MSE | MASTER OF SCIENCE IN ENGINEERING

TSM_AdvEmbSof

Scheduling for Embedded Systems (Part I)

Serge Ayer | 12.10.2023 | Cours MSE

Models of Embedded Systems

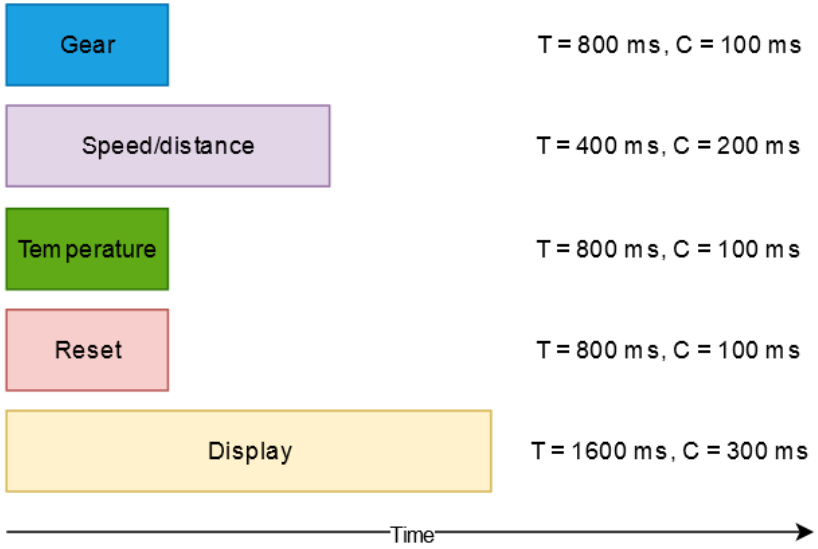
- An embedded system is usually designed for a specific applications
  - Monitor data from sensors or peripherals
  - Generate signals to control internal or external devices
  - Communicate data to the outside world
- Historically, programs of embedded systems have been written using a simple structure such as:
  - Super-loop
  - Event-driven
- However, embedded systems are becoming more complex and need to cope with the ever increasing demand for additional functionality
  - Edge computing is becoming common
  - Communication introduces another level of complexity/interaction
- Modern embedded systems require more powerful software and specifically require scheduling of tasks

Bike computer example

- Consider a bike computer with the following functions:
  - Bike gear: a value that the bike computer can read.
  - Wheel rotation: with a sensor, counts the number of wheel rotations.
  - Reset: with a button, reset timer and counters.
  - Display: display the information on a LCD screen.
- In the following, we will investigate how to implement the bike computer program:
  - Implement different scheduling algorithms
  - Evaluate some scheduling performances
  - Implement the program with full RTOS support

Bike computer tasks

- Gear: read current gear from the gear system.
- Speed/distance: read pedal rotation time and traveled distance.
- Temperature: read temperature.
- Reset: check button for resetting the computer.
- Display: Update LCD with new information.

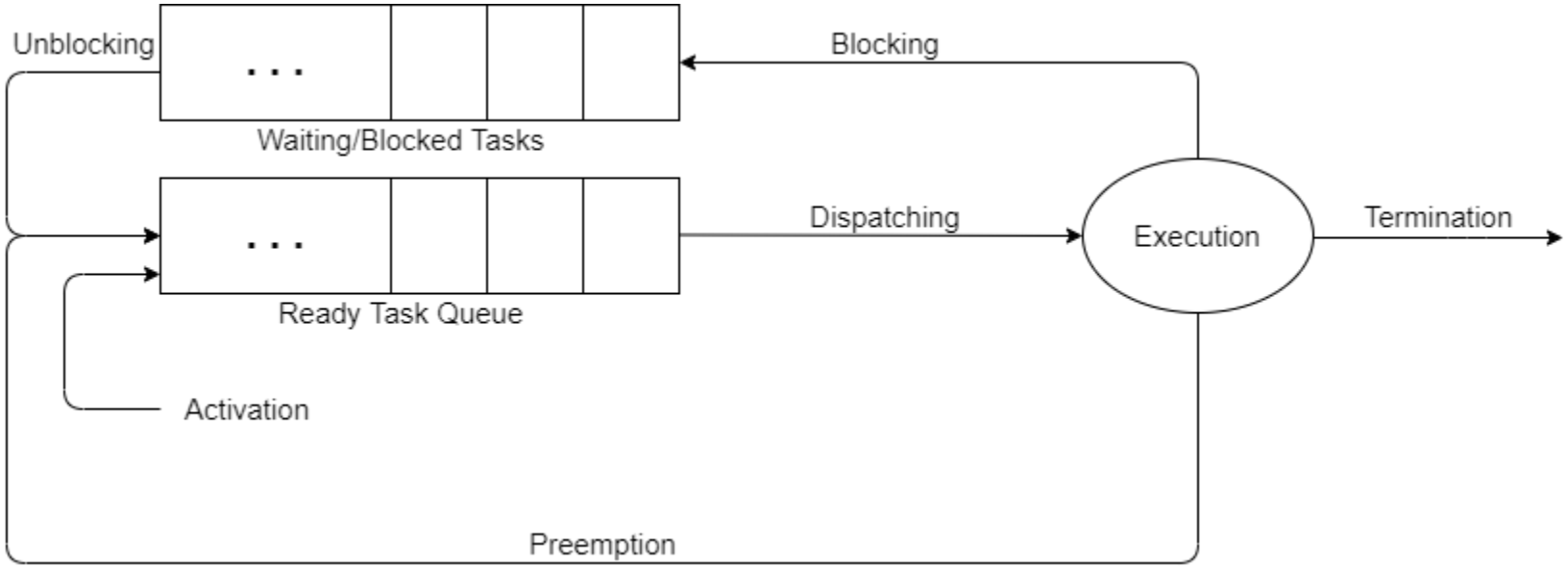


BikeComputer Tasks

Scheduling Concepts

- Assume that
  - We have one or more processing units;
  - Different tasks to execute;
  - Tasks are concurrent (can overlap in time);
- The CPU has to be assigned to the various tasks according to some criteria
- Scheduling algorithm = set of rules that, at any time, determines which task is executed.
- Dispatching = allocation of the CPU to the task selected by the scheduling
- A task has, at any time, a specific state (ready/running/waiting).
- We will consider scheduling only for systems with one processing unit.

Scheduling concept



Task Preemption

- Suspend a running task and insert it in the ready queue
  - Tasks may be executed in disjointed time intervals

Task A runs to completion, Task B starts and is preempted by Task C

- Important because
  - Exception handling/interrupt may need to preempt existing tasks
  - Schedule tasks based on their priority/importance
  - Improve efficiency (higher processor utilization)
- Introduces a runtime overhead
  - Should also be limited on a real-time system

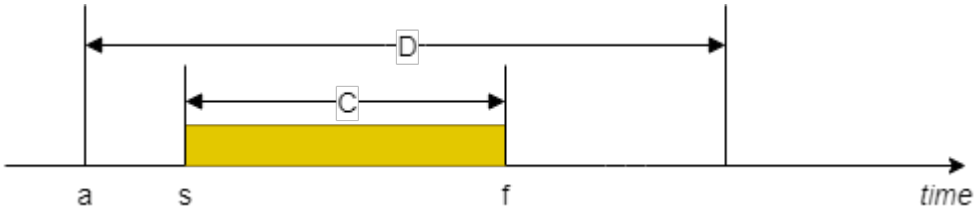
Scheduling Constraints

- Scheduling must account for different types of constraints
  - Timing constraints
  - Precedence constraints
  - Resource constraints
- Very often, constraints are combined
  - Timing and resource constraints

Timing Constraints

- The deadline is an important concept
  - Time before which a task should complete its execution
  - Relative: specified with respect to the task arrival time (as in diagram above)
  - Absolute: specified with respect to time zero
- Real-time tasks are often distinguished in three categories
  - Hard real-time: missing the deadline cause catastrophic consequences on the system.
  - Firm real-time: missing the deadline does not cause any damage to the system but the output of the task has no value.
  - Soft real-time: missing the deadline has still some utility for the system, although causing a performance degradation.

Timing Constraints



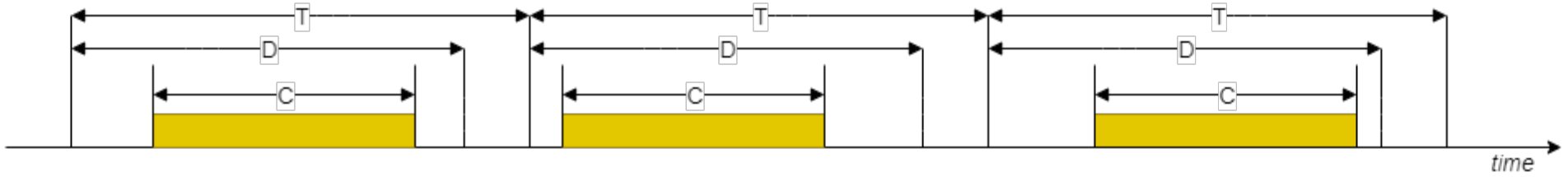
- Deadline (D)
- Arrival/release time (a or r): time at which a task becomes ready for execution
- Start time (s): time at which a task starts its execution
- Finishing time (f): time at which a tasks finishes its execution
- Computation time (C): time necessary to the processor for executing the task (without interruption)
- Criticality: consequences of missing the deadline
- A set of tasks is said to be schedulable if all tasks can finish within their deadlines.

Task Classification by Release Rate

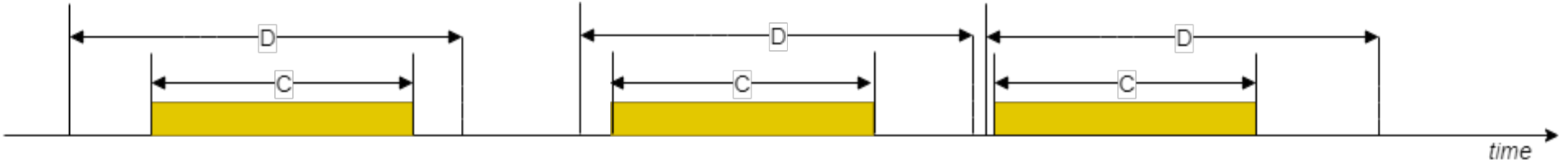
- Periodic or time-driven tasks:
  - Arriving at fixed frequency (defined by the period T)
  - Often T = D
  - For instance for sensory data acquisition or system monitoring
- Aperiodic or event-driven tasks:
  - Generated by interrupts
- Sporadic tasks
  - Aperiodic tasks with minimum inter-arrival time
- Background tasks
  - Non real-time tasks, accomplished only if CPU time is available

Periodic vs. Aperiodic Tasks

- Periodic tasks



- Aperiodic tasks



Precedence Constraints

- Precedence relations among tasks
  - Known at design time
  - Often represented through a directed acyclic graph
- Example:
  - Stereo vision system
  - Some tasks (e.g. image capture and pre-processing) can be executed in parallel for each camera.
  - Some other tasks must wait for the result of a preceding task (e.g. pre-processing must wait for image capture, edge detection must wait for pre-processing).
- The graph representing the tasks dependencies defines scheduling constraints

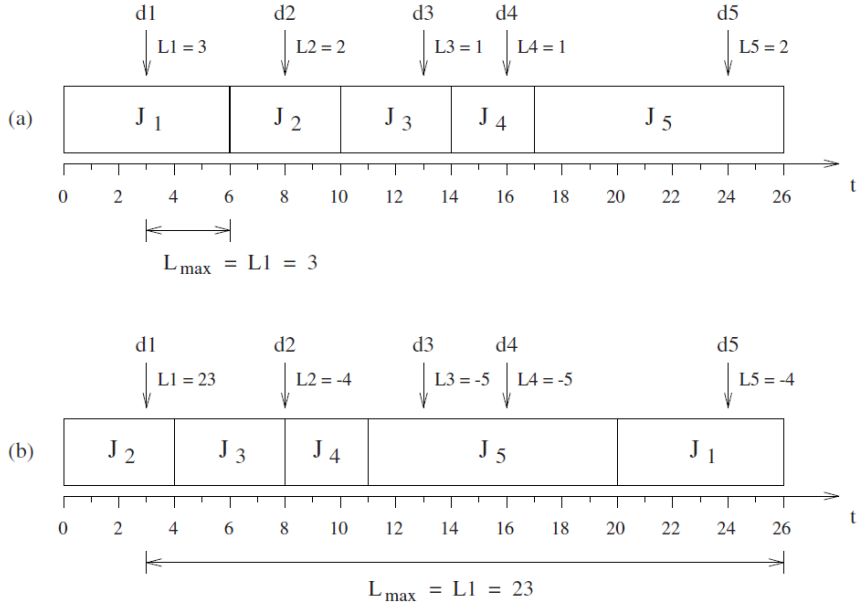
Resource Constraints

- Resource = any software structure that can be used by a task during its execution.
  - A variable
  - A memory area
  - A peripheral device
- Resources can be private (used by a single task) or shared (used by several tasks).
- Many shared resources do not allow simultaneous access by competing tasks and require mutual exclusions
  - A critical section is a piece of code executed under mutual exclusion
  - Most OSs provide synchronization mechanisms for creating critical sections and for synchronizing tasks that need to access shared resources

Metrics for Performance Evaluation

- Average Response Time
  - Time difference between the time a task is finished and the time a task is released
  - Task response time = $f - a$ or $f - r$
- Maximum Lateness
  - Defined as $max_i\ (f_i - d_i)$
- Maximum number of late tasks
  - $N_{late} = \sum_{i=1}^{n} miss(f_i)$
  - $miss(f_i) = \begin{cases} 0 & if\ f_i \leq d_i \\ 1 & otherwise \end{cases}$
- Processor Utilization Factor

  Fraction of processor time spent in the execution of a task set



Classification of Scheduling Algorithms

- Preemptive vs. Non-preemptive
  - Preemptive = the running task can be interrupted at any time.
  - Non-preemptive = the running task is executed until completion.
- Static vs. Dynamic
  - Static = scheduling based on fixed parameters assigned to tasks before their activation.
- Off-line vs. Online
  - Off-line = scheduling is computed before tasks are activated, stored in a schedule table and simply executed by a dispatcher.
  - Online = scheduling is computed every time a new task is activated, preempted or terminated.
- Optimal vs. Heuristic
  - Optimal = minimizes a given cost function or implements a feasible schedule
  - Heuristic = does not guarantee that the scheduling is optimal

Periodic Task Scheduling

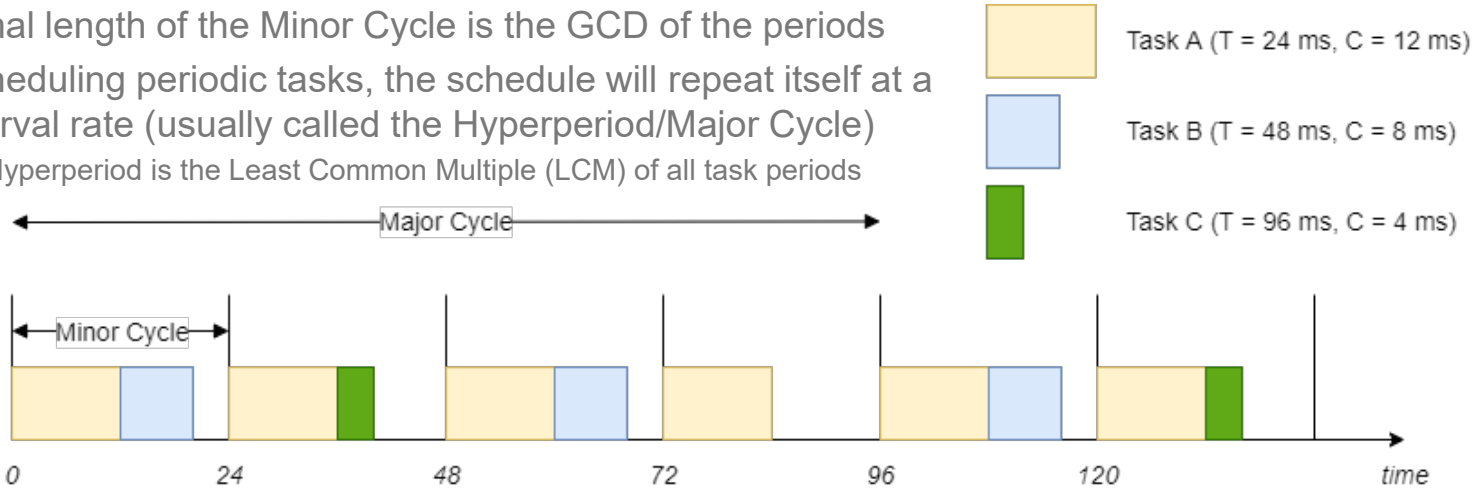
- Scheduling must guarantee that each task is activated at the proper rate and is completed within its deadline.
- Four basic most-known algorithms:
  - Timeline Cyclic Scheduling
  - Rate Monotonic
  - Earliest Deadline First
  - Deadline Monotonic
- When evaluating those algorithms, usually some assumptions are made:
  - The period of each tasks is constant
  - The worst-case execution time is constant
  - The deadline of each task is the same as its period
  - All tasks are independent and no precedence/resource constraint exists
- A set of tasks is said to be schedulable if all tasks can finish within their deadlines.

Schedulability of Periodic Tasks

- Depends on Processor Utilization Factor
  - Defined as $\mathrm{U} = \sum_{i=1}^{n} \frac{C_i}{T_i}$
  - Can be increased by increasing task computation times or by decreasing their periods
- There exists a maximum value of U below which a set of tasks is schedulable and above which it is not schedulable.
- This maximum value depends on the set of tasks AND on the scheduling algorithm

Timeline Cyclic Scheduling

- Very often used for handling periodic tasks
- Method:
  - Divide the temporal axis into slots of equal length (length = Minor Cycle)
  - Allocate one or more tasks in each time slot
  - At the beginning of each time slot, dispatch the tasks
- The optimal length of the Minor Cycle is the GCD of the periods
- When scheduling periodic tasks, the schedule will repeat itself at a given interval rate (usually called the Hyperperiod/Major Cycle)
  - The Hyperperiod is the Least Common Multiple (LCM) of all task periods



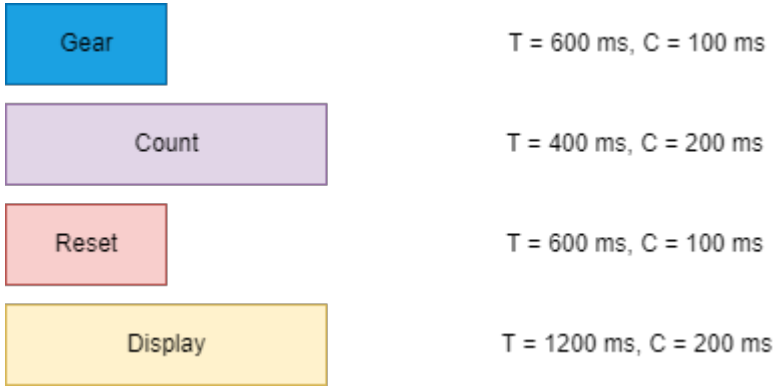
Timeline Cyclic Scheduling Feasibility

- If tasks cannot be split into sub-tasks, then a set of tasks is schedulable if the sum of execution times within each time slot is less or equal to the Minor Cycle
- Example:

$$\begin{cases} C_A + C_B \leq 24\,ms \\ C_A + C_C \leq 24\,ms \end{cases}$$

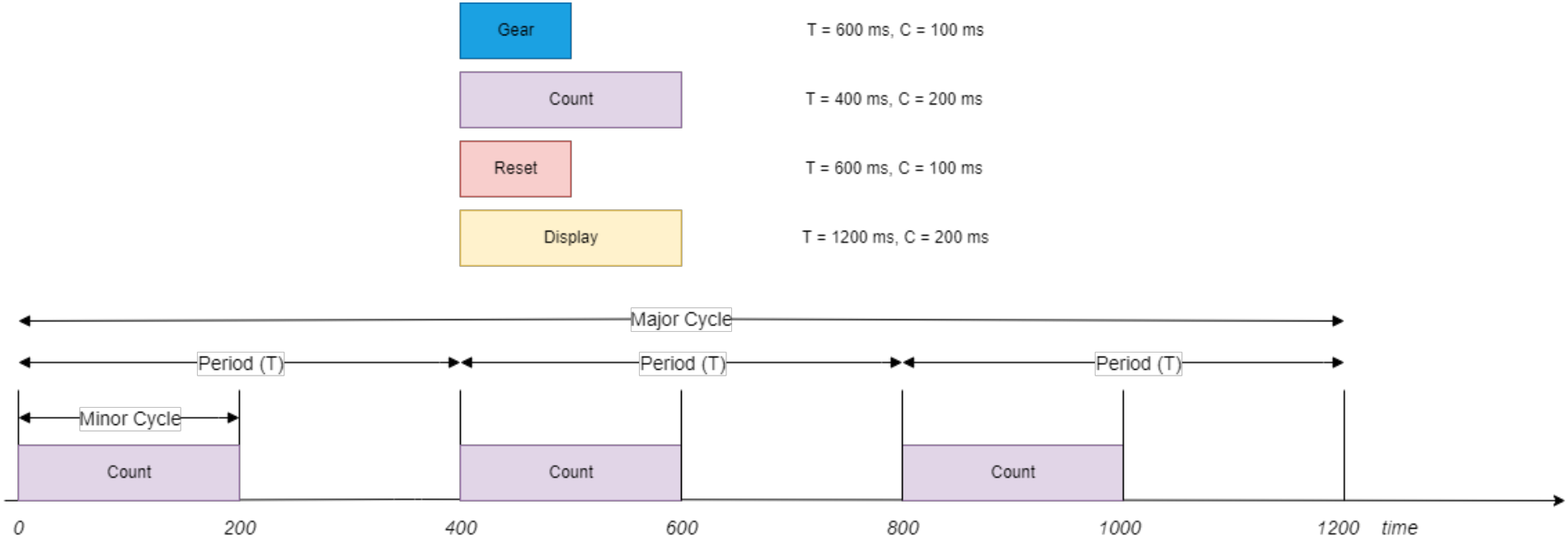
- If tasks can be split into sub-tasks, then a set of tasks is schedulable if the sum of execution times is less or equal to the Major Cycle

Timeline Cyclic Scheduling

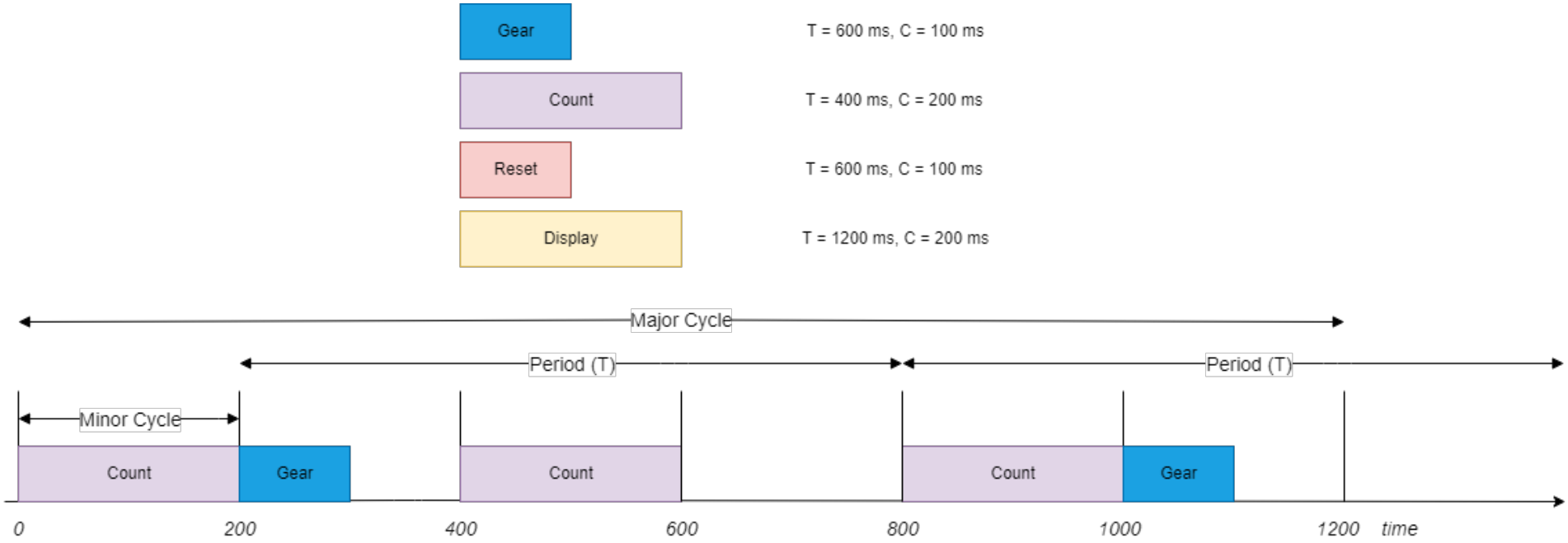


- The Minor Cycle is 200 ms, the Major Cycle is 1200 ms
- Is this set of tasks schedulable ?
  - Does the sum of computing times over a Major Cycle fit ?
  - G: 2 x 100 ms = 200 ms, C: 3 x 200 ms = 600 ms, R: 2 x 100 ms = 200 ms, D: 1 x 200 ms = 200 ms
  - Total computing time is 1200 ms (U = 1)

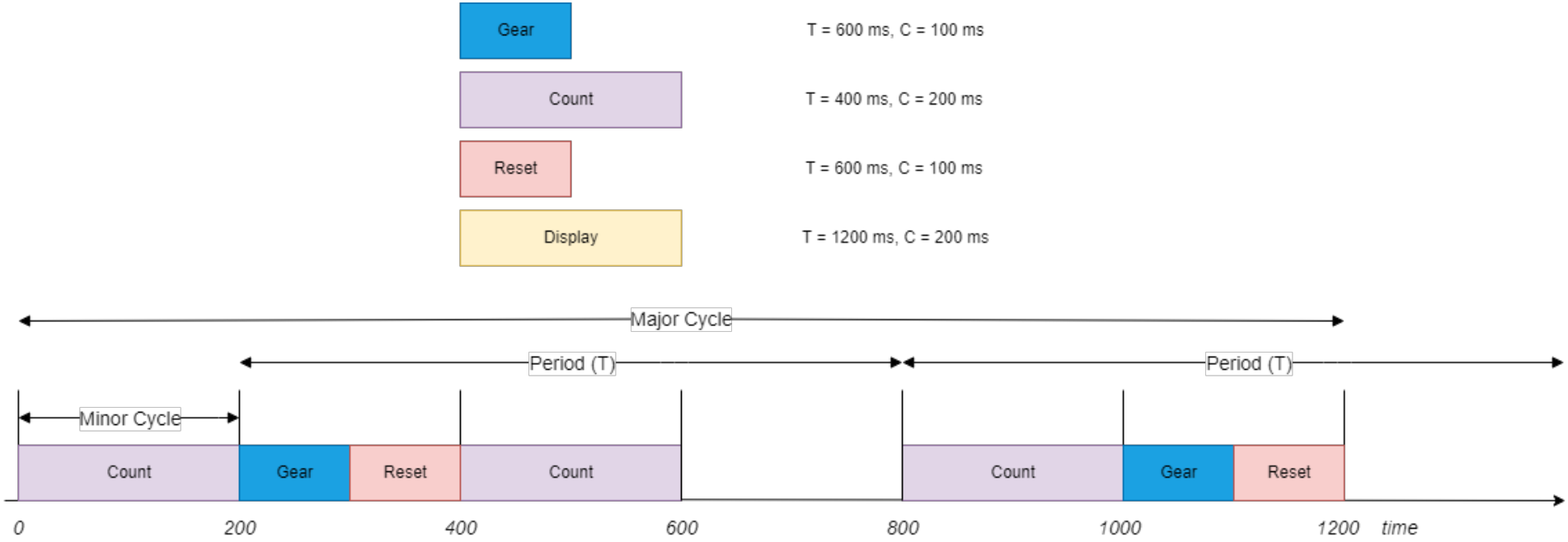
Timeline Cyclic Scheduling



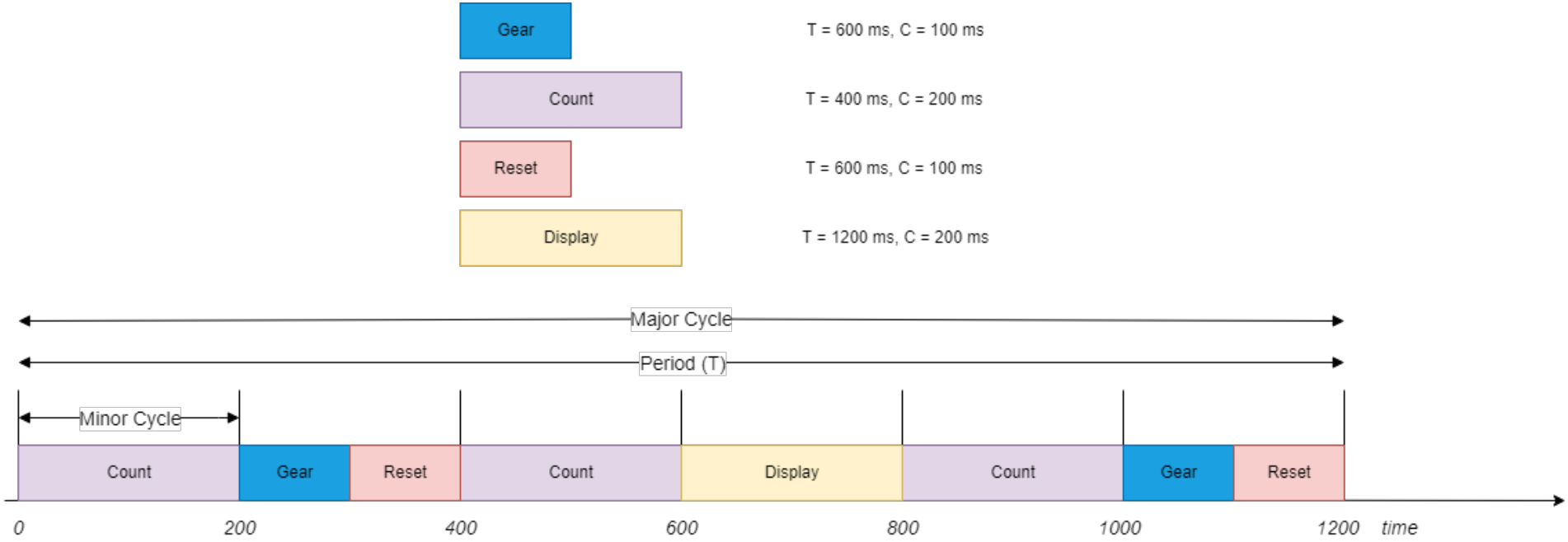
Timeline Cyclic Scheduling



Timeline Cyclic Scheduling



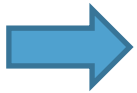
Timeline Cyclic Scheduling



Timeline Cyclic Scheduling

- The tasks in the timetable are run in a super-loop
- Recall: all tasks are considered to be periodic
  - Event-based tasks are turned to tasks with polling
- A super-loop is a program structure composed of an infinite loop, with all the tasks of the system contained in the loop, with the general form

```
main() {
  system_initialization();
  while (true) {
    check_device_status();
    process_device_data();
    output_response();
  }
}
```



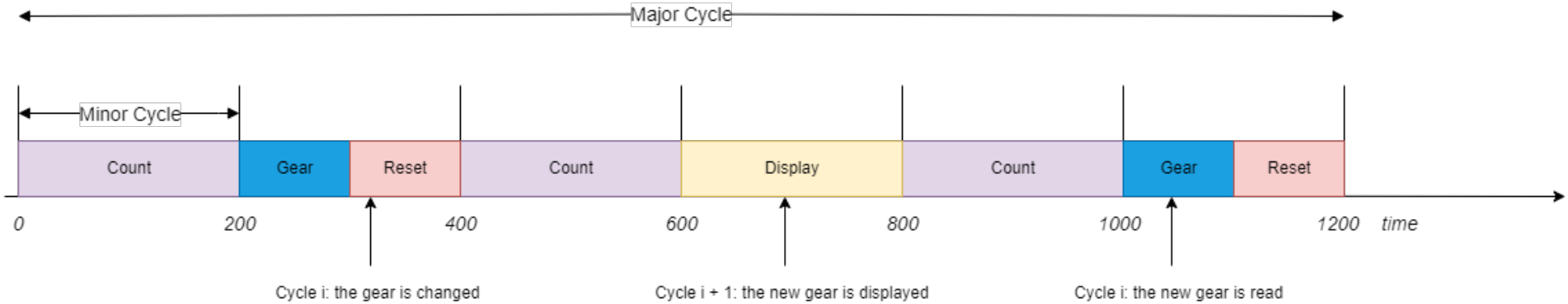
```
main() {
  system_initialization();
  while (true) {
    gear();
    count();
    display-1();
    reset();
    count();
    ...
  }
}
```

Timeline Cyclic Scheduling - Super-loop

- Advantages:
  - Easy to implement, predictable
- Disadvantages:
  - May be difficult to build the time table, that can be very large.
    - Tasks may be split in multiple sub-tasks.
    - All tasks run at the same rate or changing rates requires adding extra calls to functions.
  - Always run the same schedule
    - Regardless of changing conditions and relative importance of tasks.
  - The maximum delay is the Major Cycle time
    - Thus the polling rate is limited by 1/maximum delay.
  - The program must continually check the status of every device
    - Even if the device status has not changed or is not ready.
    - This wastes a lot of CPU time and causes excessive power consumption.
  - This approach scales badly:
    - It is very difficult to build a system with multiple activities/events that can respond quickly and the system's response time depends on all other processing that it has to do.

Timeline Scheduling Response Time

- What if the current gear is changed right after Reset starts?



- Delays !
  - Have to wait for a task period before we may read the gear again.
  - Have to wait for more than one full cycle before the new gear is displayed.
  - Although it may be considered as acceptable in this particular case, it is easy to think of scenarios where such a behavior would be problematic.

Our Bike Computer Implementations

- The implementations are accomplished in the multiple parts of the Bike Computer codelab, first as a single-task program (no context switching)
- Part 1: (without context switching)
  - Implement the classes representing the different tasks
  - Implement a timeline/cyclic static scheduling of tasks
- Part 2: (without context switching)
  - Implement the mechanism for event-driven handling of non periodic tasks
- Part 3: with multiple threads and context switching
  - Implement periodic tasks in multiple threads
  - Implement several scheduling mechanisms for periodic tasks