduled timer events. The time steps immediately after the overload are affected, but execution has recovered within two time steps.

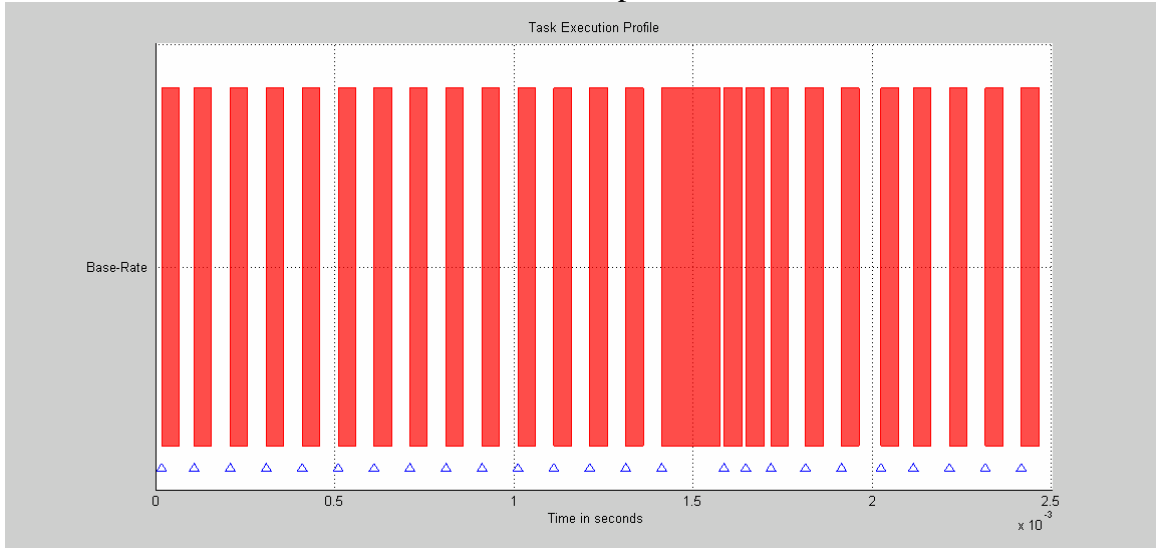


Figure 4 -- An asynchronous single-task overrun.

With a multitasking system, the problem is more complicated. Because the scheduler must maintain priority ordering, lower-priority tasks are the first to be delayed. Figure 5 illustrates a situation in which the second-highest priority is taking too long to execute.

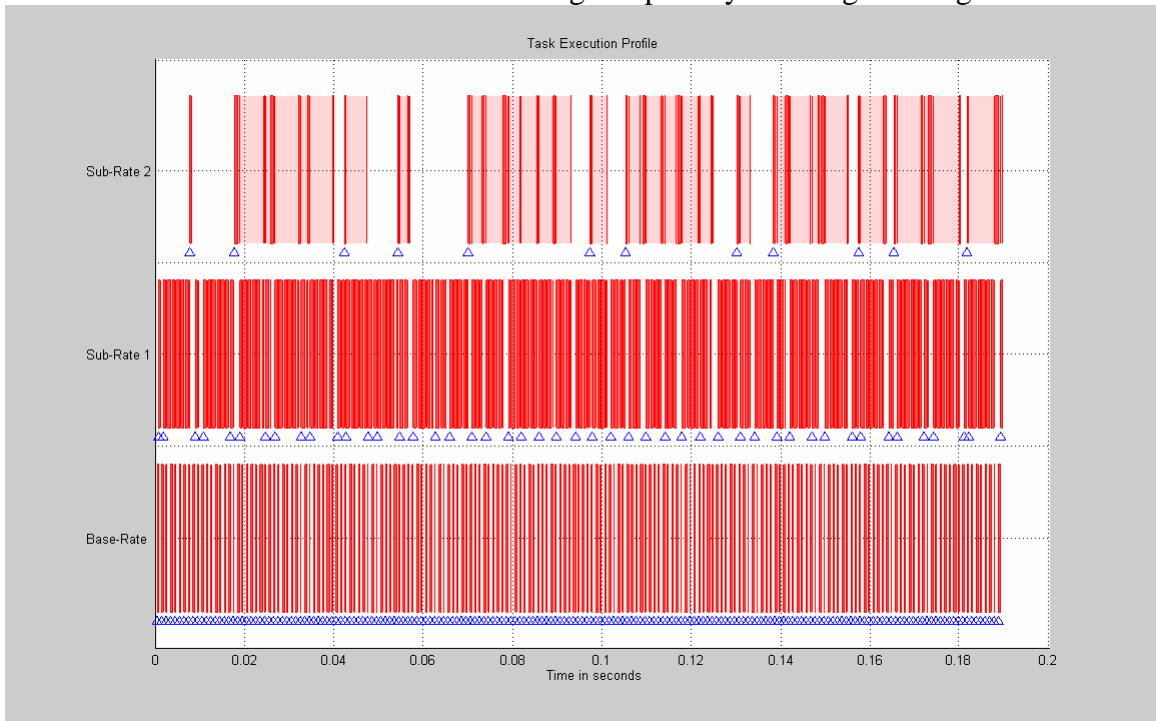


Figure 5 -- Task overrun occurring on a multitask system.

The lower-priority task execution is periodically delayed by the processor loading and by task overruns in the mid-priority task. If overruns are allowed for this priority level, each delayed iteration will be queued and eventually run. Note that this example shows a

Dealing with Task Overruns

heavily loaded processor. Although the temporary overruns scheduler can accommodate large peak loads well, it cannot compensate for an average load that is too high. Again, in this case a larger sample period is recommended.

Behavior of Inter-rate Communications During Overruns

Rate-transition blocks that are configured to ensure deterministic data transfer between tasks may be affected by overruns. If overruns occur only in the base-rate task, there is no effect on the deterministic data transfer between tasks.

If sub-rate overruns occur, however, deterministic data transfer is not guaranteed. In these circumstances, the behavior of the model in real time may differ from its behavior during simulation. This is because, if task-rate overruns occur, loss of synchronization of data between tasks allows “old” data to be communicated between tasks. For the Embedded Target products, you can specify whether base rate overruns or sub-rate overruns should be allowed. Non-deterministic data transfer is illustrated by models that are included with Embedded Target product demos, `c166_multitasking` and `mpc555rt_multitasking`. To ensure that data transfer between tasks is deterministic, you must allow only base rate overruns.

Implementing the Temporary Overruns Scheduler

The pseudo code for a temporary overruns scheduler implementation is as follows.

Dealing with Task Overruns

```
disable interrupts

increment base_rate_overrun_counter
if base_overrun_counter > 3
    error: too many concurrent base-rate overruns
end

if base_rate_overrun_counter > 1
    return
end

while base_rate_overrun_counter != 0
    enable interrupts
    set event flags for each sub-task that is hit
    run base-rate task
    disable interrupts
    decrement base_rate_overrun_counter
    for i = 1 to number of sub-tasks
        if hit for sub-task i
            increment sub_rate_overrun_counter[i]
            if sub_rate_overrun_counter[i] > 3
                error: too many concurrent sub-rate
                overruns
            end
        end
    end
end

%% sub-rate tasks
for i = 1 to number of sub-tasks
    if sub_task_running[i] == true
        return
    else
        while sub_rate_overrun_counter[i] != 0
            set running_sub_task[i] = true
            enable interrupts
            run the sub-rate task
            disable interrupts
            set running_sub_task[i] = false
            decrement sub_rate_overrun_counter[i]
        end
    end
end

end
```

When this algorithm is placed inside an interrupt handler, interrupts are enabled inside the function, which allows it to be pre-empted by new timer interrupts. If another interrupt occurs while the base rate is running, the base count will be incremented and the interrupt handler will return to the interrupted base rate. When this interrupted base rate has completed its run, the scheduler runs the new rate.

Dealing with Task Overruns

If a base-rate interrupt occurs while a non-base-rate task is running, the function will not return but will run the base rate and then update the task counters to see if any subtasks have been activated. It will then continue to the 'for each task' section of the handler, which will run each sub-task, starting with the highest priority. This process will continue until the priority corresponding to the interrupted rate occurs. The function will then return and continue with the interrupted rate.

This behavior preserves monotonicity in all conditions. It also results in the possibility that multiple versions of this pseudo code can be running concurrently, however, and thus the number of stack frames could equal the number of rates plus one. Multiple invocations of a given rate will not occur at the same time, but will be queued.

The italicized pseudo code is not represented explicitly in the actual code; it is inherent in the operation of the interrupts used by the functions implementing the scheduler.

This algorithm can be implemented in a number of different ways. The Embedded Target for Infineon C166® Microcontrollers implements the algorithm as an auto-generated main, with separate interrupts for each task. These interrupts are actually TRAP instructions issued by a scheduler kernel written in assembler. This means that the implementation for C166 differs somewhat from the pseudo code, but the behavior is the same.

The Embedded Target for Motorola MPC555 implements the algorithm as a static main function, which calls the tasks as functions. This implementation requires no assembler and is written entirely in C. Consequently, it is simpler to understand and more appropriate for the timers and interrupts of the MPC555. In this case, the commented code in previous listing is not present.

You can obtain code samples for both these targets by building the demo models provided with the Embedded Target products. You can simply generate the code to examine the scheduler, but to accurately assess the scheduler's behavior, it is best to compile and run the code on the processors.

References

- [1] Erkkinen, T. "High-Integrity Production Code Generation," *AIAA GN&C*, 2002.
- [2] Janka, R.S., "Specification and Design Methodology for Real-Time Embedded Systems," Springer, 2002, ISBN: 0-7923-7626-9.
- [3] Balarin, F.; Lavagno, L.; Murthy, P.; Sangiovanni-Vincentelli, A.; Systems, C.D.; Sangiovanni-, A. "Scheduling for embedded real-time systems," *IEEE Design & Test of Computers*, vol.15, (no.1), IEEE, Jan.-March 1998. p.71-82.
- [4] Sha, L.; Rajkumar, R.; Sathaye, S.S. "Generalized rate-monotonic scheduling theory: a framework for developing real-time systems," *Proceedings of the IEEE*, vol.82, (no.1), Jan. 1994. p.68-82.

Dealing with Task Overruns

- [5] Liu, C.L.; Layland, J.W. "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the Association for Computing Machinery*, vol.20, (no.1), Jan. 1973. Pages 46-61 provide some general RTOS references.

NOTE: The plots shown in this paper were generated with the task execution profiling utility that is provided with the Embedded Target for Infineon C166 Microcontrollers and Embedded Target for Motorola MPC555.