9.3 Lookup tables again

10.1 Timer overrun

Earlier sections assumed actions completed in a short non-zero time that was less than the synchSM period. For systems with extensive actions in a state, or with numerous tasks sharing the same microcontroller, the assumption may not hold. An embedded systems programmer may have to pay attention to the actual execution duration of actions relative to synchSM periods to ensure the system utilizes a microcontroller appropriately.

In an approach to scheduling synchSMs in C, the scheduler exists in a timer ISR that is called periodically to execute ready tasks. The flag processingRdyTasks is raised before executing any tick functions, and is lowered when the tick functions have all completed.

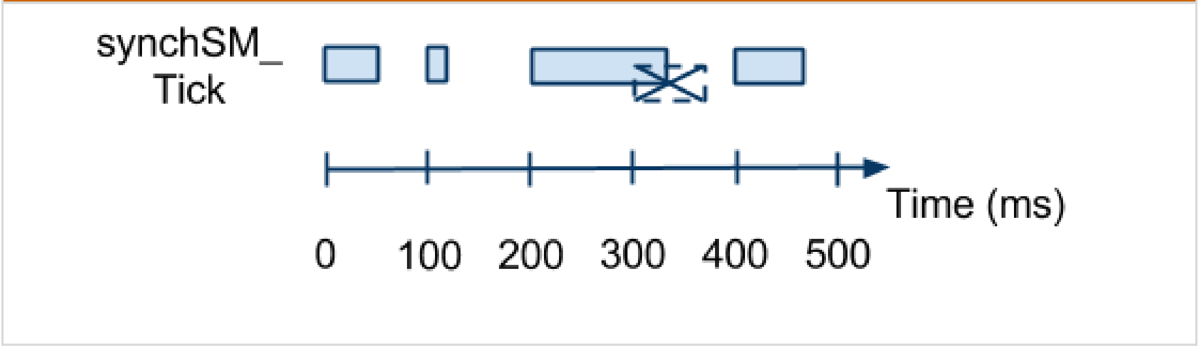
Figure 10.1.1: Scheduler in ISR with flag indicating that ready tasks are being processed.

```
unsigned char processingRdyTasks = 0;
void TimerISR() {
  unsigned char i;
  if (processingRdyTasks) {
     printf("Period too short to complete tasks.\r\n");
     return;
  }
  processingRdyTasks = 1;

  for (i = 0; i < tasksNum; ++i) { // Heart of the scheduler code
     if ( tasks[i].elapsedTime >= tasks[i].period ) { // Ready
        tasks[i].state = tasks[i].TickFct(tasks[i].state);
        tasks[i].elapsedTime = 0;
     }
     tasks[i].elapsedTime += tasksPeriodGCD;
  }
  processingRdyTasks = 0;
}
```

Thus, if TimerISR ever gets called with processingRdyTasks still being 1, this means that the microcontroller was still executing the synchSM's tick function when the next timer tick occurred. The situation on the microcontroller is illustrated by the following timing diagram.

Figure 10.1.2: Diagram showing microcontroller usage leading to timer overrun.



The synchSM has a period of 100 ms. Every 100 ms, the tick function executes actions, whose durations vary depending on the current synchSM state. At time 200 ms, the tick function's actions required 120 ms, and thus the tick at 300 ms will not have an effect using our scheduler code. The next tick that will be noticed will be at 400 ms.

Automatically-detected incorrect execution is known as an ***exception***. We call the above a ***timer-overrun exception***. The programmer decides how to handle the exception depending on the application. The programmer may output an error message or turn on an output "error" LED. The programmer may modify the system by lengthening SM periods, decreasing SM actions, decreasing the number of SMs, etc. In some systems, the programmer may have the system automatically restart itself. In other systems, the programmer may choose to ignore the exception, meaning certain ticks would be skipped.

Participation Activity 10.1.1: Detecting timer overrun within scheduler code.

Sort the various timer-overrun detection steps of the above scheduler code.

Detect if timer interrupt occurred while processing tasks; print error and return (1) ✓

Raise flag (means processing tasks) (2) ✓

Process tasks (3) ✓

Lower flag (means no longer processing tasks) (4) ✓

Reset

Participation Activity 10.1.2: Timer overrun.

#	Question	Your answer
1	The fact that a task's actions take non-zero time to execute may contribute to timer overrun. ✓ In any real implementation of a task on a processor, the instructions carrying out actions will take some non-zero time to execute.	[✓ True] False
2	The scheduler code itself (the for loop, if statements, etc.) also take some time to execute and thus may contribute to timer overrun. ✓ The scheduler's instructions do take non-zero time also. A good scheduler is written efficiently to minimize its own execution time.	[✓ True] False
3	The larger the period of a task, the more likely is timer overrun. ✓ A larger period means there is more time for the task's actions to execute before the next task tick, so timer overrun is less likely.	True [✓ False]
4	If a task has many actions, the scheduler spends most of its time in the function call tasks[i].TickFct(tasks[i].state). ✓ Most of the scheduler code just determines whether a task is ready to tick, in which case the scheduler calls the task's tick function and waits for that function to return. That waiting is where most time actually passes.	[✓ True] False

10.2 Utilization