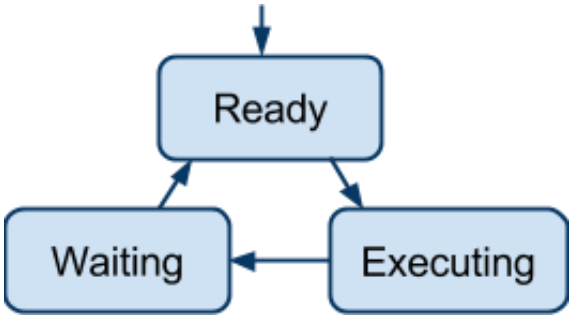
10.5 Jitter

10.6 Scheduling

A task implemented on a microcontroller can have one of three statuses at a given time:

- *Waiting*: The task should not execute because it is waiting for its period to elapse.
- *Ready*: The task's period has elapsed and the task is ready to execute, but the microcontroller hasn't yet started executing the task.
- *Executing*: The microcontroller is executing the task; for a synchSM, the microcontroller is executing the synchSM's tick function.

Figure 10.6.1: Task statuses.



By convention, all tasks are initially considered ready when the system is started (namely, at time 0). Thus, each task should execute once at startup, and then wait until its period elapses to execute again.

Participation Activity 10.6.1: Task status.

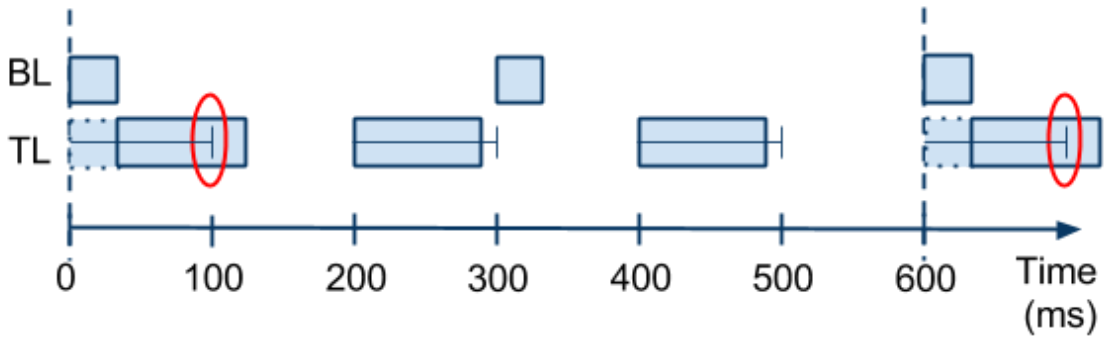
#	Question	Your answer
1	A ready task is waiting for the task's period to elapse. ✔ Ready means the period has already elapsed and the task is now ready to execute.	True **False**
2	A waiting task is ready but waiting to execute on the microprocessor. ✔ A waiting task is not ready to execute, but instead waiting for its period to elapse. Even if the microcontroller is available, a waiting task should not execute.	True **False**
3	Ideally, the moment a task becomes ready, the task would immediately begin executing. ✔ Immediate execution would eliminate jitter. But a microcontroller can only execute one task at a time, so this ideal commonly isn't achievable.	**True** False
4	A task status can change from ready to waiting. ✔ Once a task becomes ready, the task stays ready until the task executes.	True **False**

Scheduling is the job of choosing which of several ready tasks to execute. Scheduling is necessary because a microcontroller can only execute one task at a time. A ***scheduler*** is the code responsible for scheduling tasks. Scheduling does not impact utilization analysis, but does impact features of task execution, such as timer overrun (causing missed tasks), jitter, and meeting deadlines. Jitter was discussed earlier. We now discuss deadlines.

A task's ***deadline*** is the time by which a task *must complete after becoming ready*. Else, the system has "missed a deadline" and is considered to have not executed correctly. For example, in an example with tasks BL (period 300 ms, WCET 30 ms) and TL (period 200 ms, WCET 90 ms), suppose TL has a deadline of 100 ms. The period means that TL will be ready at time 0 ms, 200 ms, 400 ms, etc. The deadline means that when TL becomes ready at each of those times, TL should execute *and complete* by 100 ms, 300 ms, 500 ms, etc. We can include the deadline on a microcontroller usage diagram by drawing a line from the ready time to the deadline time, as shown below. If a programmer does not specify a

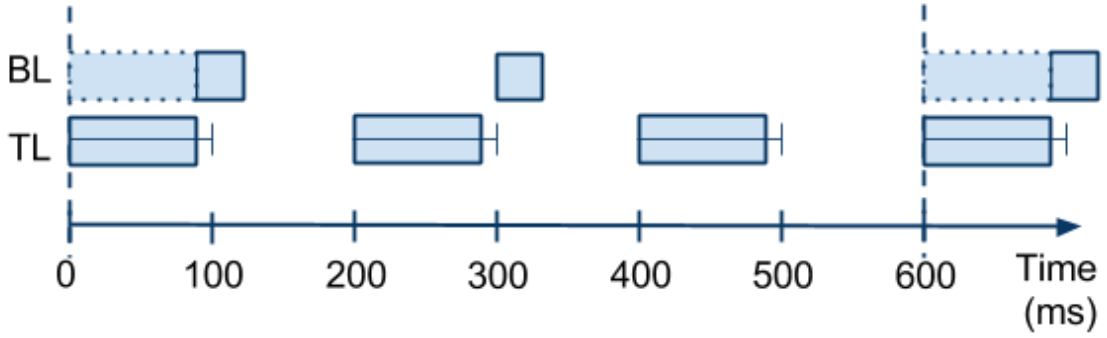
deadline for a task, then by default the deadline is equal to the period (and is usually not drawn), meaning the task should at least complete before the next time the task becomes ready. The diagram shows that task TL misses a deadline during the task's first execution of each hyperperiod, due to jitter caused by the scheduler executing BL before TL.

Figure 10.6.2: Missed deadline.



If the scheduler had executed TL before BL, the following execution would have resulted:

Figure 10.6.3: LedShow scheduled differently, no missed deadline.



Jitter in the second schedule is larger than in the first schedule, but no deadlines are missed. The example demonstrates the impact of scheduling on jitter and meeting deadlines.

Several scheduling approaches exist, and can be divided into static-priority and dynamic-priority approaches. ***Priority*** is an ordering of tasks indicating which ready task should execute first. When multiple tasks are ready, a scheduler executes the highest-priority task first.

A ***static-priority scheduler*** assigns a priority to each task before the tasks begin executing, and those priorities don't change during runtime.

- A common static-priority approach assigns highest priority to tasks with *shortest deadlines*. The intuition is that such tasks may miss deadlines if not

executed first, as in the above BL/TL example.

- If all tasks have their deadlines equal to their periods (as is often the case), then the above approach translates to assigning highest-priority to *shortest-period tasks*. This static-priority scheduling approach is called ***rate-monotonic scheduling (RMS)***, named because priorities are based on the task's period or rate, with priorities assigned in a monotonically-increasing manner according to those rates.
- Another static-priority approach assigns highest priority to shortest-WCET tasks. This approach may be better at reducing jitter, since first executing short tasks reduces jitter of other ready tasks. The approach requires knowledge of task WCETs, which aren't commonly known. The approach is useful if the microcontroller speed is such that missed deadlines are rare.
- Some schedulers allow a programmer to manually assign static-priorities to tasks. This approach is useful if the programmer knows that reducing jitter or meeting deadlines for certain tasks is more important than for other tasks. For example, a task controlling a car's speed may be more important than a task controlling a car's passenger-compartment air temperature, regardless of the periods, deadlines, or WCETs of those tasks.

Our earlier simple task scheduler code, replicated below, processes tasks in the order they appear in an array. Thus, that schedule gives *priority to tasks that appear earlier in the array.*

Figure 10.6.4: Task scheduler code.

```
for ( i = 0; i < tasksNum; ++i ) {
  if ( tasks[i].elapsedTime >= tasks[i].period ) { // task is ready
     tasks[i].state = tasks[i].TickFct(tasks[i].>state);
     tasks[i].elapsedTime = 0;
  }
  tasks[i].elapsedTime += tasksPeriodGCD;
}
```

A programmer can thus insert tasks into the array in order of priority.

A more general approach may introduce new fields to the task structure to store a deadline value, a WCET value, or a programmer-assigned priority number, and then a function can be called that sorts the tasks array by a particular field before the main code's `while (1)` loop.

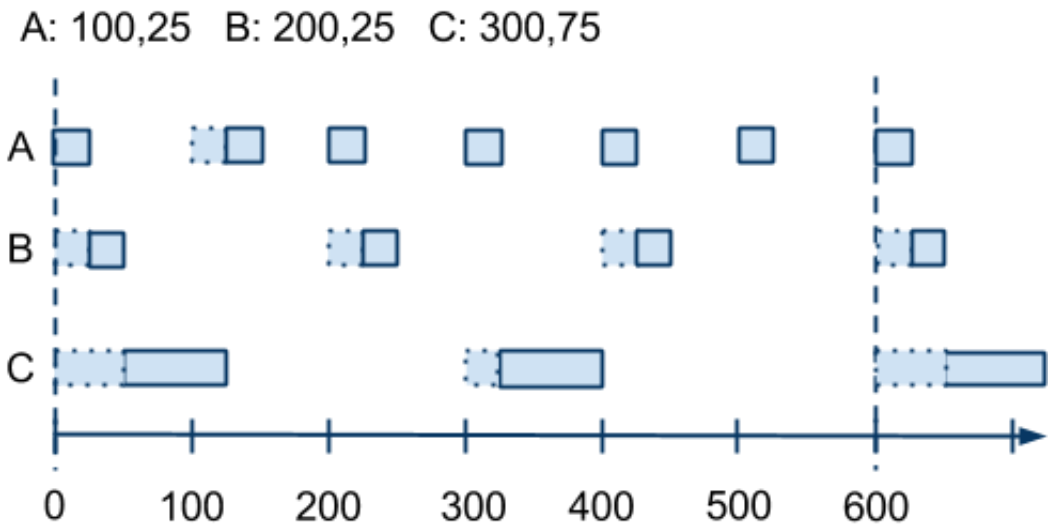
Try 10.6.1: Scheduler with sorting.

Modify the earlier-introduced scheduler code to use a programmer-assigned priority. This will include updating the task structure to include an unsigned char field named priority; having the user initialize that field when declaring a task; and sorting the tasks in the tasks array before entering main()'s while(1) loop. Write a simple routine for such sorting. Assume a higher unsigned char value means higher priority.

The earlier-introduced simple task scheduler code will experience timer overrun if a task is executing when the next timer tick occurs and another task should become ready, in which case the other task's execution may be missed entirely rather than just delayed. Such overruns will occur more often when tasks have larger execution times. A more advanced scheduler might maintain another list of ready tasks, where the list implements a ***priority queue*** (Wikipedia: Priority queue). When the timer ticks and timerISR is called, the scheduler code checks all tasks and adds any newly-ready tasks to the priority queue, where the queue keeps tasks ordered by their priority. Such a scheduler has the advantage of avoiding timer overruns; rather than missing task ticks, the scheduler adds those tasks to the list and will execute them later. The disadvantage is that such a scheduler requires more code and has more instruction overhead for each timer tick. The reader is encouraged to try implementing this more sophisticated scheduler.

Consider a system with three tasks A, B, and C, with "period, WCET" values as shown in the figure below, and deadlines equal to periods. Assume a priority-queue-based scheduler. Suppose A has highest priority, then B, then C (as would be the case using rate-monotonic scheduling). The figure shows that B experiences average jitter of 25 ms and C of (50+25)/2 = 37.5 ms, but no deadlines are missed (remember deadlines equal periods here).

Figure 10.6.5: Microcontroller usage diagram for a system with a priority-queue-based scheduler.



Note that at time 100 ms, the microcontroller was busy executing C when A became ready. The priority-queue-based scheduler does not miss that tick, but rather adds A to the priority-queue, and then executes A when C completes.

Participation Activity 10.6.2: Scheduling.

Given the above three tasks A, B, and C. Assume A is first in the tasks array (so has highest priority), then B, then C. Using the earlier-introduced basic scheduler (with no priority queue), draw the microcontroller usage diagram before answering these questions.

#	Question	Your answer
1	What task starts executing at time 0 ms? Valid answers: A, B, C, or None ✔ A A, B, and C are all ready at time 0 ms. A has highest priority so starts executing.	A
2	At what time does B start executing? ✔ 25 All tasks are ready at 0 ms. The array's first task A is checked and determined	25 ms

	ready, so A starts at 0 ms, and executes for 25 ms. Then, the array's next task B is checked and determined ready, so B starts at 25 ms.	
3	After B ends at 50 ms, the array's next task C is checked and determined ready, so starts executing at 50 ms. At what time does task C end execution? ✔ 125 C starts at 50 ms and takes 75 ms to execute, so ends at 125 ms.	125 ms
4	What task starts executing at time 125 ms? Valid answers: A, B, C, or None. ✔ None The basic scheduler ignores any timer tick that occurs while a task is executing; the ISR notes the processing flag is high, and returns. The situation is known as timer overrun.	None
5	What task starts executing at time 200 ms? Valid answers: A, B, C, or None. ✔ A C completed at 125 ms, so no task is executing at 200 ms. The timer ticks again at 200 ms, causing the scheduler code to check the first array item A, which is determined to be ready. So A starts executing at 200 ms.	A
6	Comparing to above microcontroller usage diagram using a priority-queue-based scheduler, at what time does the first difference in task execution occur? ✔ 125	125 ms

	C ended execution at 125 ms. The priority-queue-based scheduler would then pop task A from the queue and start executing that task at 125 ms.	
7	With the priority-queue-based scheduler, do any ticks of A get skipped? Answer yes or no. ✔ No The timer ticks every 100 ms; if C is executing, A gets pushed onto the queue, and then executes after C ends.	No
8	With the priority-queue-based scheduler, does A experience any jitter? Answer yes or no. ✔ Yes At time 100 ms, A is pushed onto a queue because C is executing. At 125 ms, C ends and A starts. A thus experiences jitter of 25 ms. The pattern repeats every 600 ms.	Yes

Try 10.6.2: Schedule several tasks.

Schedule the above tasks A, B, and C assuming the priority order C, B, A (C has highest priority, A lowest). Indicate whether any deadlines are missed, and list average jitter per task.

A ***dynamic-priority scheduler*** determines task priorities as the program runs, meaning those priorities may change. A common dynamic approach assigns ready tasks with the *earliest deadlines* the highest priority. The intuition is that ready tasks with the nearest deadline are more likely to miss the deadline if not executed first. The approach is known as ***earliest deadline first*** (***EDF***). Dynamic-priority schedulers may reduce jitter and missed deadlines, at the expense of more complex scheduler code. EDF scheduling is commonly

considered when some tasks are triggered by events (discussed in another section) rather than all tasks being periodic.

10.7 Preemptive scheduler